Universität Stuttgart

Automated Generation of Tailored Load Tests for Continuous Software Engineering

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von
Henning Schulz
aus Münster

Hauptberichter:	Dr.-Ing. André van Hoorn
1. Mitberichter:	Prof. Dr. Stefan Wagner
2. Mitberichter:	Prof. Dr. Petr Tůma
Tag der mündlichen Prüfung:	15. Dezember 2020

Institut für Software Engineering

2021

Abstract

Context: Continuous software engineering (CSE) aims to produce high-quality software through frequent and automated releases of concurrently developed services. By replaying workloads that are representative of the production environment, load testing can identify quality degradation under realistic conditions. The literature proposes several approaches that extract representative workload models from recorded data. However, these approaches contradict CSE's high pace and automation in three aspects: they require manual parameterization, generate resource-intensive system-level load tests, and lack the means to select appropriate periods from the temporally varying production workload to justify time-consuming testing.

Objective: This dissertation addresses the automated generation of tailored load tests to reduce the time and resources required for CSE-integrated testing. The tailoring needs to consider the services of interest and select the most relevant workload periods based on their context, such as the presence of a special sale when testing a webshop. Also, we intend to support experts and non-experts with a high degree of automation and abstraction.

Method: We develop and evaluate description languages, algorithms, and an automated load test generation approach that integrates workload model extraction, clustering, and forecasting. The evaluation comprises laboratory experiments, industrial case studies, an expert survey, and formal proofs.

Results: Our results show that representative context-tailored load tests can be generated by learning a workload model incrementally, enriching it with contextual information, and predicting the expected workload using time series forecasting. For further tailoring the load tests to services, we propose extracting call hierarchies from recorded invocation traces. Dedicated models of evolving manual parameterizations automate the generation process and restore the representativeness of the load tests. Furthermore, the integration of our approach with an automated execution framework enables load testing for non-experts. Following open-science practices, we provide supplementary material online.

Conclusion: The proposed approach is a suitable solution for the described problem. Future work should refine specific building blocks the approach leverages. These blocks are the clustering and forecasting techniques from existing work, which we have assessed to be limited for predicting sharply fluctuating workloads, such as load spikes.

Zusammenfassung

Kontext: Die kontinuierliche Softwareentwicklung hat zum Ziel, qualitativ hochwertige Software durch häufige und automatisierte Releases parallel entwickelter Dienste zu produzieren. Indem sie für die Produktivumgebung repräsentatives Lastverhalten wiedergeben, können Lasttests Qualitätsminderungen unter realistischen Bedingungen erkennen. In der Literatur werden verschiedene Ansätze vorgeschlagen, die repräsentative Lastmodelle aus aufgezeichneten Daten extrahieren. Diese Ansätze stehen jedoch in drei Aspekten im Widerspruch zu dem Tempo und der Automatisierung der kontinuierlichen Softwareentwicklung: Sie erfordern eine manuelle Parametrisierung, erzeugen ressourcenintensive Lasttests auf Systemebene und verfügen nicht über die Mittel, aus der zeitlich variierenden Produktivlast geeignete Zeiträume auszuwählen, die langlaufendes Testen rechtfertigen.

Ziel: Diese Dissertation befasst sich mit der automatisierten Generierung maßgeschneiderter Lasttests, um die erforderliche Zeit und Ressourcen für das in die kontinuierliche Softwareentwicklung integrierte Testen zu reduzieren. Bei dem Maßschneidern sind die Dienste von Interesse zu berücksichtigen und die relevantesten Lastzeiträume auf der Grundlage ihres Kontexts auszuwählen, wie z. B. der Sonderverkaufsaktion eines Webshops. Außerdem sollen Nutzer verschiedener Fachkenntnisse mit einem hohen Automatisierungs- und Abstraktionsgrad unterstützt werden.

Methode: Entwickelt und evaluiert werden Beschreibungssprachen, Algorithmen und ein automatisierter Ansatz zur Generierung von Lasttests, der die Extraktion von Lastmodellen, Clustering und Prognosen integriert. Die Evaluierung umfasst Laborexperimente, Industriefallstudien, eine Expertenbefragung und formale Beweise.

Ergebnisse: Unsere Ergebnisse zeigen, dass repräsentative, auf den Kontext zugeschnittene Lasttests generiert werden können, indem ein Lastmodell inkrementell erlernt, mit Kontextinformationen angereichert und das erwartete Lastverhalten mit Hilfe von Zeitreihenvorhersage berechnet wird. Für die weitere Anpassung der Lasttests an Dienste schlagen wir vor, Aufrufhierarchien aus aufgezeichneten Aufrufbäumen zu extrahieren. Dedizierte Modelle der sich verändernden manuellen Parametrisierungen automatisieren den Generierungsprozess und stellen die Repräsentativität der Lasttests her. Darüber hinaus ermöglicht die Integration des Ansatzes mit einer automatisierten Ausführungsumgebung Lasttesten für Nutzer mit geringen Fachkenntnissen. In Anlehnung an Open-Science-Praktiken wird ergänzendes Material online zur Verfügung gestellt.

Schlussfolgerung: Der vorgeschlagene Ansatz ist eine geeignete Lösung für das beschriebene Problem. Zukünftige Arbeiten sollten spezifische Bausteine verfeinern, auf die der Ansatz zurückgreift. Dabei handelt es sich um die Clustering- und Vorhersagetechniken aus bestehenden Arbeiten, die limitiert sind im Hinblick auf die Berechnung stark schwankender Lasten, wie z. B. Lastspitzen.

Publication List

Parts of this research have previously appeared in the following publications.

Peer-reviewed Journal Articles

- A. Avritzer, V. Ferme, A. Janes, B. Russo, A. van Hoorn, H. Schulz, D. Menasché, and V. Rufino (2020a). "Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-Based Approach Leveraging Operational Profiles and Load Tests." In: *Journal of Systems and Software* 165, p. 110564
- H. Schulz, A. van Hoorn, and A. Wert (2020c). "Reducing the Maintenance Effort for Parameterization of Representative Load Tests Using Annotations." In: *Journal of Software Testing, Verification and Reliability* 30.1

Peer-reviewed International Conference and Workshop Papers

- H. Schulz, D. Okanović, A. van Hoorn, and P. Tůma (2021). "Context-tailored Workload Model Generation for Continuous Representative Load Testing." In: *Proceedings of the 12th ACM/SPEC International Conference on Performance Engineering (ICPE 2021)*. To appear. ACM

- H. Schulz, T. Angerstein, D. Okanović, and A. van Hoorn (2019a). "Microservice-tailored Generation of Session-based Workload Models for Representative Load Testing." In: *Proceedings of the 27th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2019)*. IEEE Computer Society, pp. 323–335
- H. Schulz, D. Okanović, A. van Hoorn, V. Ferme, and C. Pautasso (2019c). "Behavior-Driven Load Testing Using Contextual Knowledge — Approach and Experiences." In: *Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE 2019)*. ACM, pp. 265–272
- A. Avritzer, D. S. Menasché, V. Rufino, B. Russo, A. Janes, V. Ferme, A. van Hoorn, and H. Schulz (2019). "PPTAM: Production and Performance Testing Based Application Monitoring." In: *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE 2019)*. ACM, pp. 39–40
- A. Avritzer, V. Ferme, A. Janes, B. Russo, H. Schulz, and A. van Hoorn (2018). "A Quantitative Approach for the Assessment of Microservice Architecture Deployment Alternatives by Automated Performance Testing." In: *Proceedings of the 12th European Conference on Software Architecture (ECSA 2018)*. Vol. 11048. Lecture Notes in Computer Science. Springer, pp. 159–174
- H. Schulz, T. Angerstein, and A. van Hoorn (2018). "Towards Automating Representative Load Testing in Continuous Software Engineering." In: *Companion of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018)*. ACM, pp. 123–126

Extended Abstracts and Posters

- H. Schulz and A. van Hoorn (2020). "Representative Load Testing in Continuous Software Engineering: Automation and Maintenance Support." In: *Software Engineering (SE 2020), Fachtagung Des GI-Fachbereichs Softwaretechnik*. Vol. P-300. LNI. Gesellschaft für Informatik e.V., pp. 149–150
- H. Schulz, A. van Hoorn, D. Okanović, S. Siegl, C. Heger, A. Wert, T. Angerstein, A. Hidiroglu, M. Palenga, C. Zorn, V. Ferme, and A. Avritzer (2019e). *ContinuITy: Automated Load Testing in DevOps*. Poster presented at the 10th ACM/SPEC International Conference on Performance Engineering (ICPE 2019) — ***Best Poster Award***

Acknowledgment

This work would not have been possible without the support of many people. First of all, I would like to thank Dr.-Ing. André van Hoorn for the outstanding supervision and enriching collaboration. I learned a lot from you, and I will always look back with gratitude on our many conversations and discussions. Likewise, I would like to thank Prof. Dr. Stefan Wagner, Prof. Dr. Petr Tůma, and Prof. Dr. Daniel Weiskopf for reviewing my dissertation and for a fair and constructive examination discussion.

I am also grateful to Novatec for giving me the chance to write my thesis in an industrial context. I would like to mention by name three people who particularly contributed to this opportunity: Stefan Siegl, Alexander Wert, and Christoph Heger not only created the organizational framework but also actively supported me as mentors. I can truly say that my work would not have been possible without you.

Likewise, I appreciate that I had the backing of two teams at once. On the one hand, the software quality and architecture group at the University of Stuttgart gave me great personal and scientific support. Thank you very much for all the group retreats, Ph. D. talks, trips, gatherings, and evenings in the wine cellar. On the other hand, I had the chance to be part of the DEDI team at Novatec. I am very grateful that you have always regarded me as a full member despite unequal projects and that I could profit from your different view on the subject. You have contributed significantly to the fact that my work became more than academic theory.

Also important to me are the scientific collaborations with many different people. First, many thanks to Tobias Angerstein, An Dang, Alper Hidiroglu, and Manuel Palenga for your excellent work in your Master's theses and student jobs. Your contributions are essential building blocks of my work. Second, I would like to thank the following researchers from all over the world for the fruitful collaborations we had: Dušan Okanović, for the active support throughout my work and for assisting me with your experience; Vincenzo Ferme, for the close cooperation during our theses; Petr Tůma, especially for your astute feedback; all the members of the ContinuITy project, to whose success I was privileged to contribute; and our research group led by Alberto Avritzer for broadening my horizon.

Last but not least, a very special thank you goes to my family for supporting me the whole time and to Sonja for always standing by me despite a hard time. I hardly can put my gratitude into words. In any case, your support was essential for me to be able to complete my work.

Contents

Chapter 1
Introduction

The need for tailored solutions is a commonly known challenge in computer science — the search query *"one size does not fit all"* submitted to the dblp computer science bibliography results in 63 hits.[1] In this dissertation, we face the tailoring of load tests. In the following, we motivate the problem (Section 1.1), define the research objectives and questions (Section 1.2), summarize our contributions (Section 1.3), and provide an overview of this document (Section 1.4).

1.1. Motivation and Problem Statement

Representative Load Testing: Load testing is a powerful means for evaluating a software application's behavior under a realistic workload (Jiang and Hassan, 2015). Being applied in a dedicated test environment, it can identify load-related functional and non-functional problems without interfering with the production environment. In doing so, load testing contributes to business success, e.g., by identifying revenue-decreasing system slowdowns (Kohavi and Longbotham, 2007) before delivery. The meaningful evaluation

[1]`https://dblp.uni-trier.de/search?q=one+size+does+not+fit+all` (visited on 07/16/2020)

of non-functional quality attributes, such as performance, requires a test workload that is carefully characterized, i.e., *representative* of the production environment (Ferrari, 1972). Otherwise, relevant issues might remain undetected (G. Jin et al., 2012). For this reason, several workload characterization approaches have been proposed to replay the resulting workload models in load tests. Calzarossa et al. (2016) summarize a commonly applied methodology, which comprises measuring the basic units of work, e.g., the user sessions, and building a workload model that characterizes these basic units, e.g., Markov chains representing the average user behavior (Menascé et al., 1999; Vögele et al., 2018).

Relevance to Research: While representative load testing is generally relevant, it faces both challenges and opportunities when being applied in today's industry standard of continuous software engineering (CSE). Opportunities arise from the neat collaboration of development and operation processes aiming at high software quality, known as *DevOps* (Bass et al., 2015). The operational monitoring eases the integration of the workload characterization methodology into the testing process (Eismann et al., 2020). As opposed to this, performance engineering tasks, including load testing, are perceived as challenging (Eismann et al., 2020) and often not applied systematically (Bezemer et al., 2019). The main reasons are the short release cycles using automated pipelines (Humble and Farley, 2010) and the architectural style of *microservices*, which denote loosely coupled services that are developed and operated by individual teams (Newman, 2015). The resulting frequent and concurrent releases conflict with the resource- and time-intensiveness of — particularly system-wide — load tests (T.-H. Chen et al., 2017).

State of the Art: The literature proposes numerous approaches that address different aspects of the described challenges. Various workload characterization approaches can extract workload models from recorded user sessions or requests (Barros et al., 2007; Cai et al., 2007; Krishnamurthy et al., 2006; Lutteroth and Weber, 2008; Menascé and Almeida, 2002; Ruffo et al., 2004; Vögele et al., 2018), which then can be replayed in load tests. As the extracted models typically require manual parameterization (Vögele et al., 2018) and, thus, are costly to (re-) generate, Syer et al. (2014, 2017) propose means for checking a load test's representativeness repeatedly. Works like the one by Ferme and Pautasso (2017, 2018) deal with automating the test execution lifecycle, while Versteeg et al. (2016) aim to reduce the resources required for the test execution by stubbing some of the tested services. Vögele (2018) proposes to use modeling techniques for selecting load test cases that most likely induce faults to the tested application. Finally, in the context of CSE, production testing techniques, such as canary releasing (Newman, 2015), have emerged, which test an individual service under (portions of) the real workload.

Limitations of Existing Approaches: These works lack a precise focus on the most relevant scenarios to fit into the limited time and resources CSE has for testing. On the one hand, the concurrent development of individual microservices demands load tests that directly target the respective service — accompanying service stubbing approaches. On the other hand, application workloads frequently change (Herbst et al., 2013), to which the testing needs to adapt. A significant influencer of workload changes is the context (Chandola et al., 2009). For instance, our evaluation with the student management system of a large university shows that different contexts, such as tuition, vacation, and course registration phases, significantly influence the number of active users and their behavior (see Chapter 14). Thus, the time and resources saved by prioritizing test scenarios relevant to the current or near-future context reinforce the already known necessity of choosing a suitable time frame of recorded data when extracting a workload model (Ferrari, 1972). Existing workload characterization approaches miss the means for automating this selection, and, additionally, require manual parameterization of each generated model. Production testing inherently uses the current workload, which, however, might not be generalizable to other scenarios, including the near future (Feitelson, 2002).

1.2. Objectives and Research Questions

For the reasons described in the previous section, we investigate how to generate representative load tests that are tailored to the context and services of interest. As the workload and its contexts vary, the interest may change over time, e.g., towards handling the course registration load before the semester starts as opposed to appropriate downscaling during the vacation phase. Because this evolving interest entails generating load tests repeatedly, we furthermore focus on automating the load test parameterization to achieve a load test generation process that integrates with CSE. Besides, the tailoring allows us to ease load testing for non-experts and, thus, to tackle the challenge of rarely adopted performance engineering techniques, which Bezemer et al. (2019) report.

Our research concentrates on session-based workloads (Menascé et al., 1999), i.e., workloads consisting of user sessions, each of which comprises one or multiple consecutive requests to the application under study. As most customer-faced web applications — including webshops, which are often the core of a business — manage sessions, this type of workload is particularly relevant for research.

Summarizing, we postulate and address the following research questions.

RQ1: How can load test parameterizations be evolved without manual intervention at test generation or execution time?

RQ2: How can representative load tests be tailored to specific services of a session-based application?

RQ3: How can representative load tests automatically be tailored to the contexts of a session-based workload?

RQ4: How can we leverage automated tailored load test generation and automated load test execution for enabling load testing for non-experts?

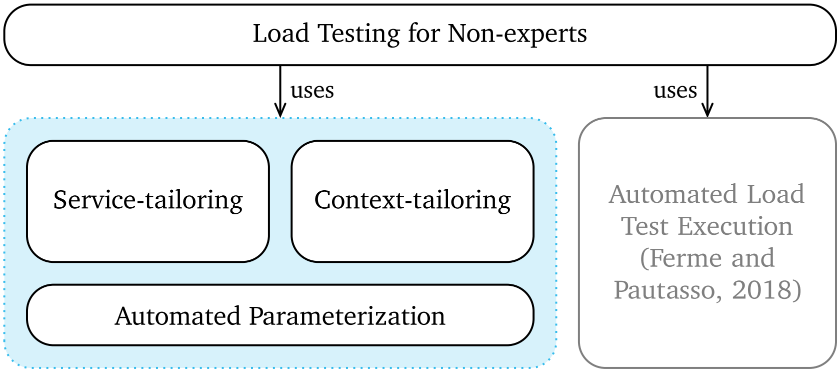

Load Testing for Non-experts
uses
uses
Service-tailoring
Context-tailoring
Automated Parameterization
Automated Load Test Execution (Ferme and Pautasso, 2018)

Figure 1.1.: Overview of the contributions of this dissertation. We do not contribute to automated load test execution, but leverage the work by Ferme and Pautasso (2018).

1.3. Overview of Contributions

Our contribution is a novel concept and process for generating tailored load tests automatically. As illustrated in Figure 1.1, we subdivide the contribution according to the research questions. First, we provide an approach to automating the load test parameterization. Based on that and the existing session-based workload characterization by Vögele et al. (2018), we introduce two respective approaches to the tailoring of generated load tests to particular services and contexts. In collaborative work, we leverage the automated load test execution engine by Ferme and Pautasso (2018) for easing load testing for non-experts. Also, we evaluate each of the introduced approaches, which we consider a separate contribution. As a final contribution, we provide implementation and evaluation replication artifacts online. The following sections depict these contributions.

1.3.1. Automated Load Test Parameterization

For the automated parameterization of generated load tests, we introduce a parameterization language for describing Input Data and Properties Annotations (IDPAs). An IDPA separates user-defined parameterizations from load tests or workload models. As an example, applications managing user accounts most likely have a login endpoint that requires providing a user name and a password as parameters. When a load test emulates a user's login process, it needs to specify credentials that fit the test environment's database, which typically differs from the production database, for security and privacy reasons (Jiang and Hassan, 2015). An IDPA can hold the credentials and automatically apply them to generated load tests.

For maximizing the degree of automation, we provide a feedback-based evolution strategy of IDPAs in case of typical API changes, as collected in the literature. Furthermore, we can extract parts of an IDPA from API specifications (OpenAPI Initiative, 2020) and transform IDPAs into different load test formats, such as JMeter (Apache Software Foundation, 2020[a]) and BenchFlow (Ferme and Pautasso, 2018). The result is a fully automated process for transforming production data, i.e., recorded user sessions, into an executable load test. All manual effort can be applied in advance.

1.3.2. Service-tailoring of Load Tests

According to the microservice architectural style, the teams test their developed services individually. Accounting for that, we introduce two alternative algorithms that extend the load test generation process from existing work (Calzarossa et al., 2016; Vögele et al., 2018) to tailor a generated load test to one or a set of services. Both algorithms base on collected traces (Okanović et al., 2016), which document how an individual request propagates through a distributed application. One of the algorithms modifies the initial artifact of the generation process, i.e., the recorded logs. The second algorithm modifies the generated workload model.

Service-tailored load tests smoothly integrate with service stubbing approaches, such as the work by Versteeg et al. (2016): our approach sends requests to the services under test, while the stubs receive the requests the services under test submit themselves. In combination, we can test a single service in isolation.

1.3.3. Context-tailoring of Load Tests

We split the established load test generation process into two parts when generating load tests tailored to particular contexts, such as the course registration phase of a student information system. First and continually, we build a workload model. This workload model reflects the evolving user behavior and the number of concurrently active users of a large time frame, e.g., one year. We learn the model by incremental clustering based on existing algorithms. Besides, we enrich it with contextual information, such as the semester phase.

Second and on-demand, we extract a load test from the workload model considering the contextual information. Using the introduced Load Test Context-tailoring Language (LCtL), a user describes a specific workload scenario, e.g., the sharpest spike during the course registration, and our approach automatically generates the load test. The scenario can lie in the future; in this case, we apply time-series forecasting. As opposed to existing approaches we base on, our predictions also consider the mix of differently behaving types of users.

1.3.4. Load Testing for Non-experts

In collaboration with Ferme and Pautasso (2017, 2018), we developed an approach that eases load testing for less experienced users. It leverages our automated generation of tailored load tests and Ferme and Pautasso's (2018) BenchFlow approach, which automates the test execution. We introduce the Behavior-driven Load Testing (BDLT) language, which allows users to describe both the load test scenario and its execution. Because it is based on natural language, the language constitutes a lower barrier for non-experts. Remarkably, participants of a case study we conducted highlighted the potential of the language to incorporate non-technical stakeholders, such as product owners, into the load testing process (see Chapter 15).

As a particular use case of the BDLT language, we describe an approach to the scalability assessment of microservice deployment alternatives, which we have developed in collaboration with Avritzer et al. (2018, 2020a). The BDLT language allows describing multiple load tests, which our approach generates and executes. From the results, we calculate a Domain-based metric, which supports the user in selecting optimal deployment alternatives.

1.3.5. Evaluation

We have qualitatively and quantitatively evaluated our approach in twelve studies. The studies comprise the following methods: laboratory experiment, case study, expert survey, estimation model, formal proof, and formal analysis.

In a nutshell, we found our approach reasonably suitable for generating tailored load tests automatically. We also showed that improper load test parameterization can lead to significantly distorted test results, which our approach prevents. The extensibility of our concepts and languages turned out to be a crucial feature in many cases. Proposing challenges to be addressed in future work, we also identified several limitations of existing approaches we used as building blocks. Mainly, the workload characterization, modeling, and forecasting techniques used need to be extended for the accurate prediction of strongly fluctuating workloads, such as load spikes.

1.3.6. Available Artifacts

For replication purposes, we provide several artifacts online. As summarized in Appendix E, we make the prototypical implementation of our approach available as open-source software, archived for permanent access (H. Schulz, 2020a; H. Schulz et al., 2020a; H. Schulz and Dang, 2020). Showing its capabilities of generating industrial-scale load tests, we use it in our evaluation. Also, we provide an interactive demonstration of selected features (H. Schulz, 2019). Finally, we publish replication packages as supplementary material for our evaluation (Avritzer et al., 2020b; H. Schulz et al., 2019b,d, 2020b, 2019f). These packages contain experiment setups, automation scripts, raw evaluation results, and analysis scripts, enabling researchers to re-execute and re-analyze our studies.

1.4. Document Structure

The remainder of this document is structured as follows.

- Part I provides foundations of this research, on which we base in the following.
    - Chapter 2 focuses on CSE, which forms the context we operate in.
    - Chapter 3 describes the established methodologies for workload characterization and forecasting. In our research, we extend this methodology, e.g., by adding automated parameterization and learning workload models incrementally.
    - In Chapter 4, we illustrate the concept of load testing. Particularly, we elaborate on representative load testing, to which this research contributes.
- Part II introduces our approach to the automated generation of tailored load tests.
    - Chapter 5 describes the research design we applied, consisting of the precise goal and research questions, assumptions we make, the research plan, an overview of the approach, and the collaborations we participated in.
    - In Chapter 6, we introduce the automated load test parameterization and the evolution of parameterizations over API changes.
    - Chapter 7 presents the algorithms for service-tailoring.
    - Chapter 8 provides the context-tailoring approach, including the incremental workload model learning.
    - Chapter 9 depicts our collaborative work on load testing for non-experts.
    - In Chapter 10, we present the implementation of our approach, which is publicly available (H. Schulz, 2020a; H. Schulz et al., 2020a; H. Schulz and Dang, 2020).

- Part III comprises the evaluation of the approach introduced in Part II, including the evaluation method, evaluation results, and discussion of the research questions.
  - In Chapter 11, we describe the evaluation design, provide an overview of all studies conducted, and introduce evaluation metrics we use.
  - Chapter 12 evaluates the automated load test parameterization approach, which consists of two experimental studies, a case study, and effort estimation models.
  - Chapter 13 provides a formal proof of the algorithms developed in Chapter 7 and an experimental evaluation of the service-tailoring.
  - In Chapter 14, we evaluate the context-tailoring approach at the subject of the student information system of a large university. We analyze the incrementally learned workload model, conduct a case study, and perform two experimental studies.
  - Chapter 15 concludes the evaluation with two case studies focussing on the BDLT language introduced in Chapter 9.
  - In Chapter 16, we discuss related work.
- Part IV concludes this research with a summary (Chapter 17) and a discussion of future work (Chapter 18).

The end matter holds the bibliography, lists of acronyms, figures, and tables, and an appendix. The appendix provides specific details of our approach, such as schemata and grammars of the languages we introduce and detailed examples (Appendices A to D). Besides, it summarizes the supplementary material available online for replication purposes (Appendix E).

Part I

Foundations

Chapter 2

Continuous Software Engineering

The software industry experiences an evolution and acceleration of software engineering, named continuous software engineering (CSE). Affecting the whole software lifecycle from planning to operations, CSE poses new challenges for load testing, which we address in our research. In this chapter, we introduce the foundations of CSE. First, we provide an overview in Section 2.1. Section 2.2 discusses the DevOps paradigm, i.e., the neat collaboration between development and operation practices. In Section 2.3, we present the microservice architectural style, which is often used in CSE contexts. Finally, in Section 2.4, we depict the concept of API specifications, which are relevant to our work. Section 2.5 summarizes the chapter.

2.1. Continuous *

CSE (Bosch, 2014) is a development mainly driven by the industry but also adopted by research. In general, it describes an acceleration of collaboration, e.g., delivering new software versions frequently whenever available as opposed to few releases per year. This acceleration spans multiple aspects

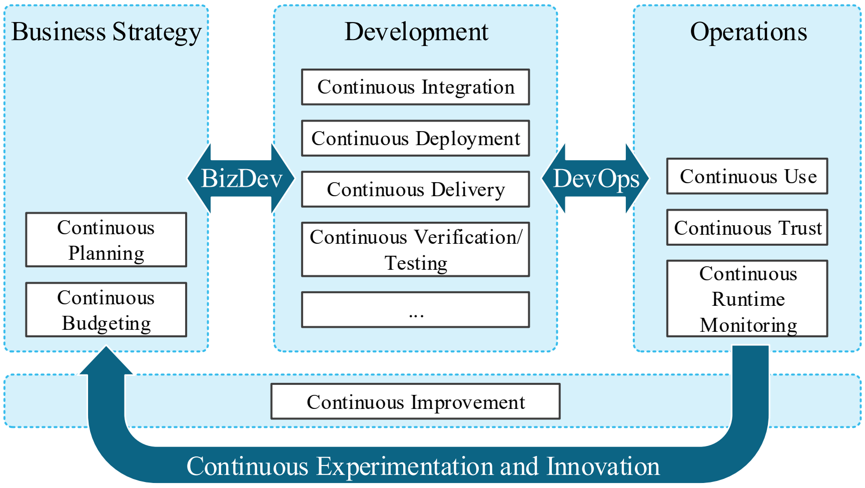


Figure 2.1.: The holistic vew of Continuous * (based on Fitzgerald and Stol, 2017).

of the software lifecycle. Fitzgerald and Stol (2017) use the notion of *Continuous* * (*"Continuous Star"*) as a *"holistic endeavor,"* affecting the main sub-phases of business strategy, development, and operations.

Figure 2.1 provides an overview of Continuous *. While each sub-phase has its continuous processes, the collaboration among them is crucial, named as *BizDev* and *DevOps* (Bass et al., 2015). Above all stands a new way of thinking, including agility (Fowler and Highsmith, 2001; Schwaber and Beedle, 2001) and continuous learning, as used in the Lean Startup method (Bosch et al., 2013). Related activities are continuous improvement, experimentation, and innovation. In the following, we explain selected continuous activities with particular relevance for our research. We will explain DevOps in Section 2.2.

2.1.1. Continuous Integration, Delivery, and Deployment

When developers complete implementing a new increment of the software, they commit it to a code repository. CSE aims at automating the handling of frequent code commits. The first step is the integration of the commit.

Definition 2.1 (Continuous Integration — Fowler, 2006)
Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily — leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

Continuous delivery and continuous deployment extend this definition to a *"holistic, end-to-end approach to delivering software"* (Humble and Farley, 2010). The difference between these extensions is that the latter automatically releases the software to clients, i.e., deploys it, while the former only creates the ability to deploy it (Fitzgerald and Stol, 2017). Humble and Farley (2010) argue that continuous delivery is better suited for many software systems, e.g., for applying manual testing. The core of all these processes is a pipeline, which is defined as below.

Definition 2.2 (Deployment Pipeline — Humble and Farley, 2010)
A deployment pipeline is, in essence, an automated implementation of your application's build, deploy, test, and release process.

To achieve fast feedback, this pipeline typically has multiple stages (Humble and Farley, 2010). The first stage reacts to the commit of a developer by compiling and analyzing the code and running unit tests. If these tests succeed, the code is assembled into binaries and automatically forwarded to the acceptance stage. There, longer-lasting acceptance tests are run. If they succeed, too, the pipeline subdivides into several branches, e.g., for running tests that require a specific environment, such as load tests, or deploying the code to production. These final steps do not need to be triggered automatically. Instead, it can be more feasible that the respective person responsible triggers the stage by clicking a button. For that, they should be provided with an overview of the previously executed stages and their outcome. Being triggered, the stages again need to run autonomously. For that, the test and production environments need to be autonomic, e.g., realized as a Cloud environment.

Remarkably, Humble and Farley use the term deployment pipeline, which, however, is not restricted to continuous deployment. To emphasize that, we use the generalized term *continuous integration and delivery (CI/CD) pipeline* in the remainder of this dissertation, meaning any pipeline that automates at least the build and testing.

2.1.2. Continuous Testing

A CI/CD pipeline already contains testing of the newly committed software version. However, it can also be beneficial to execute tests outside the pipeline. Continuous testing refers to the execution of unit tests in the background, aiming at providing early feedback to the developer (Demeyer et al., 2018; Saff and Ernst, 2004). An agile method that complements continuous testing is test-driven development (TDD) (Beck, 2003), which denotes creating test cases that describe the intended behavior of the software before implementing a new feature or bug fix.

Load testing, the focus of this work, is not suited for continuous testing, as it requires an isolated, production-like test environment to obtain reliable performance measures (Jiang and Hassan, 2015). Using the automated deployment capabilities of CI/CD, it should be instead executed in a dedicated pipeline stage at a reasonable time.

2.1.3. Continuous Runtime Monitoring

To receive feedback from the operational phase, the running application must continuously be monitored (van Hoorn et al., 2009). In the context of Cloud computing, the term *observability* (Niedermaier et al., 2019) has emerged, denoting a service's ability to be monitored. Besides, there are various open-source monitoring and observability tools available, which the OpenAPM (Novatec Consulting GmbH, 2020[a]) initiative summarizes.

Continuous monitoring is particularly relevant for our work, as it delivers the basis for workload characterization (Calzarossa et al., 2016). A model of the workload can then enable representative load testing. As a standardized format of monitoring data, we refer to OPEN.xtrace by Okanović et al. (2016).

There are further standards from the practical domain, such as *OpenTracing* (2020), *OpenCensus* (2020), and *OpenTelemetry* (2020), parts of which we have integrated with OPEN.xtrace (H. Schulz, 2020c).

2.2. DevOps

As discussed in the previous section, CSE entails a neat collaboration between software development and operations. This collaboration, named *DevOps*, is often described fuzzily. In this work, we refer to the goal-oriented definition by Bass et al.

Definition 2.3 (DevOps — Bass et al., 2015)
DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality.

For that, DevOps closes the gap between development and operations. With the support of a high degree of automation, development teams deliver, support, and maintain their developed service. This reduces the communication with operations teams and, thus, is more efficient. Some authors use the term *DevOps team* (Davis, 2019; Wiedemann and T. Schulz, 2017), and we adopt it in this thesis to emphasize the team's responsibilities for tasks beyond development.

DevOps is strongly coupled with CI/CD. On the one hand, CI/CD is a mandatory DevOps practice (Bass et al., 2015). On the other hand, Humble

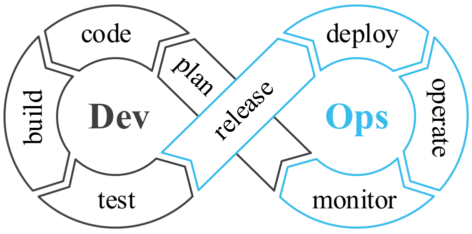


Figure 2.2.: DevOps life cycle (based on Davis, 2019).

and Molesky (2011) argue that successful continuous delivery requires applying DevOps. Therefore, it is not surprising that the often-shown DevOps life cycle, illustrated in Figure 2.2, is closely related to the Continuous * practices summarized in Figure 2.1. Development practices in the life cycle span the planning, coding, building, and testing, using pipelines for the latter two. The operations practices are release, deployment, operation, and monitoring, resulting in feedback for the planning. This infinitely repeating life cycle enables frequent releases. DORA and Google Cloud (2019) report that companies they assess as elite in DevOps perform multiple deployments per day.

DevOps puts specific emphasis on *high quality*, whereas performance is one of the most crucial attributes (Brunnert et al., 2015). To this end, DevOps poses both challenges and opportunities. Opportunities comprise automated deployment and easily accessible metrics, which ease and accelerate the testing process (Eismann et al., 2020). Also, the feedback from monitoring to development can be used to extract performance models, including workload models (Heinrich et al., 2017). Challenges are the short release cycles, unstable (Cloud) environments, and a high degree of automation, which performance engineering tasks need to fulfill (Bezemer et al., 2019; Eismann et al., 2018). Resulting from that, Bezemer et al. (2019) have found that many companies do not apply performance engineering systematically.

2.3. Microservices

The rigorously new principle of CSE also requires adapting the architecture of the developed software. The most frequently applied architectural style is microservices. Driven by companies such as Netflix and Amazon, the term *microservice* was first defined in 2012 (Lewis and Fowler, 2014) and adopted in many scientific publications starting from 2015 (Francesco et al., 2017; Pahl and Jamshidi, 2016). At very short, Newman provides the following definition.

Definition 2.4 (Microservices — Newman, 2015)
Microservices are small, autonomous services that work together.

Besides, Newman (2015) lists several inevitable characteristics and benefits of microservices, which we describe in the following.

Cohesive and Autonomous: Each microservice realizes a specific business concept, being an autonomous entity. As an example, otto.de, one of the largest European webshops, decouples concepts like the product, user, and order management (Hasselbring and Steinacker, 2017). The separation is also crucial from a technical perspective, e.g., it is mandatory to have separate codebases per microservice, and to deploy a microservice independently from others. Besides, it is a good practice not to share databases. For communication between microservices, each provides an application programming interface (API), often using Representational State Transfer (REST) or messaging protocols, such as Advanced Message Queuing Protocol (AMQP) or Kafka.

Aligned with the Organization: Conway (1968) argues that *"organizations which design systems (in the broad sense [...]) are constrained to produce designs which are copies of the communication structures of these organizations."* This statement is known as *Conway's law* and motivates a fundamental concept of microservices: the microservice architecture is also reflected in the organizational team structure. According to small services, small (DevOps) teams develop the services, typically with one-to-one or one-to-many mapping of teams to microservices (Hasselbring and Steinacker, 2017). Having individual CI/CD infrastructure and following DevOps practices, each team can realize the rapid CSE principles, such as frequent releases. Also, small autonomous teams support horizontal scaling with an increasing application, i.e., adding more teams instead of increasing the teams' size.

Easily Deployable: A key driver of CSE is the automated deployment of the developed application. This is particularly valid for microservices, which need to be deployable independently of each other. As a result, small parts of the application can be exchanged easily, e.g., for bug fixes, without redeploying the application as a whole.

Scalable: Microservices are also meant to be horizontally scalable, i.e., serving more requests by adding replicas. Unlike monolithic applications, microservices scale individually, which allows them to adjust to the specific workload situation automatically (Hasselbring, 2016; Herbst et al., 2013). Remarkably, adding replicas might not improve the application's performance, as we have found in joint work with Avritzer et al. (2018, 2020a).

Technologically Heterogeneous: Due to the clear separation between microservices, each can use a different technology stack. For instance, they might use different programming languages and database implementations. This allows the DevOps teams to use the technology best suited for the use case they develop. For communication, the microservices must provide a technology-independent API, such as REST or Kafka.

Resilient: To avoid cascading failures of the whole application, microservices should be *resilient*, i.e., they should be tolerant regarding the failures of other microservices or the infrastructure they operate on. Nygard (2018) presents a set of stability patterns, which help to increase a microservice's resilience. To this end, an entirely new way of testing has emerged, called *chaos engineering*, which randomly causes failures in the production environment *"to build confidence in its capability to withstand turbulent conditions"* (Basiri et al., 2016).

Observable: A fundamental practice of DevOps and CSE is the monitoring of the production application. Therefore, observability (see Section 2.1.3) is a crucial attribute of microservices. Due to the distributed and potentially heterogeneous application landscape, each microservice particularly must support the collection of end-to-end execution traces (Okanović et al., 2016). These traces can restore a global view of the whole application's runtime behavior.

```
openapi: 3.0.1
info:
  title: User
  description: Provide Customer login, register,
     retrieval, and card and address retrieval
paths:
  /login:
    get:
      description: Return logged in user
      operationId: Get Login
      responses:
        200:
          content:
            application/json;charset=UTF-8:
              schema:
                $ref: '#/components/schemas/
                   Getcustomersresponse'
```

Listing 2.1: Excerpt from an exemplary OpenAPI specification[1].

2.4. API Specifications

Microservices communicate with each other via APIs they expose. In doing so, it is crucial to document the API, including how to call the microservice and which response to expect. API specifications are relevant for our work because they also document the endpoints to be covered in a load test.

Providing a specific format, we point to the OpenAPI Initiative (2020), whose previous versions were called Swagger. The YAML-based (*YAML* 2020) format allows documenting REST APIs, which we illustrate in Listing 2.1. The example shows an excerpt from the API of the Sock Shop (Weaveworks, Inc., 2020) application's *users* service, which we also utilize in our evaluation (see Chapter 13). Besides informal descriptions of the service, it contains the available paths — in this case, the path to the login endpoint. When the users service receives a GET request at that path, it returns the logged in

[1]https://raw.githubusercontent.com/microservices-demo/user/master/apispec/user.json

user — as documented — in JavaScript Object Notation (JSON) with a 200 HTTP response code. Besides, the specification refers to the JSON schema of the response, which we omit for space reasons.

2.5. Summary

In our research, we focus on load testing in continuous software engineering (CSE). Hence, principles like continuous integration and delivery (CI/CD), DevOps, and microservices pose specific requirements for our approach. These are a high degree of automation, resource- and time-efficiency, and service-tailored testing, accounting for automated CI/CD pipelines, frequent releases, and the teams' focus on their developed services. At the same time, CSE brings opportunities, such as the availability of continuously collected monitoring data, API specification standards, and easily deployable services. These opposing factors make the topic of this work particularly relevant for research.

Chapter 3

Workload Characterization & Forecasting

For generating a load test that is representative of the production workload, the workload needs to be characterized and potentially forecasted to the future. Both techniques have been extensively studied. This chapter describes their foundations. First, we introduce the fundamental terminology in Section 3.1. Sections 3.2 and 3.3 describe workload characterization and forecasting. Finally, Section 3.4 provides a summary.

3.1. Terminology

The central term of this chapter is the *workload* of a software system or application, also referred to as *operational profile* (Musa, 1993) or *usage profile* (Koziolek et al., 2007). Calzarossa et al. define workload as follows.

Definition 3.1 (Workload — Calzarossa et al., 2016)
The term workload refers to all inputs received by a given technological infrastructure [software system].

The type of the inputs depends on the software system. For instance, in the context of a web-based application, the workload consists of all HTTP requests submitted by a user or client and received by the application. In other fields, the inputs can be transactions (e.g., database systems), messages (e.g., e-mail systems), or jobs (e.g., cloud systems) (Calzarossa et al., 2016; Menascé and Almeida, 2002). This thesis focuses on web-based applications. We use the term *request* as a synonym for the smallest unit of work that is part of the input and to be serviced by the application. Similarly, we denote the person or entity that submits a request as *user*, subsuming persons, devices, client software, or other entities submitting requests.

An important categorization of workloads originating from queuing theory are *closed* and *open* workloads (Balsamo and Marin, 2007; Schroeder et al., 2007). In a closed workload, a fixed set of users, i.e., the *population*, interacts with the system concurrently. Each user runs continuously and submits requests sequentially. In between of two requests, the users wait a time span denoted as *think time*. Hence, each request depends on the response of the former request of the respective user. In contrast to this, an open workload comprises requests submitted by users that arrive independently at a given *arrival rate* and depart after submitting a request. The single requests are independent of each other.

Workload characterization is the process of decomposing the global workload of the whole software system into its *basic components* — e.g., the user sessions of a session-based e-commerce application — and specifying the two main characteristics *intensity* and *service demands* per basic component (Menascé et al., 1999). For closed workloads, the intensity is defined by the number of users of the population and the think time. For open workloads, the *arrival rate* or its reciprocal — the *inter-arrival time* — define the intensity. The service demands refer to the properties of the individual basic components that have an effect on the system's performance, e.g., the size of files requested (Menascé et al., 1999). The union of all basic components with its characteristics and the relative frequencies is typically referred to as *workload mix* (Vögele et al., 2018).

The outcome of the workload characterization process is a *workload model* that aims at accurately describing the real workload while being compact and reproducible (Menascé and Almeida, 2002). For session-based systems, workload models typically comprise a *user behavior model* describing the basic components and subsuming several similar user sessions (Vögele et al., 2018). One of the most important features of a workload model is *representativeness*, denoting the similarity to the actual workload in terms of the utilization of the system (Menascé and Almeida, 2002).

It can be relevant to predict the workload the studied application is expected to observe in the future, e.g., for proactive resource provisioning (Herbst et al., 2013) or load testing of future scenarios (see Chapter 8). The process of this prediction is called *workload forecasting*. Menascé and Almeida (2002) argue it should be composed of quantitative and qualitative methods. *Quantitative forecasting* is based on historical data. *Qualitative forecasting*, in contrast, relies on experts' judgment and intuition. The latter method is mainly relevant for workload scenarios that are influenced by external factors, e.g., planned changes to a platform or technological changes.

3.2. Workload Characterization

The careful characterization of the production workload is fundamental for reasonable performance evaluation (Ferrari, 1972; G. Jin et al., 2012; Menascé and Almeida, 2002). In our work, we require it to derive load tests that are representative of the production workload. In this section, we detail the workload characterization process, the notion of a workload model, how to evaluate representativeness, and the WESSBAS approach by Vögele et al. (2018) we base on in this research.

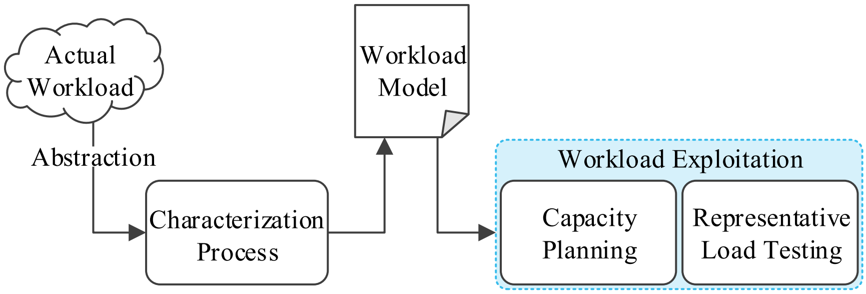


Figure 3.1.: Workload characterization process (based on Menascé and Almeida, 2002).

3.2.1. Characterization Process

Figure 3.1 illustrates the general use case of workload characterization. The characterization process operates on an abstraction of the actual production workload based on measurements. This process outputs a workload model, which can then be exploited in different studies. Menascé and Almeida (2002) use it for capacity planning; we transform it into a representative load test. In doing so, the type of workload exploitation influences the characteristics and level of detail of the workload model. For instance, for capacity planning, the impact on the hardware resources needs to be captured. For load testing, this level is less relevant.

Many approaches realize the characterization process by different analysis techniques, which Calzarossa et al. (2016) summarize in a methodology. Statistical analysis is often applied as a preliminary, exploratory step to understand the workload. Multivariate analysis, such as clustering and principal component analysis (PCA), is used to describe the workload's overall properties. Numerical fitting is useful for modeling temporal patterns, and stochastic processes are well suited to describe time-varying properties. Finally, workload models can be derived using graph analysis, e.g., applied to the overall properties identified using multivariate analysis.

In our work, we focus on session-based workloads. In this type of workload, a basic component is a user session, consisting of one or multiple user-submitted requests. Menascé et al. (1999) introduce a combination of analysis techniques we adopt in this work. First, they transform each session into an equally sized matrix. Then, they apply k-means clustering. To finally obtain a workload model, they apply graph analysis to the cluster centroids. Vögele et al. (2018) extend this procedure in their WESSBAS approach, which we describe in Section 3.2.4.

3.2.2. Workload Models

The properties of a workload model — i.e., the result of the characterization process — are diverse and depend on the type of workload and study. A general definition is the following.

Definition 3.2 (Workload model — Menascé and Almeida, 2002)
A workload model is a representation that mimics the real workload under study. It can be a set of programs written and implemented with the goal of artificially testing a system in a controlled environment. A workload model can also be a set of input data for an analytical model of a system. [. . .] Models should be compact.

To render a more precise definition, we focus on session-based workloads. In this case, the model needs to capture the behavior of the users. Differently behaving users might be separated into user groups, e.g., heavy buyers or occasional buyers of an e-commerce system. For each group, the model needs to specify the requests s single user submits, e.g., to login to the system, browse products, and add them to the shopping cart, and the timing, i.e., think times between the requests. Workload modeling formalisms that fulfill these criteria include Markov chains (Menascé et al., 1999; Vögele et al., 2014), stochastic form-oriented models (Cai et al., 2007; Lutteroth and Weber, 2008), probabilistic timed automata (Abbors et al., 2013a,b, 2014), extended finite state machines (EFSMs) (Shams et al., 2006), and request sequences (Krishnamurthy et al., 2006). In our work, we focus on Markov chains, which Z. Li and Tian (2003) have assessed to be suitable for modeling session-based workloads.

Another type of model that is relevant for session-based workloads is an intensity model. For open workloads, this model describes the arrival rate of new sessions, which might change over time. For closed workloads, it captures the number of concurrently active sessions, assuming each session restarts when it reaches an end state. The simplest intensity models are a single number, such as 500 sessions being active constantly, and a number per time unit to describe a varying intensity. As a more complex representation, which primarily allows experts to model a varying intensity manually, the Descartes Load Intensity Model (DLIM) (von Kistowski et al., 2014b, 2017), connected with the LIMBO tool (von Kistowski et al., 2014a), composes trend, seasonal, burst, and noise functions.

3.2.3. Evaluating Representativeness

A workload model that was extracted from the production workload should be validated concerning its representativeness. The general procedure is to select a reference workload, e.g., the production workload from which the model was extracted, and derive metrics for comparison (Menascé and Almeida, 2002). By emulating the workload defined by the model, e.g., by executing a load test (see Chapter 4), and collecting measurements during the emulation, the metrics can be determined for the model the same way as for the reference workload.

Several metrics have been proposed to compare session-based workloads, comprising intra-session, inter-session, request-based, and performance metrics (Goseva-Popstojanova et al., 2006; Vögele et al., 2018). Intra-session metrics include the following:

- Session length: Number of requests submitted within one session.
- Session duration: Time elapsed between the first and the response to the last request within one session. Please note that Goseva-Popstojanova et al. (2006) denote this metric as session length, while we stick to the terminology by Vögele et al. (2018).
- Bytes per session: Number of bytes submitted within one session.

Inter-session metrics address the overall characteristics of all sessions:

- Session arrival rate: Number of sessions started within a defined time unit, e.g., minute, hour, or day. This metric belongs to an open workload.
- Concurrently active sessions: Corresponding metric for closed workloads.
- Sessions per user: Number of sessions started by a single user. The start of a second session of the same user is identified by a new session ID the application has defined or a predefined time of inactivity, e.g., 30 minutes.
- Unique sessions: Number of sessions with requests to the same endpoints in the same order.

Request-based metrics disregard the notion of sessions and focus on the number of requests submitted per time unit, i.e., the request rate. The request rate can be further subdivided according to the endpoint called and the response status, e.g., successful, client-side erroneous, or server-side erroneous.

Naturally, the workload influences the behavior of the application (Ferrari, 1972). Thus, we can also argue about representativeness based on performance metrics. Menascé and Almeida (2002) suggest using the response times; further metrics can be the CPU utilization and memory consumption. Notably, a test causing the same performance metric values as the reference workload is not necessarily highly representative. However, differences are a reliable indicator that the model is not representative of the reference workload.

3.2.4. The WESSBAS Approach

Vögele et al. (2018) propose the WESSBAS approach, which bases upon the methodology by Menascé et al. (1999). In this work, we further extend it for generating tailored load tests. In this section, we present WESSBAS' characterization process and the WESSBAS-DSL as the output format.

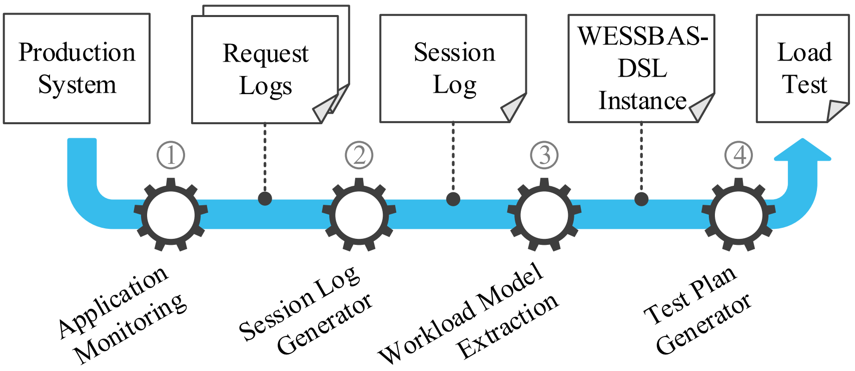


Figure 3.2.: WESSBAS workload characterization process (based on Vögele et al., 2018).

3.2.4.1. Characterization Process

Figure 3.2 illustrates the process of extracting a workload model from the production workload and transforming it into a load test. This process consists of four steps, which we explain in the following. While WESSBAS is generally applicable, we focus on its most frequent use case, namely HTTP workloads.

The first step is the collection of request logs via application monitoring ①. These logs contain one entry per request the production system received. For each request, the timestamp, response time, endpoint called, and session ID are required. Besides, the characterization process can extract parameter values sent with the request, such as login credentials. The session ID is assigned by the production system and returned to the user, who reuses it in subsequent requests. It can also be approximated using a unique identifier of the user, e.g., the client IP address.

```
SESSION_ID;"ENDPOINT_ID":START:END:REQEST_DETAILS;
    "ENDPOINT_ID_2":...
SESSION_ID_2;...
```

Listing 3.1: Illustration of the Session Log schema.

Next, the request logs are transformed into a standardized session log by grouping them according to the session ID ②. As illustrated in Listing 3.1, one entry of the log corresponds to one session. The first item of each entry is the session ID, followed by requests in chronological order. For each request, the entry contains an endpoint ID, start and end timestamps, and details about the requests, such as the HTTP path, port number, domain name, protocol, request method, and request parameters.

The actual workload model extraction happens in step ③ and is subdivided into three parts. First, the *Workload Intensity Extractor* determines the number of concurrently active sessions. For learning varying intensities, WESSBAS leverages the LIMBO (von Kistowski et al., 2014a) tooling.

Second, the *Behavior Mix Extractor* extracts Markov-chain-based behavior models. For that, it transforms each session log entry into a vector, whose entries represent the absolute transition frequencies between two endpoints. As an example, if one session contains three times subsequent requests to the *browse* and *add to cart* endpoints, the corresponding vector entry is 3. Then, the vectors are clustered using X-means. The behavior models are calculated from the resulting cluster centroids by normalizing the absolute transition frequencies to relative ones, and computing think time distributions per transition. Besides, the Behavior Mix Extractor calculates the behavior mix, i.e., the relative number of sessions aggregated in each behavior model.

In the third part of the extraction step, the *WESSBAS-DSL Model Generator* transforms the behavior models and workload intensity into an instance of the WESSBAS-DSL. In addition, it determines guards and actions (GaAs), which further control the emulated request behavior. For instance, the transitions of a behavior model might allow removing several items from the shopping cart with a non-zero percentage, regardless of the number of items in the cart. GaAs can prevent such invalid behavior by ensuring that only as many items as previously added can be removed. *Please note, in the scope of this dissertation, we exclude the extraction of guards and actions. Integrating them with our approach is left for future work.*

For exploiting the extracted workload model, WESSBAS provides transformations into a load test ④ and a performance model. For our work, the former transformation is more relevant. While WESSBAS comes with support for JMeter (Apache Software Foundation, 2020[a]), we further integrate the BenchFlow approach by (Ferme and Pautasso, 2018) (see Section 6.5.3).

3.2.4.2. The WESSBAS-DSL

WESSBAS provides a domain-specific language (DSL) for the description of workload models. As Figure 3.3 shows, WESSBAS-DSL instances consist of four parts. The first part is the application model, which is hierarchical, i.e., consists of two layers. The session layer is an EFSM describing possible

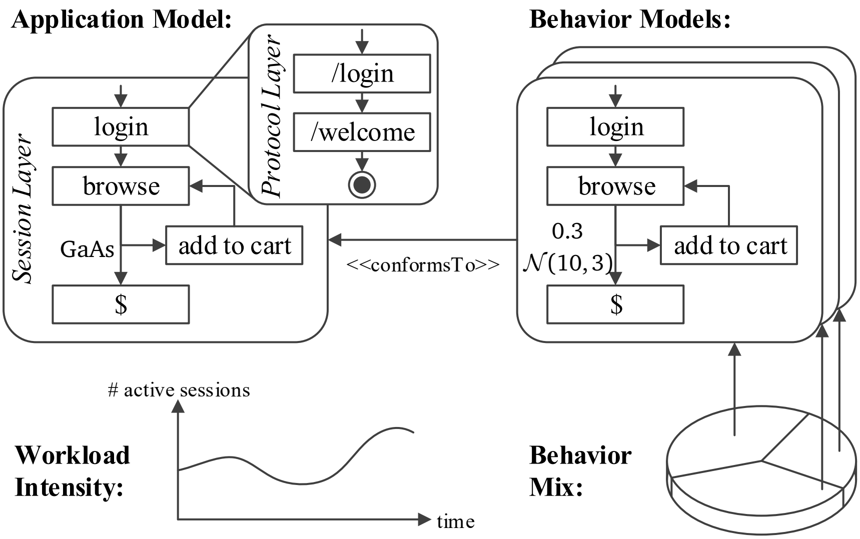


Figure 3.3.: Illustration of the WESSBAS-DSL (based on Vögele et al., 2018).

courses of actions of a user. It consists of states, e.g., the login to the application, and transitions between the states. The transitions are labeled with GaAs. Each of these states holds another EFSM at the protocol layer. Here, the states represent concrete requests. The two layers allow modeling several requests per user action, e.g., when a user submits a login in the web user interface (UI), the UI might send two requests to */login* and */welcome*. The characterization process, however, always outputs protocol-layer EFSMs with exactly one state. An exception is the final state $, which does not include any requests.

The second part of the DSL is the behavior models. They contain states that relate to the states of the application model. Instead of GaAs, the states are connected with transition probabilities and think time distributions. That is, the next state is always determined based on the previous state and the transition probabilities. Furthermore, when switching to the next state, the think time defines the time to wait, e.g., according to a normal distribution.

The behavior mix specifies the relative frequencies of the behavior models. As an example, a WESSBAS-DSL instance might contain three behavior models with a mix of 40 %, 35 %, and 25 %. The workload intensity holds the number of concurrently active sessions over time.

Summing up, we can reconstruct the production workload from a WESSBAS-DSL instance by executing as many sessions as defined by the workload intensity. Each session emulates one behavior model, whereas the behavior mix defines the probability of selecting a specific one. The application model defines which requests to submit for each state, and the GaAs ensure that no invalid request sequences are produced.

3.3. Workload Forecasting

While the workload characterization approaches described in the previous section can extract representative models of the production workload, they are restricted to the past. There can be cases, however, where we are interested in the expected future workload. For that, we can apply workload forecasting, which Menascé and Almeida define as follows.

Definition 3.3 (Workload forecasting — Menascé and Almeida, 2002)
Workload forecasting is the process of predicting how system workloads will vary in the future.

Forecasting can be done in two ways: qualitatively and quantitatively. We present both and further introduce the tools we leverage in our work.

3.3.1. Qualitative Forecasting

Qualitative forecasting denotes the prediction of workloads based on personal estimation by experts (Menascé and Almeida, 2002). While intuitive assessments tend to be of low accuracy, they can be the only option to obtain a forecast, e.g., because business decisions cause future scenarios that have never been observed before. An example is from our evaluation (see Section 15.1), where the operators of an Internet of things (IoT) platform decided to add more devices to the platform at a defined date. With a manual estimation of the workload increase, we are able to predict the future workload.

Methods that support the determination of a qualitative forecast can be group votings or the search for historical analogies. The Delphi method (Martinich, 2008) is a structured method involving multiple stakeholders. It presumes that groups of experts provide more accurate forecasts than individuals. In multiple rounds, the experts answer questionnaires. After each round, they receive an anonymized summary of the forecasts, to potentially adapt their own ones, until they reach a predefined stop criterion. In rare cases, the forecast process is straightforward, e.g., in the example from our evaluation, which only requires multiplying the current workload with the fraction of newly added devices.

3.3.2. Quantitative Forecasting

In contrast to qualitative forecasting, its quantitative opponent only relies on historical data to predict future values (Menascé and Almeida, 2002). Hence, the forecast accuracy is higher. However, the accuracy also depends on the horizon, i.e., the period between the latest known and the latest predicted value. The longer the horizon is, the lower is the accuracy. As a result, different methods should be applied to different horizons. Once a method has been selected, it should be validated, e.g., by using a part of the recorded workload to predict the remainder.

Established forecasting methods comprise linear or non-linear regression, moving average, and exponential smoothing (Menascé and Almeida, 2002). Besides, the decomposition into trend, season, and noise is often applied (Herbst et al., 2013). The trend is a monotonic function that can be easily predicted using regression methods. The season is a periodical component and reflects, e.g., daily and weekly variations. Mahalakshmi et al. (2016) survey further elaborate forecasting methods, such as machine learning techniques.

Due to the large number of available methods and the level of expertise required to select an appropriate one, we rely on tools that support us in this task. In the next section, we introduce them.

3.3.3. Forecasting Tools

In the following, we introduce the tools we use for workload forecasting in our approach. These are Telescope by Bauer et al. (2020) and Prophet by Taylor and Letham (2018). They have in common that they aim to predict future values of a given time series. That is, they can be used to forecast the workload intensity.

3.3.3.1. Telescope

Telescope (Bauer et al., 2020) is an automated time series forecasting approach developed at the University of Würzburg. It takes as input a time series and outputs the future values until a defined horizon. The forecast is calculated in three steps. First, Telescope preprocesses the time series, with methods including Box-Cox transformation (Box and Cox, 1964), Fourier terms extraction, and seasonal and trend decomposition using Loess (STL) (R. B. Cleveland et al., 1990). In the second step, Telescope uses the outputs of the preprocessing to train a model using eXtreme Gradient Boosting (XGBoost) (T. Chen and Guestrin, 2016). Finally, it does the forecasting using, among other methods, autoregressive integrated moving average (ARIMA) (Box et al., 2015), pattern forecast, XGBoost, and inverse Box-Cox transformation. The authors of the tool have also evaluated Telescope to provide forecasts with a lower error than competitive state-of-the-art approaches while keeping the time-to-result comparably low (Bauer et al., 2020).

The main benefit for us is the high forecast performance without parameter tuning. As opposed to the *"No-Free-Lunch Theorem"* (Wolpert and Macready, 1997), Telescope performs well with various time series. Hence, we can predict workloads automatedly without distorting evaluation results by poor manual configuration.

Telescope can be used as an R (R Core Team, 2019) package, whereas we utilize a modified branch for multivariate forecasting (Chair of Software Engineering, University of Würzburg, 2020). In addition to the time series, we can input *covariates*. These are time series with past and future values representing contextual information, e.g., the presence of special sales or weather conditions. The Telescope branch integrates the past values into the model training and uses the future values to influence the forecast. By providing adequate covariates, we can further improve forecast accuracy.

3.3.3.2. Prophet

Another open-source forecasting tool developed at Facebook is Prophet (Taylor and Letham, 2018). Unlike Telescope, it explicitly integrates analysts into the forecasting process. Being provided with an evaluation of the forecast accuracy, they should tune the forecasting parameters. Like Telescope, Prophet is available open-source (Facebook, 2016) as an R package and, besides, as a Python (Van Rossum and Drake Jr, 1995) library. We use the R package for integration into a common tooling with Telescope and good comparability in our work.

Essentially, Prophet treats the forecasting task as a curve-fitting problem. Being provided with a time series, it aims at fitting the following function $y(t)$ of time t into the data.

$$y(t) = g(t) + s(t) + h(t) + \varepsilon_t$$

The individual terms of the equation relate to the decomposed time series components we discussed in Section 3.3.2: $g(t)$ is the trend function, $s(t)$ represents the seasonality, and ε_t constitutes the error term, i.e., the noise. Besides, Prophet integrates the effect of irregularly occurring holidays in the term $h(t)$. As a generalization of holidays, they allow specifying *regressors*, which serve the same functionality as Telescope's covariates. Hence, we can fairly compare the forecasting performance of both tools using similar concepts regarding the input and output.

3.4. Summary

Workload characterization and forecasting are fundamental techniques for representative load testing. Characterization approaches, such as WESSBAS (Vögele et al., 2018), extract workload models that accurately represent recorded production workload. Using forecasting methods, we can further predict expected future workloads. Our research bases upon the WESSBAS approach and extends its characterization technique to generate service- and context-tailored models. Leveraging Telescope (Bauer et al., 2020) and Prophet (Taylor and Letham, 2018), we also cover future workload scenarios.

CHAPTER 4

LOAD TESTING

In this chapter, we introduce the foundations of load testing, which is the primary discipline of interest in this research. Particularly, we focus on load tests that are representative of the production workload. Section 4.1 provides an overview of load testing, followed by its general process in Section 4.2. In Section 4.3, we discuss input data used by load tests, which significantly contribute to the success or failure of a load testing strategy. Furthermore, we introduce tools we leverage in Section 4.4. Section 4.5 summarizes the chapter.

4.1. Load Testing Overview

In this work, we base upon the load testing definition by Jiang and Hassan, as follows.

Definition 4.1 (Load testing — Jiang and Hassan, 2015)
Load testing is the process of assessing system behavior under load in order to detect problems due to one or both of the following reasons: (1) functional-related problems (i.e., functional bugs that appear only under load), and (2) nonfunctional problems (i.e., violations in non-functional quality-related requirements under load).

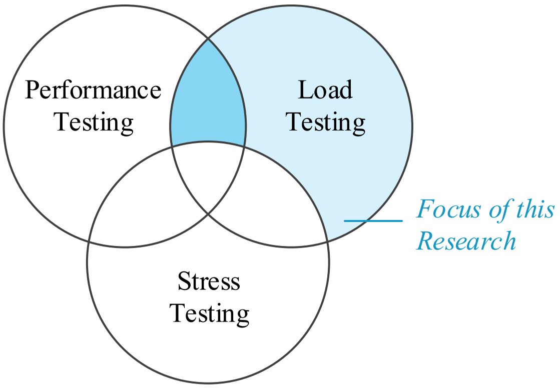


Figure 4.1.: Relationship of load, performance, and stress testing (based on Jiang and Hassan, 2015).

To assess the system behavior under load, this load needs to be generated artificially in a test. We prefer using the term *workload* instead of load to emphasize that it has similar characteristics as the inputs the system receives in production (see Chapter 3). As an example, a load test might aim at replaying a recorded workload. Besides, load tests use workload models to describe the load issued to the system under test (SUT).

As illustrated in Figure 4.1 and described by Jiang and Hassan (2015), load testing overlaps with other testing disciplines, such as performance testing and stress testing. Performance testing denotes measuring the performance of software entities varying from the unit level to the system level. Primary measures are response time, throughput, and resource utilization. Load testing that targets the collection and evaluation of performance metrics is an overlap with performance testing. Stress testing aims at testing a system's ability to behave under stressful conditions, e.g., loads that are higher than expected or limited hardware resources. Hence, tests replaying with extraordinarily high loads are both load and stress tests.

Our work's focus is load testing disjoint from stress testing, with a particular emphasis on the intersection with performance testing. That is, the load tests we generate using our approach should always be suited to evaluate the performance of the SUT. Still, they can be used for detecting functional or non-function non-performance problems. Furthermore, we test the expected scenarios under the regular availability of resources, which is different from stress testing.

A fundamental differentiation of load tests is the design of the workload they submit to the SUT. Jiang and Hassan (2015) identify two principal schools of thought. *Fault-inducing loads* aim to identify and, thus, cause as many problems as possible. These loads are designed by locating potential weaknesses of the SUT, e.g., using source code or model-based analysis. However, as many authors argue (Ferrari, 1972; G. Jin et al., 2012; Menascé and Almeida, 2002), reasonable performance analysis requires workloads that are representative of the production environment. Therefore, we contribute to testing with production-representative, i.e., *realistic loads*. As a further benefit, this type of load design allows us to evaluate whether a developed system can perform under a future workload scenario, e.g., a sharp increase of users accessing a webshop during a special sale. Notably, fault-inducing and realistic load testing do not need to be disjoint. Vögele (2018) introduces a multi-objective optimization approach based on WESSBAS (see Section 3.2.4) to derive workloads that can both be representative and likely induce problems.

4.2. Load Testing Process

Load testing is a process that consists of three steps: load test design, load test execution, and load test analysis (Jiang and Hassan, 2015). Below, we describe the steps, focusing on representative load testing, which is most relevant for our work. For a comprehensive review of load testing activities, we refer to Jiang and Hassan (2015).

- *Designing a load test*: The first step is to design a load test, primarily its workload, aligned with the test objectives. Representative load tests can be derived using workload characterization and forecasting approaches (see Chapter 3). The period of workload to characterize, which should be selected carefully (Ferrari, 1972), depends on the test objective. For instance, a typical workday workload can be used to identify the amount of resources the SUT requires under normal conditions, while expected sharp increases of the near-future workload are better suited to test whether the system can withstand an upcoming special sale.

- *Executing a load test*: While other execution methods exist, such as human testers or special platforms emulating parts of the SUT, we base upon driver-based execution. That is, a load driver issues the workload specified by the load test to the SUT. For session-based applications, the driver emulates the sessions of a potentially large number of users. Both the driver and the SUT need to be deployed in a test environment, preferably on different machines, to prevent mutual interference. The best results can be achieved with a production-like setup. Some approaches support users with automated deployment (BlazeMeter, 2015; Ferme and Pautasso, 2018).

  During the load execution, measurement data should be collected. Open-source and commercial load testing tools, such as JMeter (Apache Software Foundation, 2020[a]), Gatling (Gatling Corp, 2020), Locust (Heyman et al., 2020), LoadRunner (Micro Focus, 2020[a]), and SilkPerformer (Micro Focus, 2020[b]), generate client-side log files for analysis. Further insights can be gained using SUT-side monitoring techniques (Brunnert et al., 2015).

  Finally, the termination of the test is relevant for the test execution. An often-applied and straightforward method is to stop the test after a predefined time. Alternatively, it can be run continuously until stopped by a human, or until predefined termination criteria are met, e.g., the performance metrics of interest are statistically stable (Alghamdi et al., 2020, 2016).

- *Analyzing a Load Test*: This final step analyzes the measurement data collected during the execution regarding functional and non-functional problems, with several approaches automating this step (Jiang et al., 2008, 2009; Malik et al., 2010a,b, 2013, 2010c). It can either compare them with predefined thresholds, search for known issues, or identify behavioral anomalies. The choice of analysis also depends on the test objectives. For instance, to find the optimal hardware configuration that allows the SUT to serve the workload while satisfying a service-level agreement (SLA), the threshold-based approach is valuable. Searching for known issues is more appropriate when checking whether a bug fix resolves a previously identified memory leak.

4.3. Load Test Input Data

Load tests for session-based applications replay specific user sessions. Figure 4.2 illustrates one such session, consisting of requests for login, product browsing, and adding a product to the shopping cart. In between two requests, there are think times. The described type, order, and timing of requests are the most relevant characteristics of the basic components of a session-based workload (Menascé et al., 1999). However, the meaningful execution of the test also requires appropriate input data sent with each request. As an example, the login request needs a user name and password as parameters. Presuming the request should result in a successful login, the data input to these parameters need to be valid credentials; otherwise, the SUT will behave differently than expected.

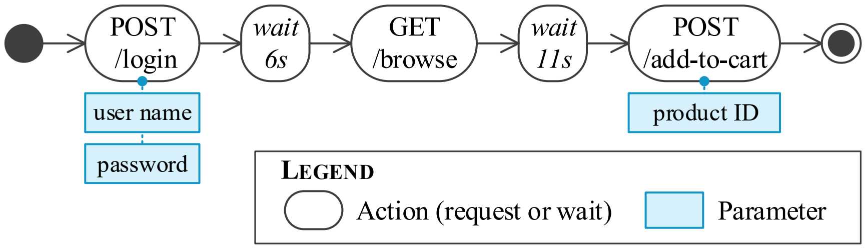


Figure 4.2.: Activity diagram of an exemplary user session interacting with a webshop via HTTP.

Session-based workload characterization approaches extract workload models from recorded sessions (see Section 3.2). While doing so, they also can capture the input data. However, the load test is executed in a test environment with different databases than in the production environment, for privacy and security reasons (Jiang and Hassan, 2015). Therefore, most of the captured input data, including our example of login credentials, are invalid. Instead, they need to be replaced with comparable data from the test database. Easing this process, approaches exist that generate test databases, e.g., by Bainbridge et al. (2009), Barros et al. (2007), Farahbod and Dadashi (2017), and Grechanik et al. (2010).

Remarkably, the notion of input data, as opposed to the basic workload components, strongly depends on the type of workload. For instance, an application might only provide a single Hypertext Transfer Protocol (HTTP) endpoint, characterized by a path and request method, e.g., for uploading messages of various sizes. Here, the messages are the basic workload components, while they are also input data according to our definition. Therefore, we introduce the term *session-dominated workload*, denoting a session-based workload mainly characterized by the requested endpoints, request order, and timing. Session-dominated workloads only require the input data to be "correct", e.g., be valid login credentials.

4.4. Load Testing Tools

In our research, we use a workload characterization approach to generate a representative workload model and transform it into a load test. For the second step of the load testing process, i.e., the test execution, we utilize two different tools. The first tool is JMeter (Apache Software Foundation, 2020[a]), which is open-source and widely adopted. The BenchFlow approach by Ferme and Pautasso (2018) provides further automation and deployment support.

4.4.1. Apace JMeter

JMeter (Apache Software Foundation, 2020[a]) is an open-source load testing tool implemented in Java. It was initially designed to test web applications and extended to other domains, such as Java Database Connectivity (JDBC) and Java Message Service (JMS). Test plans, which describe the load test to be executed, are defined as Extensible Markup Language (XML) files, with editing support by a graphical user interface (GUI). Test plans model closed workloads consisting of threads that follow a defined tree-based structure of samplers. Each time a thread reaches the end of the tree, it restarts again. A sampler executes a request, e.g., to an HTTP endpoint, and collects functional and performance data about the request. Further elements of the tree can manage the request timing or define input data. For executing a test, JMeter provides a command-line interface (CLI). Due to its extensibility via plugin mechanisms and large community, JMeter is widely used in practice.

Our approach does not rely on the GUI but uses a transformation of WESSBAS workload models (see Section 3.2.4) into JMeter test plans. This transformation utilizes JMeter's Java API (Apache Software Foundation, 2020[b]) and the Markov4JMeter plugin (van Hoorn et al., 2008), which enables executing Markov-chain-based workload models. The plugin defines the test plan tree as one or multiple concurrent Markov chains, with one sampler per state, similar to the WESSBAS DSL.

Regarding the load test execution, JMeter only supports submitting the workload to the SUT but not the SUT's deployment. To increase the level of automation, we use BenchFlow, which we introduce in the next section.

4.4.2. BenchFlow

Adding more automation and deployment support, Ferme and Pautasso (2017, 2018) introduce BenchFlow, a model-driven framework to automate the *"end-to-end process of executing performance tests."* They provide a declarative domain-specific language (DSL) based on the YAML format to describe performance tests. Each test has a stated goal, e.g., a simple load test or an exploratory testing strategy to obtain the optimal SUT configuration for a given workload. Besides, the DSL allows specifying the workload, data collected during the test execution, quality gates, and the deployment of the SUT. For the last aspect, the DSL relies on Docker (Docker Inc., 2020) deployment descriptors. Tests are executed using the *Faban* (2020) load driver or, introduced by Palenga (2018), JMeter.

Given a DSL instance, BenchFlow automatically handles the execution lifecycle. It determines the experiments to execute, e.g., the same workload with different SUT configurations, deploys the SUT, executes the workload, and collects monitoring data. Also, it analyzes the results with respect to the quality gates. Thus, it integrates well with continuous integration and delivery (CI/CD) practices and provides better support for the whole test execution phase than JMeter. As we generate workload models in our approach, the combination with BenchFlow provides a high degree of automation and abstraction (see Chapter 9).

4.5. Summary

Load testing is a fundamental technique to assess a system's ability to handle a given workload and identify load-related problems before delivery. It overlaps with performance and stress testing, whereas we exclude stress testing in this work. The main steps of load testing are test design, test execution, and test analysis. Leveraging workload characterization approaches, we contribute to the design step by extracting tailored load tests from the production workload. For the test execution, we rely on existing tools, such as JMeter (Apache Software Foundation, 2020[a]) and BenchFlow (Ferme and Pautasso, 2018). Test analysis, apart from the means provided by JMeter and BenchFlow, is out of the scope of this dissertation.

In the next part, we introduce our proposed approach to the automatic generation of tailored load tests. We start by describing the research design.

Part II

Approach

Chapter 5

Research Design

In this chapter, we provide an overview of our research design and the approach we have developed. We define our goal and corresponding research questions in Section 5.1. We make a few assumptions, which we explain in Section 5.2. In Section 5.3, we structure our work into work packages. Section 5.4 provides an overview of our approach. Finally, in Section 5.5, we describe the collaborations that were carried out in this research.

5.1. Goal and Research Questions

The following defines the goal of our research and introduces the research questions. There are four main questions, which we further split into sub-questions.

5.1.1. Overview

The main goal of our research is the following:

> ***Automate the generation of representative load tests that are tailored to the relevant services and workload-influencing contexts of a session-based, continuously developed application.***

General implications of continuous software engineering (CSE) are short release cycles, automation, and concurrency, e.g., by multiple teams developing one microservice each (Bass et al., 2015). In this context, Bezemer et al. (2019) have found that performance engineering tasks, which include representative load testing, are rarely applied regularly. The authors partially attribute this finding to the high level of expertise these tasks require. As a further reason, T.-H. Chen et al. (2017) report that load tests require a long execution time and much hardware, which hinders concurrent execution.

Generating load tests tailored to the specific situation addresses several of these aspects. By tailoring a load test to an individual (micro) service, we can reduce the required hardware, which eases concurrent execution. As the workload frequently changes (Herbst et al., 2013; Jiang and Hassan, 2015), influenced by current events and circumstances (Chandola et al., 2009), the number of load tests to execute can be reduced by focussing on the current workload context. However, the on-demand generation of tailored load tests poses the new challenge of automation, which CSE requires (Bezemer et al., 2019). Existing approaches typically require applying manual parameterizations to generated representative load tests (Vögele et al., 2018), which need to be evolved when generating new load tests.

Concluding, we address the challenges of automated load test parameterization, tailoring to services, tailoring to workload contexts, and also easing load testing for non-experts, leading to the following main research questions.

RQ1: How can load test parameterizations be evolved without manual intervention at test generation or execution time?

RQ2: How can representative load tests be tailored to specific services of a session-based application?

RQ3: How can representative load tests automatically be tailored to the contexts of a session-based workload?

RQ4: How can we leverage automated tailored load test generation and automated load test execution for enabling load testing for non-experts?

5.1.2. Details

We further detail each research question into sub-questions regarding specific aspects to be developed or evaluated.

5.1.2.1. RQ1 — Evolving Load Test Parameterizations

There are two main types of changes that may affect a load test parameterization and, thus, need to be considered for parameterization evolution: workload changes and changes to the tested application's API. In both cases, previously generated load tests become outdated, requiring the generation of new tests. We aim to evolve a user's parameterizations over the changes if they have parameterized the load tests manually. In the case of workload changes, we presume we can fully automate the evolution. The degree of automation and, thus, reduction of manual maintenance effort in the case of API changes depends on the change type. Hence, we formulate the following sub-questions of RQ1 to be investigated.

RQ1.1: How can load test parameterizations be automatically evolved if the workload changes?

RQ1.2: Which API change types exist that affect load test parameterizations?

The most crucial property of representative load tests is their representativeness. Therefore, while we aim to reduce the maintenance effort, our parameterization evolution approach should impair the representativeness as least as possible. Compared to newly generated unparameterized load tests, it should improve this property. Also, we aim at covering generic types of parameterizations, e.g., as demanded by industrial projects. Hence, we investigate the following questions in the evaluation.

RQ1.3: To which degree can we reduce the maintenance effort for the evolution of load test parameterizations if the API changes?

RQ1.4: How much does the parameterization by our approach impair the representativeness of a load test?

RQ1.5: To which degree do evolved parameterizations improve the representativeness of a generated load test?

RQ1.6: How expressive is our approach compared to parameterizations of load tests used in industrial projects?

5.1.2.2. RQ2 — Service-tailoring Load Tests

Existing approaches extract load tests from collected user requests by a series of transformations (Vögele et al., 2018). We aim to extend this process for integrating tailoring to one or a specific set of services. Hence, we address the following question.

RQ2.1: How can we extend the load test extraction process for generating service-tailored load tests?

In the evaluation of the service-tailoring approach, we again focus on the representativeness of the generated load tests. We explicitly investigate the impact on performance metrics, such as response times, CPU utilization, and memory consumption, as an indicator of impaired representativeness. While service-tailoring naturally reduces the number of services to be deployed and, thus, the required hardware, it also might reduce the complexity of the load tests. Hence, the tests might need less time until they reach stable performance metrics, which indicate the tests can be stopped. Finally, we aim at identifying multiple service-tailoring approaches, which we compare qualitatively. Concluding, we evaluate the following research questions.

RQ2.2: How representative are the workloads generated by the service-tailored load tests compared to an untailored and a request-based test?

RQ2.3: To which degree do the service-tailored load tests impair the performance metrics of the tested services?

RQ2.4: How much can service-tailoring reduce the test execution time until measured performance metrics are stable?

RQ2.5: Which qualitative differences of the service-tailored workload models exist?

5.1.2.3. RQ3 — Context-tailoring Load Tests

We aim to generate load tests that are tailored to the workload context relevant to the particular situation. As an example, in early November, the operator of a webshop will be interested in the workload that is expected for the upcoming Black Friday — here, Black Friday is the context —, while they will have a different focus in December. To do so, we need to model the past workload, which will allow us to extract specific parts or predict it to the future. As the past workload continually grows, we need to learn the workload models incrementally, resulting in the following question.

RQ3.1: How can we incrementally learn the workload models from observed user sessions for predicting future workload scenarios?

Incremental learning is an extension of existing approaches, which might affect the workload models. Therefore, we analyze this impact. To describe context-tailored load tests, we introduce a description language that should be expressive enough for real-world scenarios. Then, we apply our approach for generating load tests, whose workload models should be able to predict the future workload accurately. As a further step, we investigate the representativeness of workload scenarios that have been predicted using time series forecasting. Finally, time series forecasting can be time-consuming, which we evaluate as well. Overall, the evaluation addresses the following research questions.

RQ3.2: How much does the incremental learning affect the workload models?

RQ3.3: How expressive is the Load Test Context-tailoring Language concerning workload scenarios of a production system?

RQ3.4: How well do the continuously learned workload models describe the future workload?

RQ3.5: How representative for future workload scenarios are workload models with forecasted intensities?

RQ3.6: How long does it take to calculate an intensity forecast?

5.1.2.4. RQ4 — Enabling Load Testing for Non-experts

For enabling load testing for inexperienced users, we target the integration of our context-tailored load test generation approach and BenchFlow (Ferme and Pautasso, 2018, see Section 4.4.2). This integration provides a high degree of automation, as our approach can generate tailored load tests automatically, and BenchFlow can automatically manage its execution. For easing the interaction of a user with the approach, we develop a description language that is based on template-based natural language. This Behavior-driven Load Testing (BDLT) language — adopted from Behavior-driven Development (BDD) (North, 2006) — needs to be able to describe industrial load test concerns. Also, we investigate how practitioners in the industry would use the language and which benefits and limitations they identify compared to manual scripting of load tests. Finally, we also investigate laboratory use cases, resulting in the following research questions to be evaluated.

RQ4.1: How expressive is the BDLT language in regards to load test concerns of industrial use cases?

RQ4.2: How would BDLT be used in industrial contexts?

RQ4.3: What are the benefits and limitations of using BDLT in comparison to defining load test scripts?

RQ4.4: How expressive is the BDLT language regarding the load test concerns coming from laboratory experiments?

5.2. Assumptions

In this dissertation, we focus on load testing in CSE. Hence, we can presume short release cycles, build, test, and delivery automation, and distributed applications, e.g., following the microservice architectural pattern. Additionally, we make further assumptions that are not necessarily met in such contexts. However, as explained in the following, our research can be extended to relaxed assumptions, e.g., by integration with further approaches.

5.2.1. The Tested Application is Operated in Production

Our first assumption is that the software application, for which we aim to generate load tests, is operated in a production environment and accessed by real users. Hence, we can record the requests and sessions the users submit for extracting load tests. As CSE applies iterative development and aims at releasing the developed software early, we reasonably presume most applications to be productive.

For applications that are not released yet, we cannot apply our approach or require workarounds. One workaround is the use of a related application's workload, e.g., the legacy application when migrating to a distributed cloud environment. Future work might investigate mapping strategies in case of changed APIs.

5.2.2. There is Only One Production Environment

Related to the first assumption, we assume there is a single production environment. Thus, it is unambiguous which data we need to consider to extract load tests. Many industrial applications meet this assumption, e.g., if the application is developed and operated in-house.

However, there can be multiple production environments if the application is delivered to customers or offered as a software as a service (SaaS) solution. In such a case, our approach could be applied in two ways: selecting one production environment — e.g., the one with the highest workload — or considering all environments separately. We leave the precise strategy to the users of our approach or to be investigated in future work and focus on a single production environment in this work.

5.2.3. There is a Production-like Test Environment

Load tests are commonly executed in laboratory environments to prevent influences on the production system (Jiang and Hassan, 2015). Such a test environment should have similar characteristics to the production environment. Otherwise, the load test results achieved with a workload that is representative of the production workload are of minor validity. Therefore, we assume we can execute the load tests we generate in a production-like test environment.

However, such test environments might not exist in practice. To overcome this circumstance, we refer to existing work that introduces testing as a service (TaaS) platforms for production-like deployment of the tested application (Q. Gao et al., 2013; Yan et al., 2012). Alternatively, as most test environments are smaller than the production environment, our approach allows for horizontal (tailoring to specific services and testing only a few service instances) and vertical (reducing the workload intensity) downscaling. Besides, we refer to Foo et al. (2015), who propose an approach for performance regression detection using load tests that have been executed in heterogeneous test environments.

5.2.4. Representative Test Data is Available

In addition to the test environment's hardware and software settings, Jiang and Hassan (2015) also emphasize the importance of test data that is representative of the production data. That is, the database(s) used in the test should be similar to the ones of the production system, as they can influence the application's behavior and, thus, the test results. Therefore, we presume the presence of representative test data in this research.

Often, the actual production database(s) cannot be reused in the test environment due to security or privacy concerns. To overcome this limitation, we refer to existing approaches that construct a test database by obfuscating production data (Barros et al., 2007; Farahbod and Dadashi, 2017; Grechanik et al., 2010).

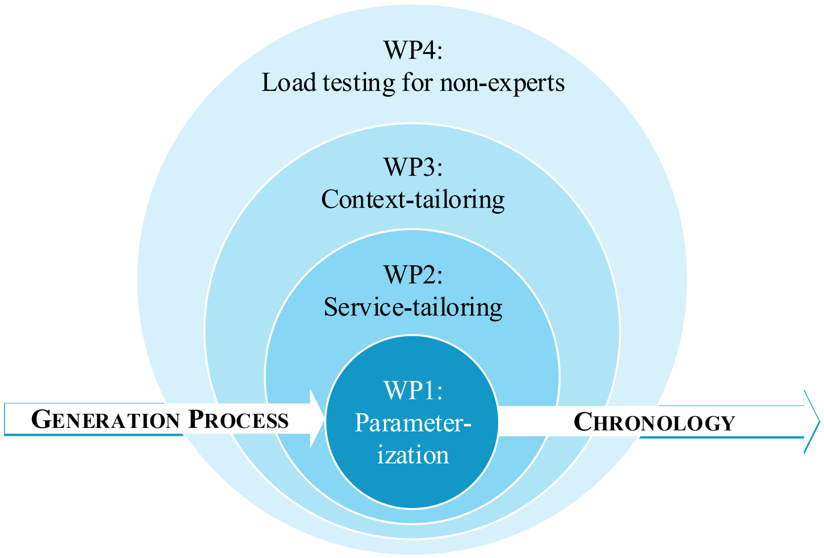


Figure 5.1.: Illustration of the work packages, which we approach in the reverse order compared to the load test generation process.

5.3. Research Plan

We structure our research into work packages that correspond to the main research questions and target the goal defined at the beginning of this chapter. As Figure 5.1 illustrates, we approach the goal in reverse order: while a user first describes a load test, which will be tailored and parameterized, we start with the parameterization and add the other aspects in sequence. Each work package comprises the development of an approach and its evaluation. Following open-science practices, we publish supplementary material for evaluation replication online (see Appendix E).

The work packages differ methodologically. While WP1 focuses on modeling and transformation techniques, WP2 comprises the design of algorithms and their (formal) verification. In WP3, we design a description language and apply data science techniques, such as clustering and time series forecasting. Finally, in WP4, we focus on the application of our approach in industry. The following provides a summary of the work packages.

At the beginning of this research, we have published a vision paper at a relevant workshop (H. Schulz et al., 2018). Besides, as a summary of our research planning, we have prepared a proposal document, which we have refined in collaboration with the Ph.D. supervisor. The vision described in these documents comprises developing performance stubs. Because mature work exists (e.g., Versteeg et al., 2016), we have not investigated this field but focused on load testing for non-experts (WP4) instead.

5.3.1. WP1 — Load Test Parameterzation

The first work package addresses RQ1 and its sub-questions and aims at automating the load test generation process. For that, we develop and evaluate an approach for the automated parameterization of repeatedly generated load tests. We design a domain-specific language (DSL) for describing load test parameterizations, which a user needs to define. Also, we develop transformations of the DSL into load tests, such as JMeter (Apache Software Foundation, 2020[a]) and BenchFlow (Ferme and Pautasso, 2018). To support the user in the creation of parameterizations, we develop evolution strategies for our DSL, as well as a transformation from API specifications. For this purpose, we collect API change types collected in the literature.

The evaluation targets RQ1.3 to 1.6 and comprises experimental studies, estimation models, and a case study. Hence, we combine quantitative and qualitative evaluations.

Chapter 6 describes the approach, and Chapter 12 presents the evaluation.

5.3.2. WP2 — Service-tailoring

Utilizing the automated load test generation process, this work package aims to tailor load tests to a user-defined set of services. We start by analyzing the existing generation process for identifying extension possibilities. For each of them, we develop an algorithm that tailors an intermediate artifact to the services to be tested, such that the resulting load test is service-tailored.

The evaluation comprises formal verification and an experimental study that combines quantitative and qualitative methods. For formal verification, we prove the correctness of the algorithms regarding previously defined requirements. In the experimental study, we apply our approach to a microservice application.

We present the approach and evaluation in Chapters 7 and 13, respectively.

5.3.3. WP3 — Context-tailoring

WP3 aims to tailor the generated load tests in a second dimension, namely the workload context, as requested by RQ3. As the first step, we collect anecdotal examples of workload contexts reported in blog posts and news articles and classify them. Corresponding to RQ3.1, we develop multiple session clustering algorithms for incremental workload model learning. Based on the context classification, we develop an approach for enriching the workload model with contexts. Then, we design a DSL that allows users to describe a context-based load test and develop a process that generates the described test. In doing so, we leverage time-series forecasting.

The evaluation is based on the student information system (SIS) of a large university and addresses quantitative — such as representativeness metrics — and qualitative — such as the DSL's expressiveness — aspects. For that, we conduct an expert survey and experimental studies.

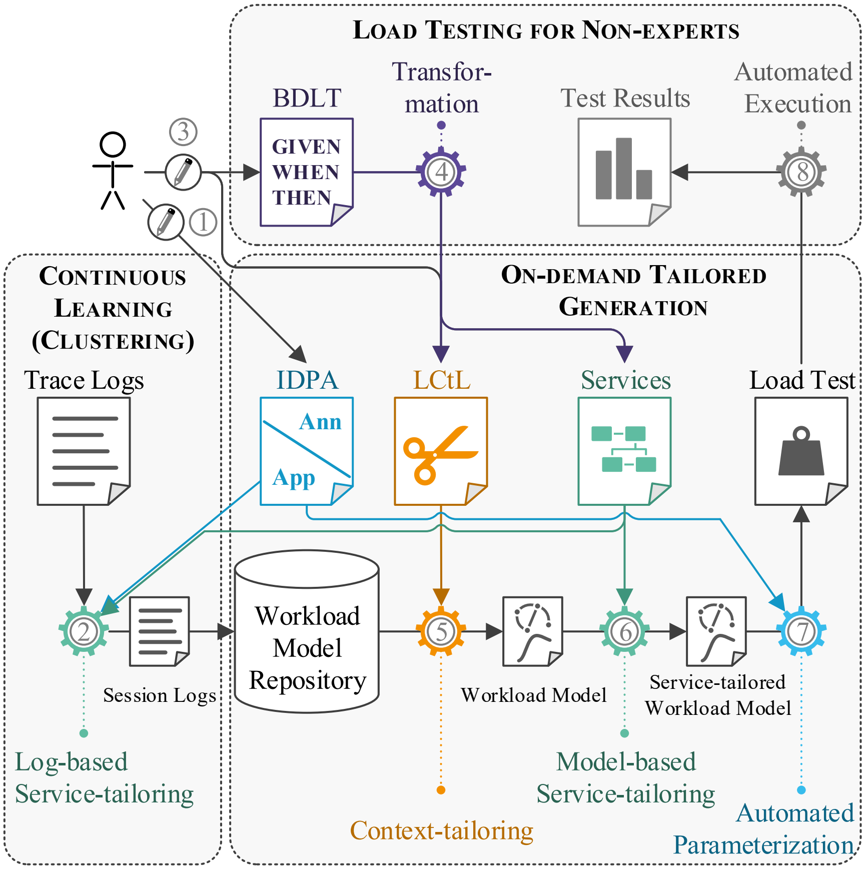
Chapters 8 and 14 present the results of the work package.

5.3.4. WP4 — Load Testing for Non-experts

In the final work package, we collaborate with different researchers to enable load testing for non-experts. In joint work with Ferme and Pautasso (2017, 2018), we integrate our approach with their BenchFlow approach. Hence, we can automatically generate and execute load tests, which abstracts complexity away from the user. Besides, we develop a description language based on template-based natural language for easing the use of our approach further. In a second collaboration with Avritzer et al. (2018, 2020a), we investigate use cases of integrated load test generation and execution. Specifically, we focus on the scalability assessment of microservice applications.

We conduct a qualitative evaluation for investigating the usefulness of our approach in industrial contexts. Precisely, we perform a case study with a project partner. Also, we enrich the evaluation with quantitative insights from the scalability assessment.

We present the approach in Chapter 9 and the evaluation in Chapter 15.



LOAD TESTING FOR NON-EXPERTS
BDLT
GIVEN
WHEN
THEN
Transfor-mation
Test Results
Automated Execution
③
①
④
⑧
CONTINUOUS LEARNING (CLUSTERING)
ON-DEMAND TAILORED GENERATION
Trace Logs
IDPA
Ann
App
LCtL
Services
Load Test
②
Session Logs
Workload Model Repository
⑤
Workload Model
⑥
Service-tailored Workload Model
⑦
Log-based Service-tailoring
Context-tailoring
Model-based Service-tailoring
Automated Parameterization

Figure 5.2.: Overview of our approach.

5.4. Overview of the Approach

Figure 5.2 shows an overview of the approach, which we have developed in the work packages described above. There are three major constituents: the continuous learning of a context-enriched workload model, the on-demand generation of a tailored load test, and load testing for non-experts, which integrates the on-demand generation with the automated load test execution by Ferme and Pautasso (2018). In the following, we detail the constituents as a summary of the remaining chapters of this part. In doing so, we consider a webshop as an exemplary microservice application to be load tested.

API and Parameterization Description: The first step is always the creation of an Input Data and Properties Annotation (IDPA) ①. An IDPA is a model that describes the API of the system under test (SUT) and the parameterizations to be applied to generated load tests. While we can automatically generate the API description based on OpenAPI specifications (OpenAPI Initiative, 2020) — which are commonly used in CSE —, the user has to define the parameterizations manually. Regarding the webshop, they have to specify the credentials of test users the load test should use when requesting the login endpoint. For later use, the parameterizations are connected with endpoint IDs, such as *loginUsingGET*. The IDPA is to be maintained by the developers of the tested application or service. To reduce the maintenance effort, we provide feedback-based evolution strategies for the IDPA in case of API changes and reuse it for repeated load test generations. *Developing the IDPA and its evolution mechanisms are part of WP1.*

Continuous Workload Model Learning: Using the API description of the IDPA, we continuously learn a workload model. For that, we collect the trace logs from our application running in the production environment — e.g., by application performance management (APM) — and extract user sessions. This extraction considers the IDPA for assigning IDs to the endpoints of the traces' requests, which allows us to synchronize the later generated load tests with the parameterizations of the IDPA. For example, we label requests to the login endpoint with *loginUsingGET*. If configured, we apply *log-based service-tailoring* ②, which is one approach we have identified in WP2. It tailors the session logs to specific services, e.g., the *users* service of our webshop, by replacing the root endpoints of the traces with those directly targeting the users service. Then, we cluster the extracted sessions incrementally, i.e., we integrate former clusterings. The resulting workload model is stored in a workload model repository (WMR). Also, we enrich the workload model with contextual information, e.g., the fact that there was a *Black Friday* at a specific date. *The request labeling is part of WP1, the log-based service-tailoring belongs to WP2, and the incremental workload model learning contributes to WP3.*

Behavior-driven Load Test Definition: For non-experienced users, we provide an integration with the BenchFlow load test execution automation by Ferme and Pautasso (2018). The users can define a load test using our provided BDLT language ③, as illustrated by Listing 5.1. The example states to test the workload expected at the next Black Friday — which relates to the Black Friday context we have stored in the WMR — tailored to the users service for testing how many instances of this service need to be deployed to ensure response times below a defined threshold. The BDLT definition is readable as natural language, which eases communication with non-technical stakeholders, such as product owners. To generate the described test, we transform the BDLT definition into other artifacts ④. *The BDLT language and its transformation were developed as part of WP4.*

```
Given the next Black Friday
and the service users,
when varying the number of users instances between 8 and 12,
then ensure the 95th percentile response time is less than 1 second.
```

Listing 5.1: Exemplary BDLT definition.

Context-tailoring: The first step for the on-demand generation of a tailored load test is context-tailoring ⑤. It extracts a workload model from the workload model repository by applying time-series forecasting. For that, a user needs to define an instance of the LCtL; if they have used the BDLT language, the LCtL instance is generated automatically. As an example, the LCtL instance corresponding to Listing 5.1 would state to extract a workload model from the time frame where the *Black Friday* context is present. To achieve that, we apply multivariate time-series forecasting tools (Bauer et al., 2020; Taylor and Letham, 2018), which consider the context, to the past workload we have learned incrementally. The result is a WESSBAS (Vögele et al., 2018) model that reflects the expected number of users and their behavior for the upcoming Black Friday. The benefit of using the LCtL instead of the BDLT language are broader and more fine-grained configuration possibilities, while it requires more technical experience, e.g., for understanding the *YAML* (2020) format. *The context-tailoring is part of WP3.*

Model-based Service-tailoring: We need to apply *model-based service-tailoring* ⑥ if we want to test the users service of our webshop directly but have not used log-based service-tailoring during the incremental workload model learning. Hence, it constitutes the second option we have developed for service-tailoring. It takes as input the context-tailored workload model, which targets the webshop application as a whole, and replaces individual states with new states that target the users service. The services to be considered are defined as a simple user-specified list; when using BDLT, we generate the list. The output of this step is a model that reflects the same workload as the input did but tailored to the users service. *The model-based service-tailoring belongs to WP2.*

Automated Parameterization: For transforming the generated workload model into an executable load test, we need to parameterize it ⑦. We need to add input data, such as the user credentials for the login endpoint, and configure the test to be executable in a test environment, e.g., by specifying the correct IP address and port number. These parameterizations are part of the IDPA the user has defined in the very beginning. Because the previous transformations preserved the endpoint IDs, such as *loginUsingGET*, which we have assigned to the requests of the session logs, we can map the parameterizations to the states of the workload model. Hence, we fully automate the on-demand load test generation, as opposed to existing work (Vögele et al., 2018). The load test formats we support are JMeter (Apache Software Foundation, 2020[a]) and BenchFlow (Ferme and Pautasso, 2018), but further formats can be added. *The automated parameterization is part of WP1.*

Automated Execution: Our load testing approach for non-experts also comprises the automated execution of the generated load test ⑧. Hence, we can provide the user with the load test results only based on their BDLT definition (and a previously defined IDPA). In this research, we do not contribute to the load test execution but rely on the framework by Ferme and Pautasso (2018).

5.5. Collaborations

During our research, we have collaborated with multiple students, researchers, and industry partners. The following provides an overview of these collaborations.

5.5.1. Students

Several students contributed to this research while preparing their Master's theses at the University of Stuttgart. As detailed below, these students were co-supervised with the following researchers: André van Hoorn, Dušan Okanović, Vincenzo Ferme, and Petr Tůma (Charles University, Prague).

- Angerstein (2018) has developed the service-tailoring approaches from WP2 (Chapter 7) and has performed an initial experimental evaluation, which we have extended (Chapter 13). Also, we have extended the used formalisms. His work resulted in a joint publication (H. Schulz et al., 2019a).

- Hidiroglu (2019) has contributed to WP3 by developing an initial version of the context-tailoring (Chapter 8), comprising a context description language. Some concepts of this language came into our LCtL. He has also done exploratory work for our evaluation (Chapter 14) by using the same dataset in his experimental evaluation. His research was conducted as a collaboration with Charles University, Prague.
- Contributing to WP4 (Chapter 9), Palenga (2018) has developed the transformation of WESSBAS workload models and IDPAs into BenchFlow load tests. Besides, he has implemented a BenchFlow execution engine based on JMeter.

5.5.2. Researchers and Industry

Our research was part of the *ContinuITy* (2020) project. Throughout our research and all publications, we have collaborated with colleagues from the Reliable Software Systems group of the University of Stuttgart and Novatec Consulting GmbH.

Regarding the context-tailoring (WP3, Chapter 7), we have furthermore collaborated with Charles University, Prague. Particularly, Maňásek and Tůma (2019) have contributed a large dataset of user requests, which we have used in our evaluation (Chapter 14). Besides, we have jointly supervised Hidiroglu's thesis. The collaboration resulted in a joint publication (H. Schulz et al., 2021).

WP4 (Chapters 9 and 15) comprises multiple collaborations. First, the BDLT language and integration of our and the BenchFlow approach was a joint work with the University of Lugano, whereas Ferme was a visiting researcher at the University of Stuttgart during the collaboration. It resulted in a joint publication (H. Schulz et al., 2019c). For evaluating our integrated approach, we have collaborated with an industry partner of the ContinuITy project (Section 15.1). Besides, we have developed the domain-based approach for microservice scalability assessment (Section 9.3) in collaboration with multiple researchers from EsulabSolutions, Inc, Free University of Bozen-Bolzano, and Federal University of Rio de Janeiro, resulting in several publications (Avritzer et al., 2018, 2020a, 2019).

Chapter 6

Automating Load Test Parameterization

A fundamental requirement for our approach is the ability to generate load tests automatically without manual intervention. Based on a given contextual description and a set of services to be tested, we generate tailored load tests that represent the expected workload for the context and directly operate on the set of services. Existing approaches allow extracting load tests with representative workload specifications from recorded request logs (Barros et al., 2007; Cai et al., 2007; Krishnamurthy et al., 2006; Lutteroth and Weber, 2008; Menascé and Almeida, 2002; Ruffo et al., 2004; Vögele et al., 2018) but require manual parameterization of the generated load tests (Vögele et al., 2018). The parameterization includes adjusting static properties like the domain name or port number, which will differ in the test environment, and input data for the individual requests. For instance, an expert needs to specify the user names and passwords for a login request according to the test environment's database. Additionally, they have to consider data dependencies such as IDs or tokens the system under test (SUT) returns for the use in later requests. As described by T.-H. Chen et al. (2017) and Jiang and Hassan (2015), maintaining such generated load tests is an existing research challenge. Because of the evolution of the production workload and the SUT's API, the generated load tests need to be evolved simultaneously to stay representative. As an additional challenge, our approach requires parameterizing newly generated load tests automatically, e.g., as part of a continuous integration and delivery (CI/CD) pipeline.

Existing approaches determine whether an expert should update a generated load test by comparing the workload of a load test execution to the production workload (T.-H. Chen et al., 2017; Syer et al., 2014, 2017). Commercial load testing tools support users in defining parameterizations (Micro Focus, 2020[a],[b]). However, these approaches are not capable of dealing with frequently generated load tests as by our approach. Because of the sheer amount of combinations of contexts and sets of services that a load test could simulate, generating load tests once and replacing them when needed requires significant manual effort for parameterizing the newly generated load tests. Also, multiple generated load tests will be parameterized similarly, e.g., with the same user names and passwords. Furthermore, the requirement for generating load tests fully automatically disallows for parameterizing them manually. The commercial tools can support reducing the manual effort but still cannot automate the parameterization.

Therefore, we aim at decoupling the manual effort required for parameterization from the load test generation process and reusing once defined parameterizations for multiple load tests. Furthermore, we want to reduce the manual effort to a minimum. Our approach needs to evolve once created parameterizations over changing workloads and APIs of the tested application. Thus, we address the research question RQ1 defined in Section 5.1: *How can load test parameterizations be evolved without manual intervention at test generation or execution time?*

For this purpose, we introduce the reusable Input Data and Properties Annotation (IDPA) for load test parameterization separated from a generated load test. An IDPA stores the parameterizations defined by an expert and can be automatically merged into the generated test. Hence, it is reusable for different workloads. For the evolution over API changes, we analyze API change types proposed in the literature and design a feedback-based approach to the evolution of IDPAs over the identified changes. As a result, the load test generation process can be fully automated because IDPAs can be defined and updated in advance.

The remainder of this chapter is structured as follows. Section 6.1 summarizes possible load test parameterizations we consider for the design of the IDPA in a taxonomy. In Section 6.2, we describe the load test parameterization process using IDPAs. In Section 6.3, we introduce the IDPA metamodel and serialization. Section 6.4 depicts the evolution of IDPAs over workload and API changes. Section 6.5 provides the IDPA transformations used in our approach. Finally, in Section 6.6, we describe how the load test parameterization with IDPAs integrates into continuous software engineering (CSE).

Parts of this chapter have been published in advance in our following publications, as marked in the text:

- H. Schulz, T. Angerstein, and A. van Hoorn (2018). "Towards Automating Representative Load Testing in Continuous Software Engineering." In: *Companion of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018)*. ACM, pp. 123–126
- H. Schulz, A. van Hoorn, and A. Wert (2020c). "Reducing the Maintenance Effort for Parameterization of Representative Load Tests Using Annotations." In: *Journal of Software Testing, Verification and Reliability* 30.1

6.1. A Taxonomy of Parameterizations

For illustrating required parameterizations of generated load tests and motivating the need for automation, this section introduces a taxonomy of load test parameterizations. To account for as many parameterization concepts as possible, we base the taxonomy on multiple load tests used for the case study systems in our evaluations (see Chapter 12). Precisely, we refer to the following: the load tests of four industrial projects, which we analyzed regarding the parameterizations used; a load test derived from recorded

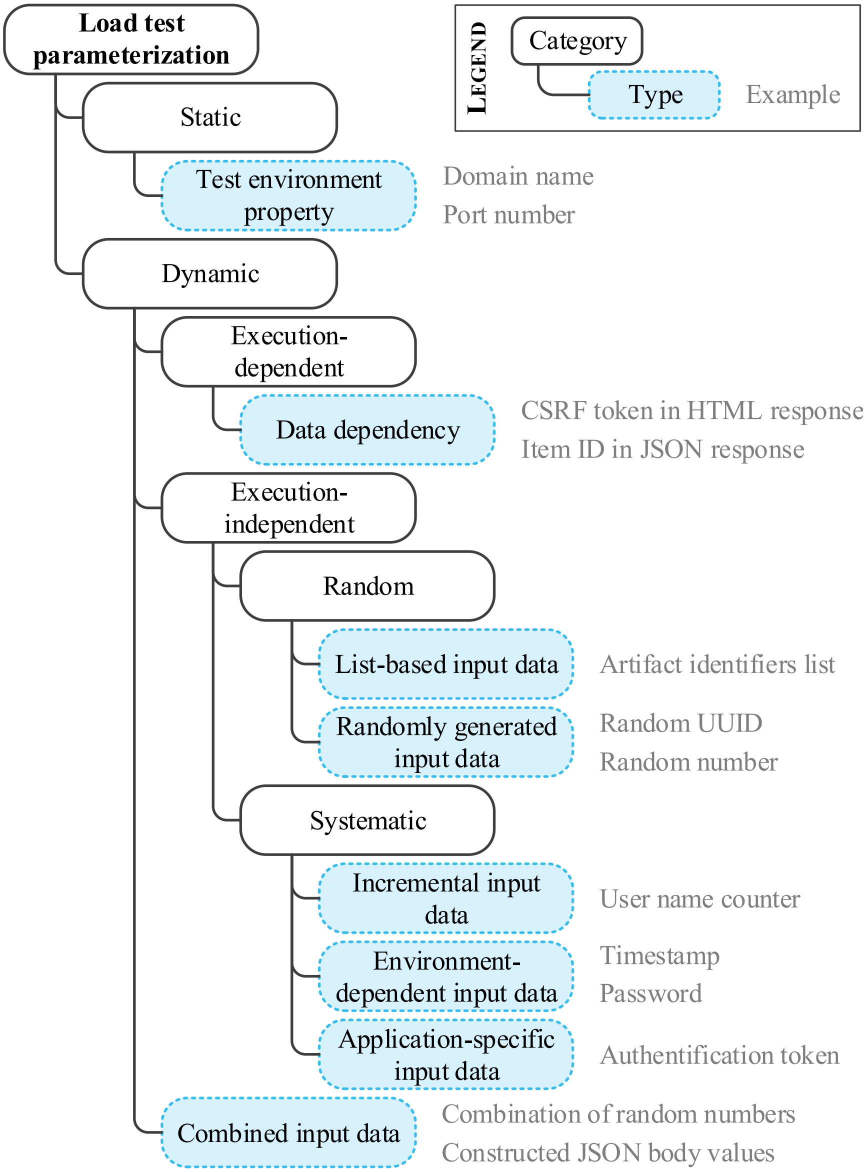


Figure 6.1.: Taxonomy of load test parameterizations.

request logs for the build artifact management system Nexus (Sonatype, Inc., 2020[a]); and a load test we defined for the e-commerce sample application Heat Clinic (Broadleaf Commerce, LLC, 2017). We presume that these load tests cover the most frequently used concepts. However, there can still be parameterization concepts that any of these load tests use. Therefore, we respect the potential incompleteness by providing extension possibilities for our models.

We illustrate the taxonomy in Figure 6.1. We divide the load test parameterizations into two main categories of *static* and *dynamic* ones. Static parameterizations are *test environment properties* that need to be specified, such as the domain name and port number of the SUT. Especially these two properties need to be specified in all cases where the extracted load test is to be executed in a test environment that differs from the production environment. In the industrial projects, we furthermore identified a need for changing the base path of requests, e.g., by adding */stage/test* as a prefix.

The category of dynamic parameterization denotes the specification of dynamic values such as input data to be used for individual requests. We further divide it into *execution-dependent* and *execution-independent* parameterizations. Execution-dependent parameterizations refer to *data dependencies*, which the load test needs to resolve during the execution. Typical examples are IDs or tokens that the target application generates and returns as a part of a response, which the (simulated) users need to reuse in the following requests. For instance, different web applications such as the Heat Clinic use cross-site request forgery (CSRF) tokens for preventing security vulnerabilities. The users need to keep this token in subsequent requests (OWASP, 2020). Another example originates from the industrial projects, where the virtual users extract an item ID from a JavaScript Object Notation (JSON) response for later use.

Execution-independent parameterizations are input data that are specified independently of the actual execution order of the requests of the load test but are dynamically determined as well. Such input data can be either defined *randomly* or *systematically*. As random parameterizations, we identified *list-based input data*, which denote lists of values specified in comma-separated values (CSV) files or directly in the load test, and from which the virtual users take parameter values for the requests randomly. For Nexus, we used CSV files holding artifact identifiers that the load test should access in random order. Another type of random parameterization is *randomly generated input data*. In contrast to list-based input data, values are entirely generated based on, e.g., a pattern or a numerical range. Two examples from the industrial projects are universally unique identifiers (UUIDs), which were randomly generated based on a pattern, and numbers that were randomly chosen from a range.

Opposed to random parameterization, systematic execution-independent parameterization comprises input data that are systematically defined. *Incremental input data* are a compact and systematic specification of numerous values that only differ in a number. The number is changed, e.g., by counting the number of starts of new virtual users, for ensuring each of them has a different number. As an example, we made use of this concept for defining 200 different user accounts for the Heat Clinic, which only differ in a number. Each time a new user starts, the counter increases, and hence, the user uses a new account. Furthermore, the industrial projects use *environment-dependent input data*. In particular, they generate timestamps based on the current time and read passwords from environment variables. The final parameterization type falling into this category is *application-specific data* such as an authentication token, which one of the industrial projects used. In this case, they generated a Java Web Token (JWT) initially and then used it during the load test execution. Because inputs of this category can highly depend on the SUT, they can hardly be generalized but rather demand for extensible input data specifications.

Finally, there is a type of dynamic parameterization that cannot be classified to be execution-dependent or independent. These are *combined input data*, which denote input data consisting of a combination of other input data. As an example, the load tests of one of the industrial projects combine multiple randomly generated numbers to a string. Additionally, several of these projects construct JSON body values based on various other input data, such as list-based input data and data dependencies.

Execution-independent parameter values can often be learned from production requests. However, they are typically not reusable as input data for the load test because the test database differs from the production database. Hence, they have to be replaced by values that are suitable for the test environment. For execution-dependent parameter values, commercial load testing tools provide support (Micro Focus, 2020[a],[b]) but typically require at least manual guidance. Our approach takes into account that some or even all parameterizations of a load test need to be applied manually. Based on the taxonomy, we develop the IDPA, which stores the parameterizations separated from generated artifacts such as workload models and load tests. The IDPA allows for reusing parameterizations when workloads and APIs evolve. Accounting for the potential incompleteness of the taxonomy, the IDPA is extensible.

In the next two sections, we describe the process of load test parameterization with and without using our approach and present the IDPA in detail.

6.2. Load Test Parameterization Process Overview

This section describes the process of parameterizing generated representative load tests using IDPAs compared to solely using existing approaches. We published the process in previous work (H. Schulz et al., 2018), and Figure 6.2 illustrates it.

Without using IDPAs, the process consists of workload characterization and transformation to a load test, as described in Section 3.2.1 (solid black arrows in the figure). In a first step, state-of-the-art workload characterization approaches use request logs recorded from a production system for generating a workload model that represents the actual production workload ①. An example of such an approach is WESSBAS based on Markov chains (Vögele et al., 2018). The second step is a transformation of the generated workload model into an executable load test, e.g., a JMeter test plan (Apache Software Foundation, 2020[a]). Typically, an expert has to attend this transformation to parameterize the load test ②.

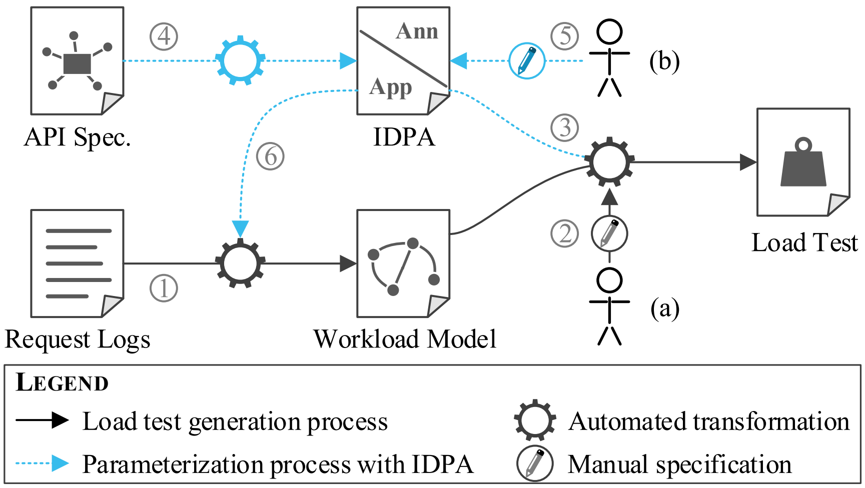


Figure 6.2.: Generation of representative load tests using existing approaches with manual parameterization (a) and parameterization with an IDPA (b).

The main drawback of this parameterization approach is that it has to be applied at the time a load test is generated. It does not allow for reusing parameterizations for multiple generated load tests, e.g., representing different workload scenarios. Hence, manual effort is required each time a load test is generated. In the context of our approach, the manual effort is especially critical because we generate load tests based on high-level descriptions directly before testing. As a consequence, the manual effort accumulates and prevents integration into CI/CD pipelines.

For these reasons, we extend the parameterization process, as shown by the dashed arrows in the figure. The manual parameterization in step ② is replaced by automatically merging an IDPA holding all parameterizations into the generated load test ③. Consequently, the transformation chain from request logs to a load test is entirely automated. For proper merging into the load test, an IDPA consists of two parts. First, it holds an application model describing the API of the tested application, including the endpoints and parameters. Our approach can transform the application model from API specifications such as OpenAPI (OpenAPI Initiative, 2020) ④. This transformation allows for feedback-based updates of an IDPA in case of API changes. The second part of an IDPA is an annotation that constitutes the actual parameterization by annotating the application model elements. Annotations are to be defined by experts, who, however, can do this offline from the load test generation process ⑤. For ensuring proper recognition of API elements, we utilize the application model of an IDPA for labeling the individual requests of the request logs ⑥.

Based on the illustrated process, we describe the details of an IDPA in the following section.

6.3. The Input Data and Properties Annotation

In the two sections before, we derived a taxonomy of load test parameterizations and illustrated how we separate parameterizations from generated artifacts by using an Input Data and Properties Annotation (IDPA), which we base on the taxonomy. In this section, we describe the IDPA in detail. First, in Section 6.3.1, we depict the underlying concepts we applied when designing the IDPA. Section 6.3.2 presents the metamodel of the IDPA. Finally, in Section 6.3.3, we illustrate the serialization of IDPAs in the YAML format (*YAML* 2020) and provide examples.

6.3.1. Underlying Concepts

We designed the IDPA considering the following concepts *(based on H. Schulz et al., 2020c)*.

Separation of automatically and manually created artifacts. The main advantage of the IDPA is the separation of automatically generated workload models or load tests and manually defined parameterizations. We pursue this concept of separating automatically and manually created artifacts in the internals of the IDPA as well. An annotation needs to refer to the endpoints and parameters of the target application's

API. In practice, API specifications like OpenAPI (OpenAPI Initiative, 2020) are often used to describe REST APIs. Hence, specifying the API a second time manually in an IDPA constitutes unnecessary effort. For this reason, we divide the IDPA into an application model that holds information about the API and can be generated automatically from API specifications and an annotation model holding the manual specifications.

Separation of "what" and "where". Input data specifications consist of two essential attributes: "what" data to be used and "where" to place it. The "what" could be list-based input data such as the product list described in Section 6.1 and the "where" the product parameter of the product details endpoint. For better understandability and to avoid duplicated information, we separate "what" and "where". As an example, we presume the product list to be needed for several endpoints, e.g., product details and add to cart. By specifying the product list once and referring to it for all endpoints, we prevent redundant specification of the extraction.

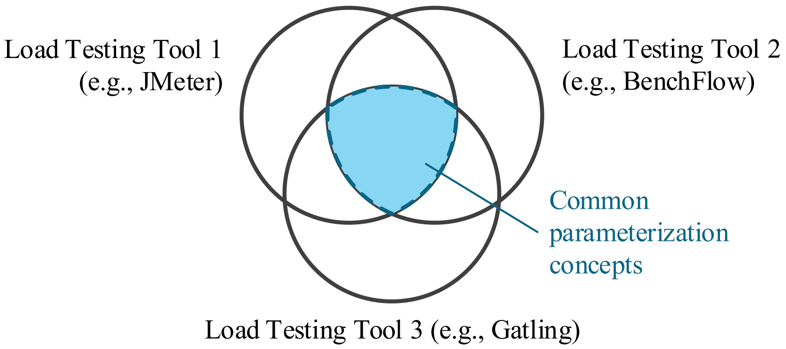


Figure 6.3.: Illustration of the relation of the parameterization concepts of different load testing tools.

Tool independence. We designed the IDPA for usage with different workload characterization approaches and load testing tools. Thus, we cannot integrate all possible parameterization concepts. This is illustrated in Figure 6.3. Considering three different load testing tools — e.g., JMeter (Apache Software Foundation, 2020[a]), BenchFlow (Ferme and Pautasso, 2018), and Gatling (Gatling Corp, 2020) —, each of them provides a different set of parameterization concepts. If we integrated a concept from one tool into the IDPA that has no equivalent in one of the other tools, we would lose the tool-independence. However, there is a large overlap of common concepts. As an example, input data retrieved from CSV files and regular expression extractions exist in all tools. Thus, we focus on the common parameterization concepts but allow using further ones as extensions of the IDPA.

Extensibility. For still enabling any parameterization — e.g., parameterization that is specific to the load testing tool or the SUT —, the IDPA provides several extension points. We focus on a concept and a reasonable subset of all possible parameterizations that are sufficient for our evaluations and allow us to extend the IDPA easily. Adding further concepts missing from the practical and scientific load testing projects on which we have developed the IDPA can be done using the extension points.

Traceability. To evolve IDPAs over changes in the workload and the target application's API, we need to be able to trace the evolution of single elements of an IDPA, especially the application model. For this purpose, each element holds a unique ID.

Integration with commonly-used technologies. IDPAs are to be used in the context of CI/CD pipelines. Therefore, we serialize an IDPA in the YAML format (*YAML* 2020), which state-of-the-art technologies such as Docker (Docker Inc., 2020) and OpenAPI (OpenAPI Initiative, 2020) use, too.

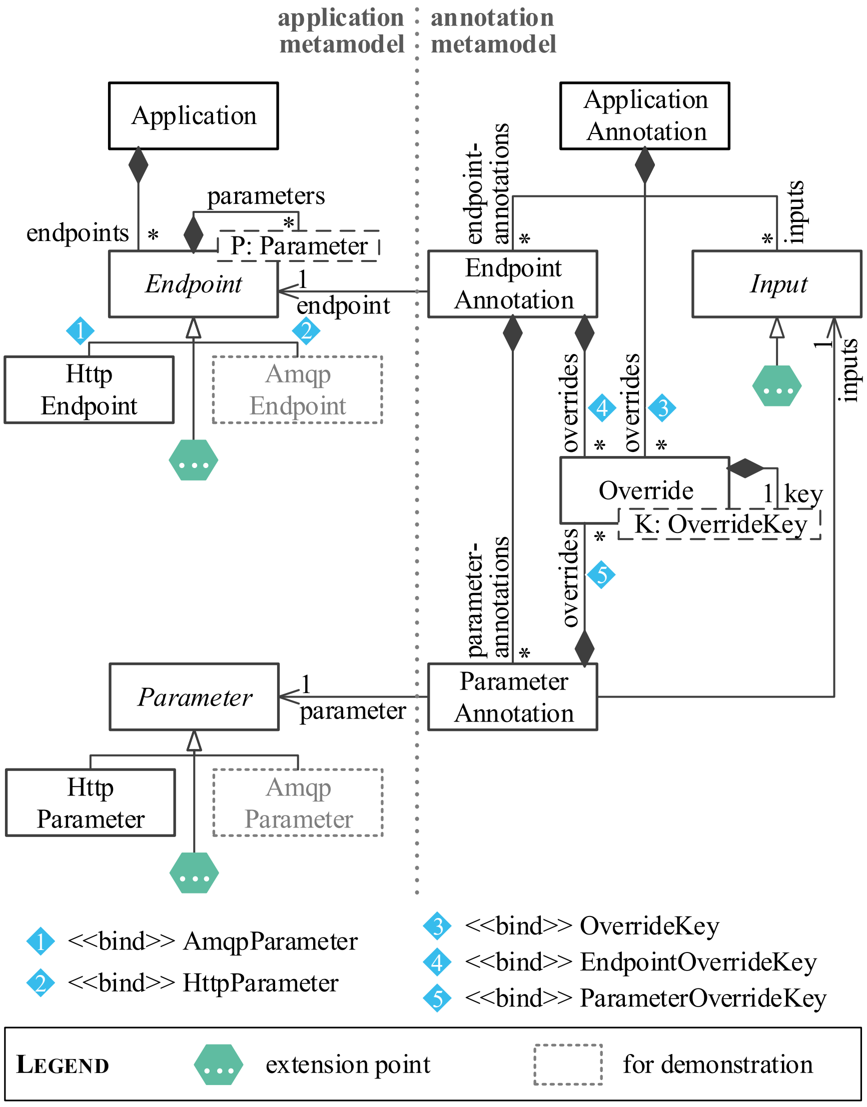


Figure 6.4.: Metamodel of the Input Data and Properties Annotation (based on H. Schulz et al., 2020c).

6.3.2. Metamodel

Figure 6.4 shows the core metamodel of the IDPA. Figures 6.5 to 6.9 provide further details. We base the metamodel on the taxonomy depicted in Section 6.1 and the concepts defined in Section 6.3.1. Hence, we divide it into an application metamodel and an annotation metamodel, as illustrated in Figure 6.4. Application models describe an SUT's API and are meant to be generated from API specifications such as OpenAPI. Annotation models hold the manual parameterizations summarized in the taxonomy.

We reach the separation of "where" and "what" by defining the dynamic parameterizations of the taxonomy in `Inputs`, which a `ParameterAnnotation` then maps to a `Parameter`. Hence, the same Input can be reused for multiple Parameters. By only integrating core functionality, we ensure tool independence but allow extension of the application metamodel through extension points at `Endpoint` and `Parameter` and extension of the annotation metamodel through extension points at Input and specific `OverrideKeys`.

In the following, we describe the individual elements of the metamodel. *We already published most of these elements in previous work (H. Schulz et al., 2020c). Others that Angerstein (2018) has introduced or we introduce newly in this dissertation are marked accordingly.*

6.3.2.1. Application Metamodel

The application metamodel is shown on the left side of Figure 6.4 and consists of the core elements `Application`, `Endpoint`, and `Parameter`. `Endpoint` and `Parameter` have specific implementations — for the protocols Hypertext Transfer Protocol (HTTP) and Advanced Message Queuing Protocol (AMQP).

Application: The `Application` is the core element of the application metamodel. It represents an application, whose load tests are to be parameterized. It holds a set of `Endpoints` comprising the application's API. Furthermore, for tracing evolving API versions, it holds either a timestamp representing the date when the API version was introduced or the corresponding API version.

Endpoint: The `Endpoint` is an interface representing a single entity of an API. It has a generic type as a subtype of `Parameter`, to which it also holds a reference *parameters* of indefinite cardinality.

Parameter: A `Parameter` denotes an endpoint parameter, which needs to be served with input data when submitting a request to the endpoint. According to the `Endpoint` implementations, there can be different implementations of `Parameter`.

HttpEndpoint and HttpParameter: Because we mainly focus on load testing of web applications, we provide `Endpoint` and `Parameter` implementations `HttpEndpoint` and `HttpParameter` belonging to each other. That is, the type of parameters the `HttpEndpoint` holds is `HttpParameter`. The `HttpEndpoint` is characterized by a *domain name* and *port number* (e.g., *www.mydomain.org:80*), a *path* (e.g., */login*), an *HTTP method* (e.g., *GET*), an optional *content encoding* (e.g., *UTF-8*), a *protocol* (*http* or *https*) and a list of *headers*. The `HttpParameter` has a *name* (e.g., *user*) and a *parameter type*. The parameter type has to be one of the following:

req-param A request parameter, also known as a query string parameter. It is added to the end of the URL starting with a '?', e.g., *?user=Jane*.

body The body of a POST, PUT, or PATCH request.

url-part A parameter that is encoded as part of the URL path, resulting in a dynamically defined path, e.g., */details/{product-id}* where *product-id* is the parameter. Also, regular expressions can be used for restricting the allowed values of the parameter. For instance, the following path only accepts numbers as *product-id*: */details/{product-id:\d+}*.

header A header to be set dynamically.

form A body parameter holding form data of type *multipart/form-data* or *application/x-www-form-urlencoded*.

AmqpEndpoint and AmqpParameter: Users of our approach can extend the IDPA by adding more implementations of `Endpoint` and `Parameter` without affecting the remainder of the IDPA. As a demonstration, Figure 6.4 holds the corresponding `Endpoint` and `Parameter` implementations `AmqpEndpoint` and `AmqpParameter` to be used for modeling AMQP queues such as RabbitMQ (Pivotal Software, Inc., 2020[a]). The endpoint implementation could be characterized by an AMQP *host* and *exchange* or *queue name* and would hold `AmqpParameters`. An `AmqpParameter` could represent the *message* or a *message header*. All entities of the annotation model could be combined with the AMQP entities, similar to the HTTP entities.

6.3.2.2. Annotation Metamodel

We design the annotation metamodel to represent the parameterization concepts described in the taxonomy in Section 6.1. The metamodel consists of three major parts. For the static parameterization, we introduce `Overrides`, which allow specifying static test environment properties such as the domain name or port number, which will override the respective property in the generated load test. For the dynamic parameterization, `Inputs` define input data that the load test should use for the parameters of the requests submitted in the load test. `EndpointsAnnotations` and `ParameterAnnotations` map `Inputs` and `Overrides` to specific `Endpoints` and `Parameters`.

We include all parameterization concepts of the taxonomy into the metamodel, except for authentication data. Authentication data are highly dependent on the target application and, therefore, cannot be generalized for the use with any application. Thus, we exclude it from the IDPA. Instead, we provide extension points, which a user of our approach can utilize for implementing application-dependent `Inputs` such as authentication data inputs.

In the following, we depict all entities of the annotation metamodel and how they relate to the taxonomy.

ApplicationAnnotation: The `ApplicationAnnotation` is the core entity of the annotation metamodel and is illustrated in Figure 6.4. It implicitly refers to an `Application` that is parameterized and holds the references *inputs*, *overrides*, and *endpoint-annotations*. The *inputs* are a set of `Inputs`, which constitute the input data specifications. The *overrides* hold application-level `Overrides`. We will explain the differentiation between application-, endpoint-, and parameter-level *overrides* in the following. *Endpoint-annotations* is a set of `EnpointAnnotations` mapping the `Inputs` and `Overrides` to `Endpoints` of the API of the parameterized `Application`.

EndpointAnnotation: An `EndpointAnnotation` refers to one `Endpoint` of the parameterized `Application` and contains a set of `ParameterAnnotations`, which refer to `Parameters` of the `Endpoint`. Furthermore, its *overrides* allow overriding endpoint-level properties.

ParameterAnnotation: A `ParameterAnnotation` maps one `Input` to one `Parameter`. That is, a virtual user of a load test should use the input data defined by the `Input` when submitting requests to the respective endpoint. Besides, the annotation holds a set of parameter-level `Overrides`.

Override: For specifying test environment properties, we introduce the `Override`. It is parameterized with a generic subtype of `OverrideKey`. The type defines the scope of the `Override`, which we explain below. An `Override` refers to one `OverrideKey`, which defines the *property* to be overridden. The new value to be used is defined as a string. `Overrides` can be defined in `Application-`, `Endpoint-`, and `ParameterAnnotations` (Figure 6.5).

OverrideKey and Sub Types: `OverrideKey` (Figure 6.5) is a common interface for defining test environment properties that are to be overridden. Because there are different properties at different levels, we utilize a hierarchical inheritance structure. `EndpointOverrideKey` extends `Over-`

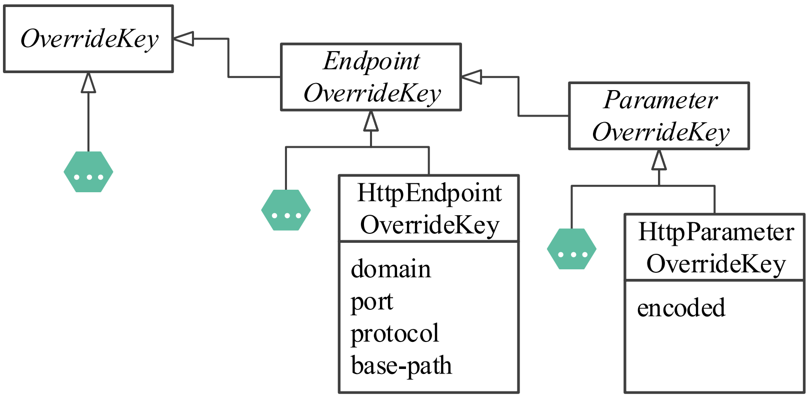


Figure 6.5.: Metamodel of IDPA overrides (based on H. Schulz et al., 2020c).

`rideKey`, and `ParameterOverrideKey` extends `EndpointOverrideKey`. This inheritance structure allows defining `Overrides` referring to low-level entities in higher-level annotations. For instance, a user can define an `Override` using an `EndpointOverrideKey` in `Endpoint`- and `ApplicationAnnotations`, but not in `ParameterAnnotations`. Specifying an `Override` in higher-level annotations means that all instances of the property the `OverrideKey` defines are overridden in the scope of the annotation, e.g., all hostnames of all `Endpoints` of the annotated `Application`.

Enumerations implementing the respective interfaces define the specific `OverrideKeys` that are available. While users or future researchers can add new enumerations, we provide `OverrideKey` implementations for HTTP endpoints and parameters.

HttpEndpointOverrideKey and HttpParameterOverrideKey: The `HttpEndpointOverrideKey` and `HttpParameterOverrideKey` are `OverrideKey` enumerations referring to HTTP properties. `HttpEndpointOverrideKey` defines properties that can be overridden at the endpoint level, namely the *domain*, *port* number, *protocol*, and *base-path* of an endpoint. These keys are especially useful, as load tests are generated based on production request logs, which contain the production domain name and port number. Hence, these properties are to be overridden. Overrides with the *base-path* key need to be in the format *n/path* — e.g., *2/new* —, which will override the first *n* segments of the original path, e.g., change */old/path/login* to */new/login*. On the parameter level, we provide the *encoded* key as part of `HttpParameterOverrideKey`, which a user can use for ensuring that parameter values are properly encoded for the use in URLs.

Input: `Input` is a common interface for input data specifications, i.e., dynamic parameterizations. Figures 6.6 to 6.9 illustrate the implementations. In separating "what" and "where", an `Input` defines the "what". While we already provide implementations for the most common types of input data specifications, `Input` constitutes an extension point for new implementations. When adding a new implementation, the remainder of the IDPA metamodel does not have to be adjusted, because `ParameterAnnotations` are independent of the specific `Input` implementation. In the following, we depict the provided implementations. Because well-known load testing tools already provide mature means for input data specification, several `Input` implementations base on these. Specifically, we rely on existing input concepts of the JMeter tool (Apache Software Foundation, 2020[a]). However, the `Input` concepts provided are themselves tool-independent.

ExtractedInput: As described in the taxonomy, there can be data dependencies between different requests of a load test. That is, individual requests need to use a value — e.g., an ID or token — responded by the SUT to a previous request. Therefore, we provide the `ExtractedInput`, which is illustrated in Figure 6.6, for specifying such data extractions. It represents a value that is extracted by a set of `ValueExtractions`. A `ValueExtraction` can either be a `RegExExtraction` or a `JsonPathExtraction`, for extracting values based on regular expressions or JSON paths. Each time one of the `ValueExtractions` extracts a value, it overwrites the value of the `ExtractedInput`. Furthermore, a user can optionally define an *initial value* for specifying the value to be used before the first value has been extracted.

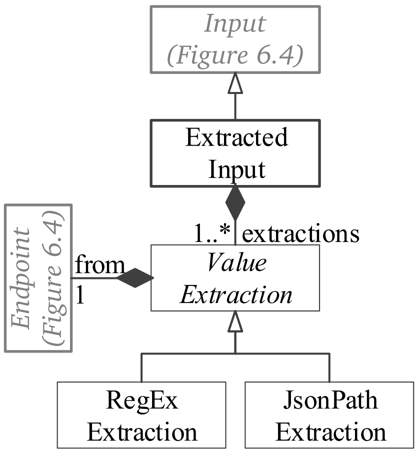


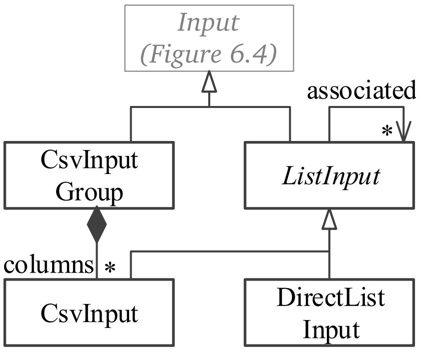
Figure 6.6.: `ExtractedInput` implementation for execution-dependent parameterization (based on H. Schulz et al., 2020c).

ValueExtraction: A `ValueExtraction` refers to an `Endpoint` (*from*) for extracting a value from its requests' responses. Furthermore, it defines a string *response key* defining the part of the response that the extraction targets. The *response key* can either be the status line, the headers, or the response body. For handling cases in which no value can be extracted, a user can define a *fallback value*. Finally, a *match number* states which value to use when the extraction matches several times (0 denotes selecting a value randomly). `ValueExtraction` is an abstract type and is implemented by `RegExExtraction` and `JsonPathExtraction`.

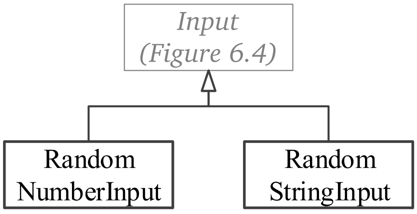
RegExExtraction: A `RegExExtraction` is an implementation of the `ValueExtraction` containing a *regular expression* that defines how to extract the value and a *template* that states how to assemble the extracted value. Using a *template (1)*, reference to the appropriate `Endpoint`, response body as *response key*, and a *match number* 1, the following exemplary *regular expression* extracts the CSRF token from the endpoint's response: *<input name="csrfToken" type="hidden" value="(.*)"/>*

JsonPathExtraction: For extracting values from JSON responses, we provide the `JsonPathExtractions`. It holds a *json path* (Goessner, 2007), which identifies an entity in a JSON object tree. For extracting the item ID from the JSON response in the taxonomy's example, we can use a *json path* such as *$.item.id* for identifying the field *id* in the object with the key *item*. The other attributes are similar to the `RegExExtraction` before.

ListInput: `ListInput` is a commonly-used type of input data specification, shown in Figure 6.7a. It represents the list-based input data of the taxonomy. The parameter value for a specific request is randomly selected from a *list of values*. Multiple `ListInputs` can be *associated*, meaning that a request should use the values from the same list index. For instance, `ListInputs` can state the user names and passwords of the test users. In this case, the virtual users have to ensure to take corresponding parameter values for the login request. The `ListInput` has two implementations `DirectListInput` and `CsvInput`, specifying the value lists differently.



(a) `ListInput` implementations (based on H. Schulz et al., 2020c).



(b) `RandomNumberInput` and `Random-StringInput` implementations.

Figure 6.7.: `Input` implementations for random execution-independent parameterization.

DirectListInput: The `DirectListInput` is a `ListInput` implementation that defines the value list directly in the annotation model. It is best suited for small lists because the annotation would become very long otherwise. For specifying the exemplary list of product colors, we can use this implementation, because the colors are limited to black, red, and silver.

CsvInput: The `CsvInput` is a `ListInput` that is better suited for large value lists. Instead of storing the values directly in the annotation model, it refers to an additional CSV file. Hence, the annotation model is kept short. The `CsvInput` refers to the file via a *file name* and the specific column via the *column index*. Furthermore, the *separator* of the CSV file entries can be specified. The default value is the semicolon (`;`). Finally, users can specify whether the file has a *header*.

CsvInputGroup.[†] For specifying CSV files without redundancy, we provide the `CsvInputGroup` in addition to the `CsvInput`. It allows defining inputs of multiple columns of the same CSV file. For that, the `CsvInputGroup` has the properties *file name*, *separator*, and *header*. In addition, it holds multiple `CsvInputs` as *columns*, which inherit these properties. The *column index* is determined based on the order of specified *columns*. Hence, there is no additional information required per column.

RandomNumberInput.[†] For the taxonomy category of randomly generated input data, we provide two `Input` implementations (Figure 6.7b). The first implementation is the `RandomNumberInput`, which generates a random integer number based on an *upper* and *lower limit*. The limits are either defined by a static value or a dynamically retrieved value from another `Input`.

RandomStringInput.[†] The second randomly generated input data type is the `RandomStringInput`. It generates a random string based on a *template*,

[†]We first envisioned this `Input` in previous work (H. Schulz et al., 2020c) and introduce it anew in this thesis.

which depicts the structure of the generated string. The template utilizes regular expression terminology so that the generated strings match the template. In the example of a randomly generated UUID, we used the template *[0-9A-D]{8}\-[0-9A-D]{4}\-[0-9A-D]{4}\-[0-9A-D]{4}\-[0-9A-D]{12}*.

CounterInput: The `CounterInput` (Figure 6.8) is another alternative for specifying large input data lists easily. It falls into the category of incremental input data of the taxonomy (see Section 6.1). It defines a *start* and *maximum* value as well as an *increment*. Starting from *start*, a counter value is incremented by the *increment* each time a value is retrieved. If the counter reaches the *maximum*, it restarts at the *start* value. A *format* defines a string that is built out of the counter value. The counter can be applied to different *scopes*, which can be either *global*, *user*, or *user-iteration*. The *global* scope means that there is only one instance of the counter, which all virtual users share. With the *user* scope, each virtual user has its instance and increments it independently. In the *user-iteration* scope, each virtual user has its counter instance, too, and resets it after each iteration. As an example, a list of 100 user names can be modeled by a `CounterInput` with a *start* value of 1, *maximum* of 100, *increment* of 1, the *format user-#@test.com* for different e-mail addresses (# is the placeholder for the counter value), and the *global scope* for increasing it each time the login request retrieves a value.

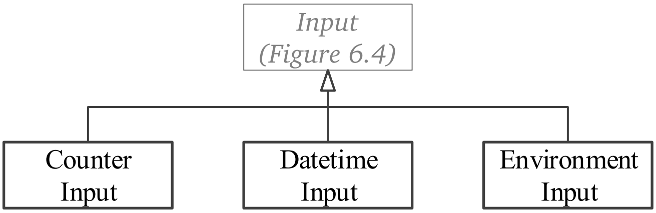


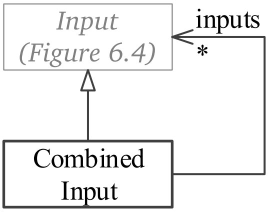
Figure 6.8.: `Input` implementations for sytematic execution-independent parameterization.

DatetimeInput.† For the category of environment-dependent input data, we provide the `DatetimeInput`, which generates a value based on the current date and time. It has a *format* and an *offset*. The *format* is based on the SimpleDateFormat of the Java Platform SE 8 (Oracle, 2020) and defines the date and time format. The *offset* states a duration to be added to the current time based on the duration format *PnDTnHnMn* (ISO 8601). For instance, we can retrieve the current date plus two days in a form such as *2019/12/14* by using the *format yyyy/MM/dd* and the *offset P2D*.

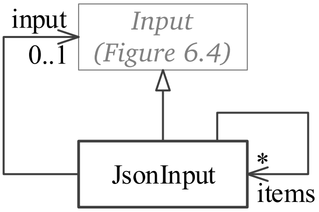
EnvironmentInput.† As the second type of environment-dependent input data, we introduce the `EnvironmentInput`. It reads a value from the environment of the load test, e.g., an environment variable. For that, it defines a *property*, which fills the value of the input initially at test startup. This type of input specification is especially useful if a property is only known at test startup time or cannot be defined in plaintext due to confidentiality or security reasons, e.g., a password.

CombinedInput.† The last considered category of the taxonomy is combined input data. For this category, we introduce three different `Input` implementations. The first implementation is the `CombinedInput` (Figure 6.9a), which merges several other `Inputs` into one string. For this, it holds a list of `Inputs` and a *format* defining how to combine the inputs. As an example, for combining two random numbers, we can use the *format (1)-(2)* and refer to two `RandomNumberInputs`, which result in strings such as *42-73*.

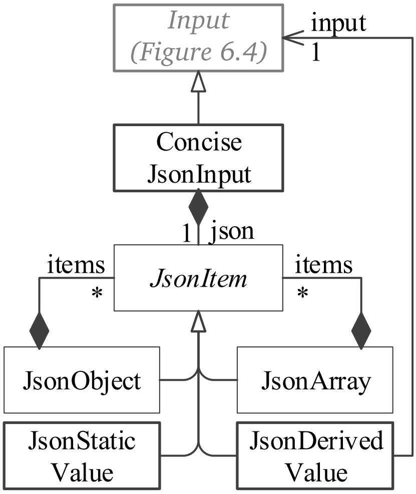
JsonInput: Angerstein (2018) introduces the second combined input data type, which is `JsonInput`. It is shown in Figure 6.9b. It allows combining several Inputs into a JSON object tree. Even though there is a new, more concise implementation, we provide both implementations for downward compatibility. A `JsonInput` holds a *type*, *name*, list of other `JsonInputs` (*items*), and refers to another `Input`. Except for the *type*, all attributes are optional. The *type* defines the semantics of the input and can be one out of *object*, *array*, *string*, and *number*. In the case of an *object*, the *items* represent



(a) `CombinedInput` implementation.



(b) First `JsonInput` implementation (based on Angerstein, 2018).



(c) Improved `ConciseJsonInput` implementation.

Figure 6.9.: `Input` implementations for combined parameterization.

```
{
    "profile": {
        "name": # retrieved from another input
        "country": "Germany"
    }
}
```

Listing 6.1: Exemplary JSON object tree.

child nodes in the JSON tree. The respective *names* are used as the keys of each node. In the case of an *array*, the *items* represent the elements of the array. In the case of a *string* or *number*, the *input* defines the effective value.

ConciseJsonInput.† Because the `JsonInput` turned out to be cumbersome (see Section 12.4), we introduce a more concise implementation. The `ConciseJsonInput` (Figure 6.9c) allows JSON trees to be defined while maintaining the trees' structure. For this, it contains a single `JsonItem` (*json*), which can either be a `JsonObject`, `JsonArray`, `JsonStaticValue`, or `JsonDerivedValue`. The `JsonObject` represents an object and has an *items* reference to several other `JsonItems`. Each element in *items* additionally has a key, which defines the key in the generated JSON object. The `JsonArray` holds a list of other `ConciseJsonInputs`, which constitute the content of the array. The `JsonStaticValue` represents a static value defined in the field *value*. Finally, values can be retrieved from other `Inputs` using the `JsonDerivedValue`. The *input* reference states from which `Input` the value is to be retrieved. The main advantage of this new representation is the object structure, which is close to the structure of the finally generated JSON object tree. Hence, it is be more concise.

Comparing JsonInput and ConciseJsonInput: We compare the `JsonInput` and `ConciseJsonInput` in an example. Listing 6.1 shows an exemplary JSON object tree we want to use in an IDPA application. When we model the JSON tree, we need to respect the value of the field *name*, which we need to retrieve from another `Input`.

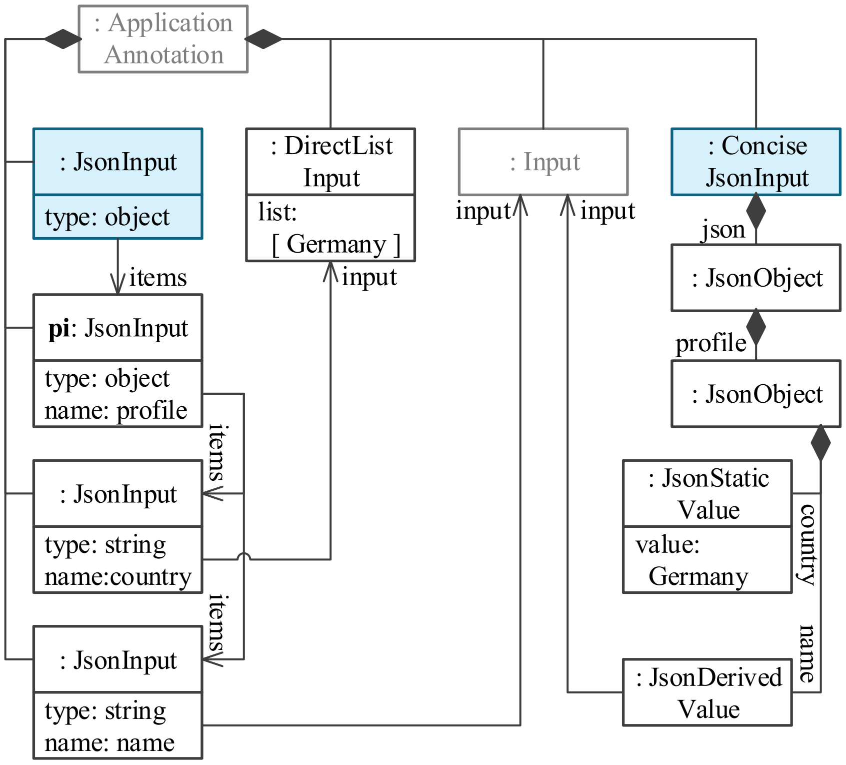


Figure 6.10.: Comparison of a `JsonInput` and a `ConciseJsonInput`. The root `Inputs` are highlighted.

As illustrated in Figure 6.10, for modeling the JSON tree as a `JsonInput`, we need to define one input per element of the tree. That is, there is one input with type *object* for the root object, one with type *object* for the *profile* object, one with type *string* for the *name* field, and one with type *string* for the *country* field. Furthermore, we need to define a `DirectListInput` containing the value of the country field. Then, each `JsonInput` refers to the nested inputs, and each nested input optionally defines its key via the *name* attribute. The input for the name field refers to the `Input` delivering the name value. This structure has the advantage that nested `JsonInputs` can be used individually. For instance, the `JsonInput` constituting the profile (*pi* in the figure) can be reused for another `Endpoint`, if only the profile is required as input. However, the `JsonInput` decomposes the original structure of the JSON tree. Hence, the structure is more extensive, and readability is decreased.

For this reason, we introduced the `ConciseJsonInput`, which maintains the original JSON tree structure. In the example, the `ConciseJsonInput` is composed of a `JsonObject`, which represents the root object. In turn, this `JsonObject` contains another one, which represents the profile object. Finally, a `JsonStaticValue` and a `JsonDerivedValue` define the values of the country and name fields. Hence, the original JSON structure is maintained. However, this representation does not allow reusing parts of the whole JSON object.

6.3.3. YAML Serialization

As described in Section 6.3.1, we designed the IDPA for the use in the context of CSE, and therefore, we use the commonly-used YAML format for serialization. In the following, we provide an example of a serialized IDPA and present the precise JSON schemata (Wright, 2019).

6.3.3.1. A YAML Example

Listings 6.2 and 6.3 provide examples of an application model and an annotation. The examples are excerpts based on the models we used for the evaluation with the Heat Clinic (Section 12.3), which is a webshop for hot sauces. We consider two HTTP endpoints. The *hotSaucesDetails* endpoint returns an HTML page, which presents the details of a specific sauce. The type of the endpoint — *http* — is defined using a `!<·>` YAML tag. The ID of the endpoint is defined using the built-in `&` operator. The properties of the endpoint are defined using key-value pairs with values of the respective type, e.g., a *string* for the domain and an *array* for the parameters. Several properties, such as the lists of headers and parameters, are optional and can be left out, meaning that the endpoint has no defined headers or parameters.

```yaml
---
&heat-clinic
timestamp: 2018-03-07T17-57-00-000Z
endpoints:
- !<http>
  &hotSaucesDetails
  domain: 172.16.145.67
  port: 8080
  path: /hot-sauces/{sauce}
  method: GET
  parameters:
  - &hotSaucesDetails_sauce
    name: sauce
    parameter-type: url-part
  protocol: http
- !<http>
  &addToCart
  domain: 172.16.145.67
  port: 8080
  path: /cart/add
  method: POST
  headers:
  - 'Content-Type: application/x-www-form-urlencoded'
  - 'X-Requested-With: XMLHttpRequest'
  parameters:
  - &addToCart_csrfToken
    name: csrfToken
    parameter-type: form
  - &addToCart_quantity
    name: quantity
    parameter-type: form
  - &addToCart_productId
    name: productId
    parameter-type: form
  protocol: http
```
Listing 6.2: YAML serialization of an examplary IDPA application model.

```yaml
---
overrides:
- HttpEndpoint.domain: localhost
- HttpEndpoint.port: 8080
inputs:
- !<csv>
  &Input_sauce_name
  file: hot-sauces.csv
  column: 1
  separator: ,
- !<extracted>
  &Input_csrfToken
  extractions:
  - from: hotSaucesDetails
    pattern: <input name="csrfToken" type="hidden"
        value="(.*)"/>
- !<direct>
  &Input_quantity
  data: [ 1, 2, 3, 4, 5 ]
endpoint-annotations:
- endpoint: hotSaucesDetails
  overrides:
  - HttpParameter.encoded: true
  parameter-annotations:
  - parameter: hotSaucesDetails_sauce
    input: *Input_sauce_name
- endpoint: addToCart
  parameter-annotations:
  - parameter: addToCart_csrfToken
    input: *Input_csrfToken
  - parameter: addToCart_quantity
    input: *Input_quantity
  - parameter: addToCart_productId
    input: *...
```

Listing 6.3: YAML serialization of an examplary IDPA annotation.

This endpoint has one parameter, which specifies the sauce to be visited. The parameter is encoded as part of the URL (*sauce*). It is specified using similar concepts as for the endpoint but as a nested attribute.

The *addToCart* endpoint adds a specific sauce to the shopping cart. This endpoint has three parameters, which are all encoded as form parameters. First, the already mentioned CSRF token has to be specified. Second, a quantity of sauces to be added to the cart has to be defined. Last, the endpoint requires the ID of the sauce. By first visiting the product details and then buying it, end-users typically call the *hotSaucesDetails* first before calling *addToCart*.

The annotation parameterizes a load test that is generated for the Heat Clinic. First, we define static parameterizations as `Overrides` in the *overrides* list. We override the host and port of all endpoints using the *HttpEndpoint.domain* and *.port* keys. Second, we specify `Inputs`, which we use to define the parameter values of *hotSaucesDetails* and *addToCart*. Similar to `Endpoints`, the type of each `Input` is defined by a YAML tag. We define a `CsvInput` holding all possible sauce names (*Input_sauce_name*), an `ExtractedInput` using a `RegExExtraction` for retrieving the *CSRF* token from the *hotSaucesDetails* response and using it for *addToCart* (*Input_csrfToken*), and a *DirectListInput*, which defines several valid quantities that can be passed to *addToCart* (*Input_quantity*). Finally, we define the `EndpointAnnotations` and `ParameterAnnotations`, which map the `Inputs` to the endpoints and parameters. For the sauce parameter of the *hotSaucesDetails* endpoint, we use the *Input_sauce_name*. Furthermore, we define this parameter to be encoded with the *HttpParameter.encode* override. Note that this `Override` only applies to this parameter, while the top-level `Overrides` apply to all endpoints in the scope of the whole annotation model. For the parameters of the *addToCart* endpoint, we use the *Input_extracted_csrfToken* as CSRF token and the *Input_quantity* as quantity. For the sake of space, we omit the `Input` for the product ID parameter, which we would extract from the *hostSaucesDetails* request, similar to the CSRF token.

```yaml
title: Application Model
type: object
properties:
  id:
    type: string
  timestamp:
    type: string
  endpoints:
    type: array
    items:
      oneOf:
      - $ref: '#/definitions/HttpEndpoint'
```

Listing 6.4: Application schema excerpt.

6.3.3.2. YAML Schemata

As a formal definition of the YAML serialization, we define JSON schemata (Wright, 2019), which are based on the JSON format as well. Because the YAML and JSON formats are related, JSON schemata can also describe YAML entities. To stick to the format and for better readability, we represent the schemata in the YAML format. The complete schemata can be found in Appendix A. Here, we present a small excerpt for illustrating the concept.

Listing 6.4 shows an excerpt from the `Application` schema. Each entity of the IDPA is represented by such a definition. In this case, the *title* defines the title of the whole schema. For other entities such as the `HttpEndpoint`, a respective name is defined so that other entities can refer to it. The *type* defines the general type of the entity. In most cases, it is an object, but for the `JsonItem` implementations, other types are used also. For instance, the `JsonArray` is of type *array* (see Listing A.23). In the *properties* section, we define the attributes of the elements. For the `Application`, we define the ID, the timestamp, and the list of endpoints. Each of the properties is defined using the same concepts as for the root entity. For the list of endpoints, we make use of the *oneOf* notation, which states that one or more specific types can be used for the respective property. In this case, we only refer to the `HttpEndpoint` schema. However, if we added new `Endpoint` implementations such as an `AmqpEndpoint`, we would add them at this point.

6.4. Evolving Input Data and Properties Annotations

The core feature of the IDPA is its reusability for multiple generated load tests. The entailed parameterizations are automatically evolved over changes in the production system's workload and, thus, over updated workload models and load tests. However, the tested application can also change in its API. Therefore, we need to support evolving IDPAs over API changes.

To address this challenge, we analyze API change types collected in the literature and develop feedback-based strategies for adjusting an IDPA to such changes. The identified API change types are additions and removals of endpoints and parameters, changes of endpoint and parameter properties, changes of the input data, and changes of the response behavior of an endpoint. While we can handle removals and property changes of endpoints and parameters automatically, the other change types require an expert's feedback. However, as described in Section 6.2, the feedback can be processed offline from the test generation and execution.

In the following, we describe the intended IDPA evolution process (Section 6.4.1), the collected API changes types (Section 6.4.2), and our proposed evolution strategies (Section 6.4.3).

The section is based on our already published work (H. Schulz et al., 2020c).

6.4.1. Evolution Process

Every time the API changes, the IDPA for the corresponding application can be impaired and needs to be adapted. Figure 6.11 illustrates the adaption process as a refinement of steps ④ and ⑤ of Figure 6.2 (Section 6.2). First, an expert such as an application developer introduces a delta to the API ①. As a consequence, the API specification changes. Using our automated transformation from OpenAPI (see Section 6.5.1), we transform the API specification to an updated version of the IDPA application model ②. Due to the ID tracing, the new application model is similar to the old one except for the introduced delta. Because of the changes to the application model, the annotation can be invalid ③. For instance, in the case of endpoint or parameter removals, there can be references to non-existing entities, e.g., from an `EndpointAnnotation` to a removed `Endpoint`. Furthermore, endpoint or parameter additions might require extending the annotation. Therefore, our approach collects all conflicts such as illegal references and possibly missing annotation elements and reports it as feedback to the expert ④. Based on the API delta and the conflicts, the expert can update the annotation ⑤. Finally, the expert commits the updated IDPA for generating load tests ⑥.

As described, our proposed process still entails manual effort for evolving IDPAs. However, all efforts can be applied offline from load test generation, e.g., at the time when the API is changed. Precisely, because an API delta is introduced in a local development environment by a code change, the IDPA can be updated immediately and before committing the code change to the central code repository.

The following sections present the considered API change types and describe the precise strategies applied for handling the individual change types.

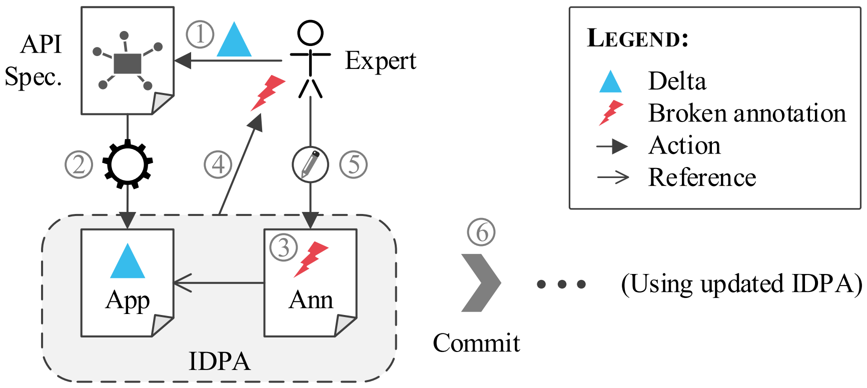


Figure 6.11.: Process of adapting an IDPA to API changes (based on H. Schulz et al., 2020c).

Table 6.1.: Proposed API Change Types (based on H. Schulz et al., 2020c)

Change Type	**Code**
Add Endpoint	AE
Remove Endpoint	RE
Change Endpoint [Property]	CE
Add Parameter	AP
Remove Parameter	RP
Change Parameter [Property]	CP
Change Input	CI
Change Response Behavior	CR

6.4.2. API Change Types

In a literature research, we identified four publications by Fokaefs et al. (2011), J. Li et al. (2013), Sohan et al. (2015), and S. Wang et al. (2014) providing relevant API change type classifications. In total, the authors examined the APIs of 25 different applications. Additionally, Wang et al. provide frequencies of each change type, which we can base on for generating a representative mix of API changes. From the collected change types, we derive our own classification, which is most suitable for IDPA evolution. In doing so, we address the research question RQ1.2: *Which API change types exist that affect load test parameterizations?* (see Section 5.1)

In the following, we provide our classification and explain the mapping to the change types from the literature.

6.4.2.1. Proposed Change Types

We summarize our proposed API change types in Table 6.1. We assign a code to each type for referencing them later. Each change type has a different impact on an IDPA. We discuss these impacts under the assumption that after the API changed, the IDPA application model is automatically updated, e.g., based on an API specification. Hence, the annotation needs to be adapted to the changed application model.

The first relevant change type is the addition of an `Endpoint`. For the annotation, the consequences are that no `EndpointAnnotation` covers the newly added `Endpoint`. Hence, in a generated load test, the new `Endpoint` might not be correctly parameterized. For removals of endpoints, the annotation can hold invalid references to an `Endpoint` that does not exist anymore. These invalid references affect both `EndpointAnnotations` and `ExtractedInputs`. Hence, an expert needs to resolve the invalid references. Note that we base on the unique IDs of the IDPA for tracing the evolution of a single entity. Hence, if an `Endpoint` is replaced by a new one that has a different ID, we classify this change as *Add Endpoint* plus *Remove Endpoint*. Another change type directly affecting an `Endpoint` is the change of an endpoint property such as the domain name, port number, or path. In general, all properties of an `Endpoint` can be affected. Because the endpoint properties are solely stored in the application model, the annotation does not have to be adjusted in this case.

The same set of change types described above also exists for `Parameters`, i.e., parameter additions, removals, and property changes. Here, the same conditions apply as for `Endpoints`. Exemplary properties that can change are the parameter name and type. Again, we rely on the unique IDs for tracing a `Parameter`. For instance, if the parameter name changes, but the ID remains the same, we consider it the same. However, if the ID changes but the name remains the same, we consider it to be two different `Parameters`.

Change Input is a change type subsuming all API changes that affect the input data specifications in an annotation. Such changes do not change the application model, and the IDPA itself remains valid. However, the target application might not respond as expected anymore. A typical example is a JSON body whose structure has been changed, e.g., by adding a new required attribute. If the corresponding `(Concise)JsonInput`, which defines the input data for this endpoint, is not adjusted, the required attribute will be missing and might cause an error. Hence, an expert needs to refine the defined `Inputs`.

The final change type of our classification is *Change Response Behavior*, which subsumes all changes to the responses of endpoints. Similar to *Change*

Input, changes of this type do not affect the application model. However, as the response value of a specific `Endpoint` has changed, `ExtractedInputs` that refer to this `Endpoint` might not work anymore. As an example, if the input uses a regular expression to extract the value, the expression might not match the new responses anymore. As a consequence, a generated load test can fail.

6.4.2.2. Mapping to Change Types from the Literature

Having defined the relevant API change types, we illustrate the mapping to change types presented in the literature. These change types are listed in Tables 6.2 to 6.5. Some presented change types are not mapped to our classification, because they do not impact an IDPA. Overall, the change types from the literature have been derived from 25 different applications.

Table 6.2.: Mapping of the Change Types to J. Li et al. (2013) (based on H. Schulz et al., 2020c)

Change Type	Mapping
Combine/Split Methods	AE, RE
Delete Method	RE
Unsupport Request Method	RE
Rename Method	CE[path]
Add or Remove Parameter	AP, RP
Rename Parameter	CP[name]
Change Format of Parameter	CP[encoding], CI
Change XML Tag	CI
Change Type of Parameter	CI
Change Upper Bound of Parameter	CI
Restrict Access to API	CI
Change Type/Format of Return Value	CR
Expose Data	CR
Change Default Value of Parameter	–

Mapping to J. Li et al.: In the list of J. Li et al. (Table 6.2), our *Add Endpoint* change type only exists as part of *Combine* and *Split Methods*. API changes of both types entail replacing several endpoints by one or replacing one endpoint by several ones. Even though there is a semantic relation of the replaced and new endpoints, the `Endpoints` in the IDPA will have new IDs and, thus, will be considered as separate `Endpoints`.

Remove Endpoint is part of *Combine* and *Split Methods*, *Delete Method*, and *Unsupport Request Method*. The latter change type is analogous to removing an `Endpoint` from an IDPA because it splits `Endpoints` according to the HTTP request method. Hence, if a specific request method is not supported anymore, the corresponding `Endpoint` will be removed.

The *Rename Method* change type refers to our *Change Endpoint [Path]* type. J. Li et al. refer to the endpoint path as the *method*, and hence, renaming the method means changing the path.

Add and *Remove Parameter* can be found as similarly named change types.

Furthermore, there are two change types relating to our *Change Parameter [Property]*. First, *Rename Parameter* means renaming the parameter name while — as presumed — the ID of the corresponding `Parameter` remains the same. However, this depends on the precise renaming mechanism. In the case that the ID has changed, this change type is reflected by subsequent *Remove Parameter* and *Add Parameter*. Second, changes of type *Change Format of Parameter* subsume changing the encoding. Therefore, it relates to our *Change Parameter [Encoding]*.

Change Format of Parameter also relates to our *Change Input*, because the input data specifications need to be adapted. Furthermore, the proposed change types of *Change XML Tag*, *Change Type of Parameter*, *Change Upper Bound of Parameter*, and *Restrict Access to API* all relate to our *Change Input* because they require adapting the input specifications. While the first two change types directly affect parts of the format of input values, the latter two ones have an impact as follows. *Change Upper Bound of Parameter* can require adjusting the input data specifications in the case that they use values higher than the new upper bound. *Restrict Access to API* can require using new authentication tokens and thus, changing the authentication input data.

Our *Change Response Behavior* type can be found as *Change Type* or *Format of Return Value*. Furthermore, *Expose Data* is a particular type of changing the format of a return value. Precisely, it flattens hierarchical data structures. As a consequence, `ValueExtractions` might become invalid and need to be adjusted.

Finally, the *Change Default Value of Parameter* change type cannot be mapped to any of our change types. However, we presume this change type to be irrelevant for the IDPA evolution. Changing a default value will not change the end user's behavior. Hence, as IDPAs are meant to be used for generating representative load tests, the virtual user's behavior should not be changed, either.

Mapping to Sohan et al.: In the list of Sohan et al. (Table 6.3), our *Add Endpoint* type is part of *Move API Elements*, which entails removing endpoints and adding them at different locations. Such a moving of `Endpoints` likely changes their IDs, and thus, our approach will consider the `Endpoints` to be new. Similarly, the change type subsumes *Remove Endpoint* and *Add* and *Remove Parameter*.

Our *Change Endpoint [Property]* maps to the *Rename API Elements* and *HTTP Header Change* types. Changes of the first type affect the path of an endpoint, while changes of the second type affect the header. Furthermore, *Rename API Elements* also subsumes renaming a parameter, which relates to our *Change Parameter [Name]*.

The authors define the *Behavior Change* type similar to our *Change Response Behavior*.

Finally, *Post* and *Error Condition Change* are not reflected in our change type classification, because they only affect the user behavior after the request and the error definition. Hence, the IDPA does not need to be adjusted.

Mapping to Fokaefs et al.: Fokaefs et al. propose change types that affect the type of parameter or return values (Table 6.4). Because the authors do not explicitly distinguish between parameter and return types, mostly all change types are to be classified as *Change Input* and *Change Response*

Behavior. In detail, *Aggressive Evolution* refers to a massive change of types, i.e., several parameter values and responses change. *Renaming Variables* means renaming the attributes of types. Hence, input specifications can become invalid — e.g., if JSON attributes are renamed — and `ValueExtractions` can break, e.g., because of invalid JSON paths. *Adding New Types* refers to adding new attributes to existing types, which requires adaption of the IDPA if the new attributes are required. Also, `RegExExtractions` can be affected because the responses might be formatted differently, e.g., by inlining the new attributes in the response string. *Changing Input or Output Types* subsumes all changes to the type of the parameter or return values.

The final change type of *Inline Type* does not affect the IDPA, because it only describes changing the underlying type structure, but does not affect the interface to the user. Hence, a load test is not affected, either.

Table 6.3.: Mapping of the Change Types to Sohan et al. (2015) (based on H. Schulz et al., 2020c)

Change Type	**Mapping**
Move API Elements	AE, RE, AP, RP
Rename API Elements	CE[path], CP[name]
HTTP Header Change	CE[header]
Behavior Change	CR
Post/Error Condition Change	–

Table 6.4.: Mapping of the Change Types to Fokaefs et al. (2011) (based on H. Schulz et al., 2020c)

Change Type	**Mapping**
Aggressive Evolution	CI, CR
Renaming Variables	CI, CR
Adding New Types	CI, CR
Changing Input or Output Types	CI, CR
Inline Type	–

Table 6.5.: Mapping of the Change Types to S. Wang et al. (2014) (based on H. Schulz et al., 2020c)

Level	Change Type	Frequency	Mapping
API	Change Resource URL		CE[domain], CE[base-path]
	Change Authentication Model		CI
	Change Response Format		CR
	Delete Response Format		CR
	Add Response Format		–
	Add Authentication Model		–
	Change Rate Limit		–
Method	Add Method	41.52%	AE
	Delete Method	15.65%	RE
	Change Method Name	11.52 %	CE[path]
	Change Domain URL	0.43%	CE[domain]
	Change Authentication Model	2.18%	CI
	Change Response Format	1.52%	CR
	Add Error Code	0.43%	CR
	Change Rate Limit	1.52%	–
Parameter	Add Parameter	11.09%	AP
	Delete Parameter	4.57%	RP
	Change	6.96%	CP[name]
	Change Format or Type	2.83%	CI
	Change Require Type	0.87%	CI
	Change Rate Limit	0.22%	–

Mapping to S. Wang et al.: S. Wang et al. distinguish between API-, method- (endpoint), and parameter-level changes. Furthermore, they provide relative frequencies of the method- and parameter-level change types.

The API-level change types relate to ours as follows. *Change Resource URL* refers to changes to the domain or global base path. Hence, our corresponding change types are *Change Endpoint [Domain]* and *[Base Path]*. *Change Authentication Model* can require using new authentication tokens and, thus, adjusting the input data specifications. *Change* and *Delete Response Format* impact the response of an endpoint and, thus, relate to *Change Response Behavior*. *Add Response Format* and *Add Authentication Model* are only additions of new concepts while the old concepts are still available. Hence, the IDPA is still valid and does not need to be adjusted. Finally, *Change Rate Limit* does not relate to load test parameterization but rather to the user behavior. Thus, there is no corresponding change type in our classification.

The method-level change types are partially identical to ours. Our *Add* and *Remove Endpoint* are similar to *Add* and *Delete Method*. Furthermore, *Change Endpoint [Property]* can be found as *Change Method Name* and *Change Domain URL*. Similar to the API level, *Change Authentication Model* relates to *Change Input*. *Change Response Format* and *Add Error Code* change responses and, thus, relate to our *Change Response Behavior*. Finally, as before, we do not classify *Change Rate Limit*.

The parameter-level change types also match our classification well. Analogous change types to our *Add* and *Remove Parameter* are *Add* and *Delete Parameter*. The *Change* type relates to *Change Parameter [Name]*. Furthermore, our *Change Input* is part of *Change Format or Type* and *Change Require Type*. Changes of the former type directly affect the input data for a specific parameter. Changes of the latter type can require adapting the IDPA because formerly not required parameters were left out and now need to be specified. Again, there is no mapping for *Change Rate Limit*.

6.4.3. Evolution Strategies

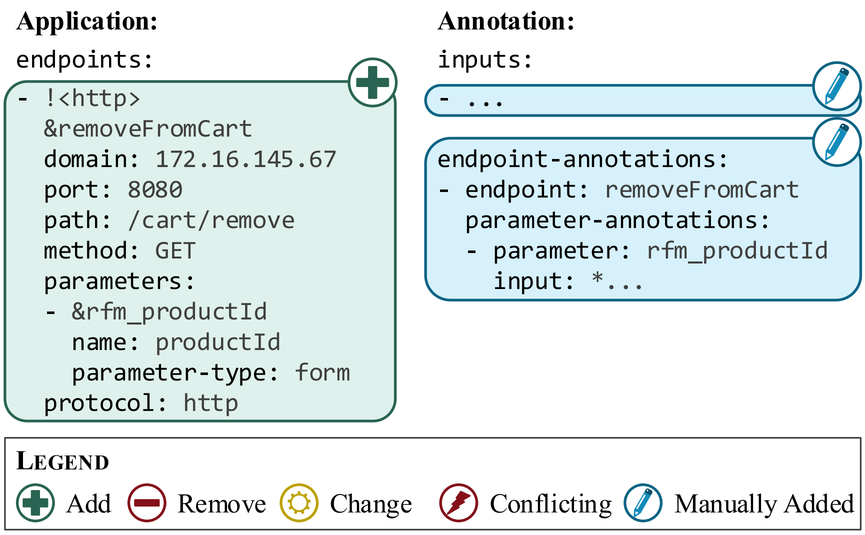
Knowing the possible API change types that can occur, we develop feedback-based strategies for handling changes of each type. As illustrated in Figure 6.11 (Section 6.4.1), all strategies have in common that first, the IDPA application model is updated, e.g., by transforming it from an updated API specification. Then, there are four subsequent steps to update possibly conflicting annotations:

1. *Detection step:* All changes and possible conflicts are detected.
2. *Resolution step:* As many conflicts as possible are resolved automatically.

3. *Feedback step:* The expert responsible for the API change is informed about possible conflicts.
4. *Update step:* The expert updates the annotation model.

In the following, we describe the four steps for each change type.

Add Endpoint: When a new `Endpoint` is added to the application model, the IDPA is still valid, because the existing annotations are not affected. However, in a generated load test, the requests to the new endpoint can fail because of missing parameterizations. We illustrate such missing parameterizations in Figure 6.12. We consider the Heat Clinic we already referred to in Section 6.3.3.1. We consider a new `Endpoint` with ID *removeFromCart*, which was added to the application model. Requests to this endpoint will remove a specific product from the shopping cart. For that, the `Endpoint` has a `Parameter` *productId*, which specifies the product to be removed.



Application:

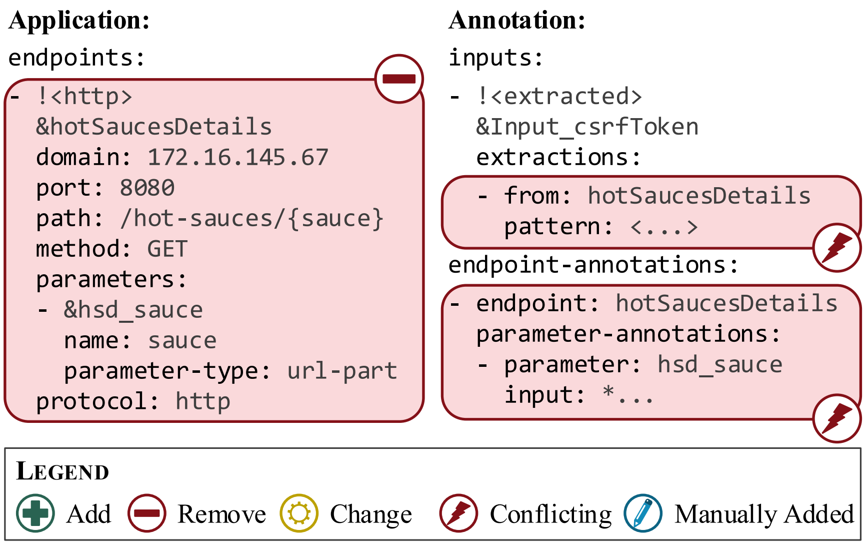
```
endpoints:
- !<http>
  &removeFromCart
  domain: 172.16.145.67
  port: 8080
  path: /cart/remove
  method: GET
  parameters:
  - &rfm_productId
    name: productId
    parameter-type: form
  protocol: http
```

Annotation:

```
inputs:
- ...
```

```
endpoint-annotations:
- endpoint: removeFromCart
  parameter-annotations:
  - parameter: rfm_productId
    input: *...
```

Figure 6.12.: Exemplary *Add Endpoint* change.



```
endpoints:
- !<http>
  &hotSaucesDetails
  domain: 172.16.145.67
  port: 8080
  path: /hot-sauces/{sauce}
  method: GET
  parameters:
  - &hsd_sauce
    name: sauce
    parameter-type: url-part
  protocol: http
```

```
inputs:
- !<extracted>
  &Input_csrfToken
  extractions:
  - from: hotSaucesDetails
    pattern: <...>
endpoint-annotations:
- endpoint: hotSaucesDetails
  parameter-annotations:
  - parameter: hsd_sauce
    input: *...
```

Figure 6.13.: Exemplary *Remove Endpoint* change.

In the detection step, we automatically detect `Endpoint` additions by comparing the application model version before and after the update, based on the `Endpoints`' IDs. The resolution step is omitted for endpoint additions. In the feedback step, we report the addition of the `Endpoint` to the expert. In this example, we report the addition of the *removeFromCart* `Endpoint`.

In the update step, the expert is responsible for parameterizing the new `Endpoint`. As an example, they could add a new `Input` for the *productId* parameter and add corresponding `EndpointAnnotations` and `ParameterAnnotations`, which map the `Input` to the `Endpoint` and `Parameter`.

Remove Endpoint: Removing an `Endpoint` from the application model can cause illegal references in the annotation. We illustrate this in Figure 6.13. We consider the *hotSaucesDetails* `Endpoint` of the Heat Clinic to be removed. As a consequence, the references of the corresponding `EndpointAnnotation` and `ParameterAnnotation` are invalid, because the *hotSaucesDetails* endpoint and *hsd_sauce* parameter are not existing anymore. Furthermore, the `RegExExtraction` of the *Input_csrfToken* is broken due to the same reason.

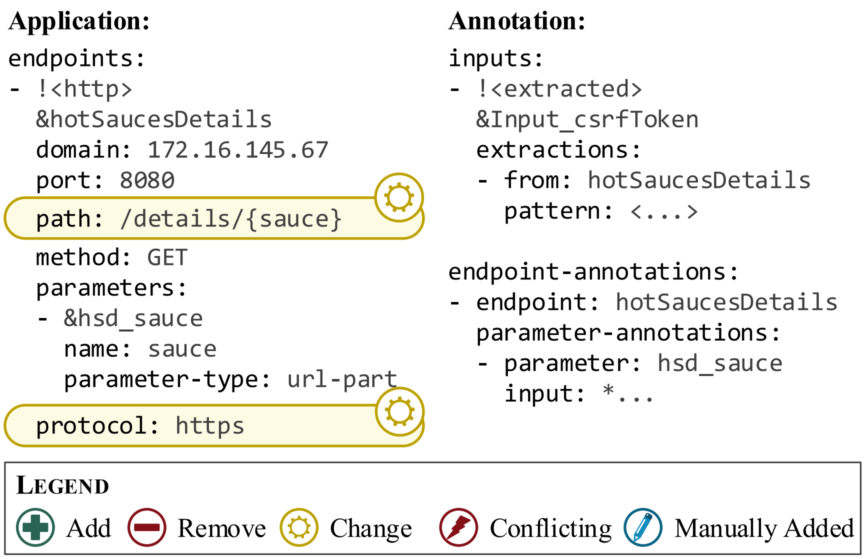
Similar to `Endpoint` additions, we detect the removals based on the IDs in the detection step. Furthermore, we identify all invalid references to the removed `Endpoints` and `Parameters`. In the resolution step, we remove all `EndpointAnnotations` and `ParameterAnnotations` holding such invalid references automatically. In the example, we remove the `EndpointAnnotation` for the *hotSaucesDetails* `Endpoint` and the `ParameterAnnotation` for the *hsd_sauce*. Furthermore, we remove all `ValueExtractions` holding invalid references. However, if the corresponding `ExtractedInputs` do not hold any `ValueExtractions` after the removals, no values will be extracted, and consequently, the requests using the `ExtractedInput` can fail. This is the case for the *Input_csrfToken*. In such a case, the expert has to add new `ValueExtractions` in the update step.

In the feedback step, we report all `Endpoint` removals to the expert. Also, we report all removals in the annotation, namely the `EndpointAnnotations`, `ParameterAnnotations`, and `ValueExtractions`. We also report if an `ExtractedInput` is now empty.

In the update step, the expert has to review all removals and possibly resolve empty `ExtractedInputs`. In the example, they have to add a new `ValueExtraction` to the *Input_csrfToken* or change it to another `Input` type.

Change Endpoint [Property]: The automated updating of the application model already covers property changes of `Endpoints`. This is illustrated in Figure 6.14. The path and protocol of the *hostSaucesDetails* `Endpoint` are changed to */details/sauce* and to *https*. However, all annotation elements only refer to the `Endpoint` via its ID *hotSaucesDetails*. Hence, they are not affected by the changes.

In the detection phase, we identify all property changes by comparing the `Endpoint` attributes before and after the application model update. Because the annotation is not affected, there are no resolution and update steps. Still, we report all property changes in the feedback step.



Application:

```
endpoints:
- !<http>
  &hotSaucesDetails
  domain: 172.16.145.67
  port: 8080
  path: /details/{sauce}
  method: GET
  parameters:
  - &hsd_sauce
    name: sauce
    parameter-type: url-part
  protocol: https
```

Annotation:

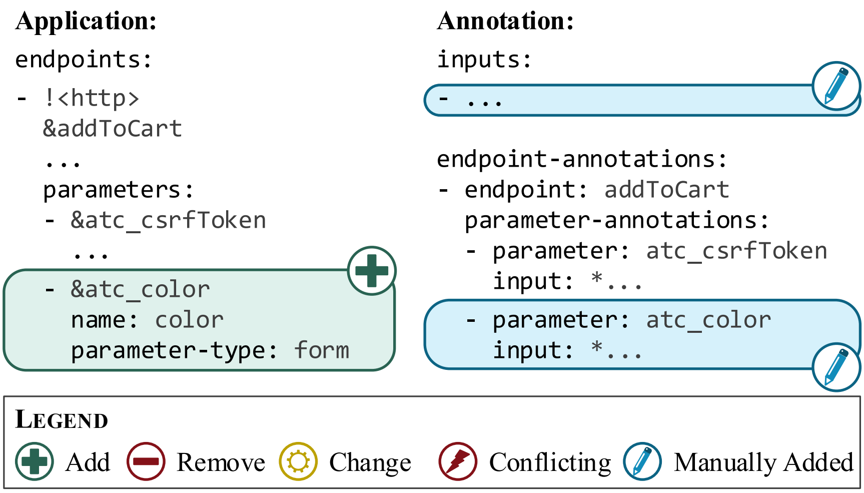
```
inputs:
- !<extracted>
  &Input_csrfToken
  extractions:
  - from: hotSaucesDetails
    pattern: <...>

endpoint-annotations:
- endpoint: hotSaucesDetails
  parameter-annotations:
  - parameter: hsd_sauce
    input: *...
```

LEGEND
Add Remove Change Conflicting Manually Added

Figure 6.14.: Exemplary *Change Endpoint [Path]* and *[Protocol]* changes.

Add Parameter: Parameter additions are treated similarly as endpoint additions, as illustrated in Figure 6.15. We consider the *addToCart* `Endpoint`, to which a new *color* parameter is added. Even though there are no invalid references in the annotation, we need to define the input data for the *color* parameter.

In the detection step, we identify the `Parameter` additions based on the IDs. The resolution step is skipped because we cannot resolve anything automatically. In the feedback step, we report the `Parameter` additions to the expert. Finally, the expert is responsible for parameterizing the new `Parameters` in the update step. In our example, they could add a new `Input` holding the allowed colors and a new `ParameterAnnotation` mapping the new `Input` to the *color* `Parameter`.

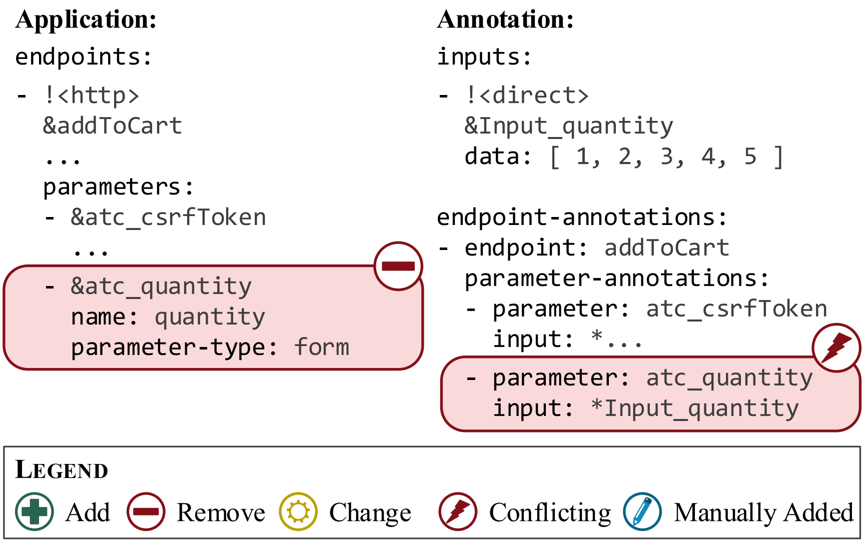

Application:

```
endpoints:
- !<http>
  &addToCart
  ...
  parameters:
  - &atc_csrfToken
    ...
  - &atc_color
    name: color
    parameter-type: form
```

Annotation:

```
inputs:
- ...

endpoint-annotations:
- endpoint: addToCart
  parameter-annotations:
  - parameter: atc_csrfToken
    input: *...
  - parameter: atc_color
    input: *...
```

Figure 6.15.: Exemplary *Add Parameter* change.


Application:

```
endpoints:
- !<http>
  &addToCart
  ...
  parameters:
  - &atc_csrfToken
    ...
  - &atc_quantity
    name: quantity
    parameter-type: form
```

Annotation:

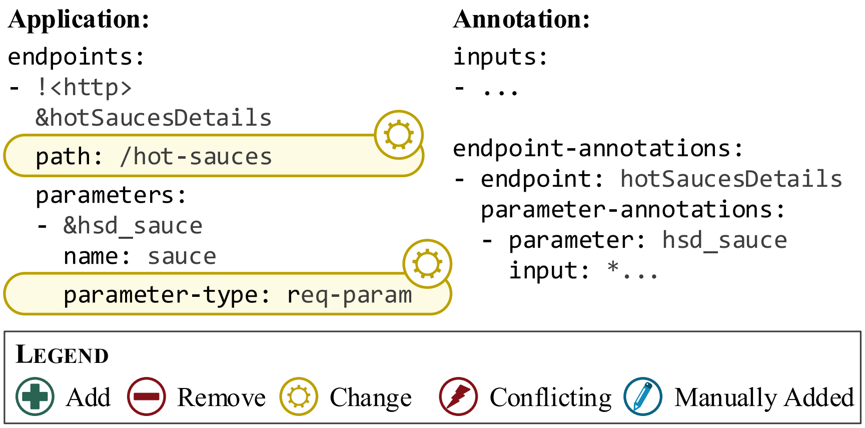
```
inputs:
- !<direct>
  &Input_quantity
  data: [ 1, 2, 3, 4, 5 ]

endpoint-annotations:
- endpoint: addToCart
  parameter-annotations:
  - parameter: atc_csrfToken
    input: *...
  - parameter: atc_quantity
    input: *Input_quantity
```

Figure 6.16.: Exemplary *Remove Parameter* change.

Remove Parameter: We can handle parameter removals fully automatically. This is illustrated in Figure 6.16. Again, we consider the *addToCart* `Endpoint` and the *quantity* `Parameter`, which is removed, e.g., because from now on, every product can only be bought once with one order. As a consequence, the reference of the corresponding `ParameterAnnotation` is invalid. However, we can remove the reference automatically as follows.

In the detection step, we identify the `Parameter` removals based on the IDs. In the resolution step, we remove all `ParameterAnnotations` that refer to the removed `Parameter`. Hence, we resolve all conflicts of the annotations. In the feedback step, we report the removals of the `Parameters` and `ParameterAnnotations` to the expert. The update step is skipped because we resolved all conflicts already in the resolution step. In the example, we remove the invalid reference by removing the `ParameterAnnotation` for the *quantity* `Parameter`.

Change Parameter [Property]: Similar to `Endpoints`, changes to `Parameter` properties do not affect the annotation and, thus, are automatically resolved. We illustrate this in Figure 6.17. We consider the *hotSaucesDetails* `Endpoint` and the *sauce* `Parameter`. The type of this parameter is changed



Application:

```
endpoints:
- !<http>
  &hotSaucesDetails
  path: /hot-sauces
  parameters:
  - &hsd_sauce
    name: sauce
    parameter-type: req-param
```

Annotation:

```
inputs:
- ...

endpoint-annotations:
- endpoint: hotSaucesDetails
  parameter-annotations:
  - parameter: hsd_sauce
    input: *...
```

Figure 6.17.: Exemplary *Change Parameter [Type]* change.

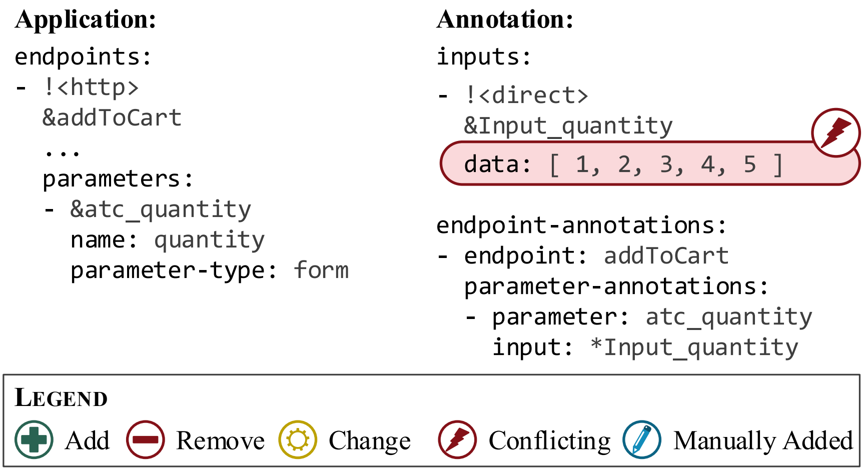
from *url-part* to *req-param*, i.e., it is now added as part of the query string: *?sauce=...*. At the same time, the sauce parameter is removed from the path of the `Endpoint`. We handle this `Endpoint` property change as described above. The `Parameter` property change is handled as follows.

In the detection step, we detect all property changes by comparing the `Parameter` attributes before and after the application model update. The resolution step is skipped. In the feedback step, we report the property changes to the expert. However, they do not have to take action because there are no conflicts in the annotation.

Change Input: Changes of type *Change Input* are more complex to resolve because the precise impact on the annotation is not apparent. We provide an example in Figure 6.18. We consider a change to the range of the allowed values of the *quantity* `Parameter` of the *addToCart* `Endpoint`. While before, there was no limit, the upper limit is now three. Hence, the *Input_quantity* is not a valid `Input` for the *quantity* `Parameter` and needs to be updated. However, the application model does not reflect the change to the quantity range. We process this change type as follows.

In the detection phase, we cannot detect anything because the application model is not changed. However, for future work, we suggest detecting input changes directly in API specifications such as OpenAPI (OpenAPI Initiative, 2020), because they can hold information about the allowed parameter values. Regardless of that, we presume the expert who changed the input format also updates the annotation. Hence, they know about the new requirements for the `Inputs`.

In the current state, the detection, resolution, and feedback steps are skipped, because no conflicts can be detected automatically. In the update step, we rely on the expert's knowledge about input changes to update the `Inputs` accordingly. In the example, the expert would need to remove all quantity values above three from the *Input_quantity*.



Application:

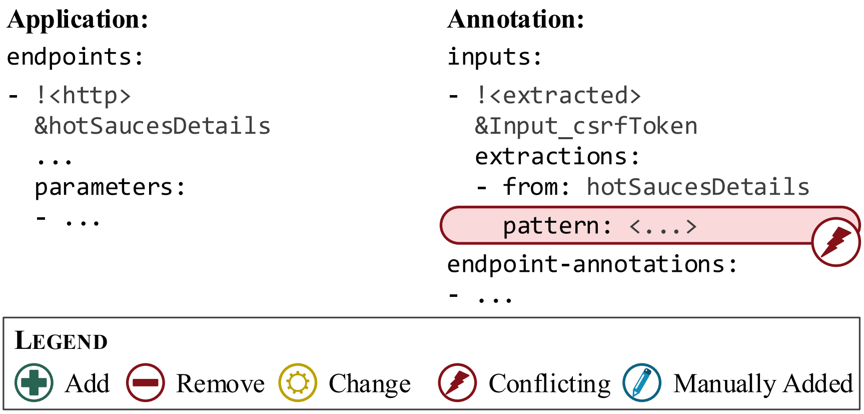
```
endpoints:
- !<http>
  &addToCart
  ...
  parameters:
  - &atc_quantity
    name: quantity
    parameter-type: form
```

Annotation:

```
inputs:
- !<direct>
  &Input_quantity
  data: [ 1, 2, 3, 4, 5 ]

endpoint-annotations:
- endpoint: addToCart
  parameter-annotations:
  - parameter: atc_quantity
    input: *Input_quantity
```

Legend

Figure 6.18.: Exemplary *Change Input* change.



Application:

```
endpoints:
- !<http>
  &hotSaucesDetails
  ...
  parameters:
  - ...
```

Annotation:

```
inputs:
- !<extracted>
  &Input_csrfToken
  extractions:
  - from: hotSaucesDetails
    pattern: <...>
endpoint-annotations:
- ...
```

Legend

Figure 6.19.: Exemplary *Change Response Behavior* change.

Change Response Behavior: The last considered change type is *Change Response Behavior*, which is as complex as *Change Input*. We illustrate this in Figure 6.19. We presume that the response format of the *hotSaucesDetails* `Endpoint` is changed. As a consequence, the pattern of the `RegExExtraction` of the *Input_csrfToken* does not match the responses anymore. Hence, the CSRF tokens are not extracted properly. However, similar to input changes, the application model does not reflect the response format. Therefore, we proceed as follows.

Again, the detection phase is skipped due to absent application model changes. However, directly detecting response behavior changes based on API specifications are possible future work as well. As an alternative, a dry run to check all `ValueExtractions` could be conducted in this step. In the current state, we again rely on the expert's knowledge about the changes.

Thus, the resolution and feedback steps are skipped, and the expert has to update the annotation solely based on their knowledge. In the example, they have to adjust the pattern of the `RegExExtraction` to the new response format.

6.5. Transforming Input Data and Properties Annotations

As described in Section 6.2, there are the following transformations including an IDPA: transformation from an API specification such as OpenAPI (OpenAPI Initiative, 2020) into an IDPA application model, transformation of request logs into a workload model based on an IDPA application model, and transformation of an IDPA and a workload model into an executable load test. In this section, we described these transformations. Section 6.5.1 describes the transformation of OpenAPI specifications into an IDPA application model. Section 6.5.2 describes the transformation of request logs into a workload model. Section 6.5.3 describes the transformation of an IDPA into a generic load test as well as to JMeter (Apache Software Foundation, 2020[a]) and BenchFlow (Ferme and Pautasso, 2018) load tests.

```
---
openapi: "3.0.2"
info:
  title: $t$
  version: $v$
servers:
- url: $h$://$d$:$q$/$p_{\text{base}}$
paths:
  $p_i$:
    $m_{i,j} \in \{\texttt{get}, \texttt{put}, \texttt{post}, \texttt{delete}, \texttt{options}, \texttt{head}, \texttt{patch}, \texttt{trace}\}$:
      operationId: $\phi_{i,j}$
      parameters:
      - name: $n_{i,j,k}$
        in: $\tau_{i,j,k} \in \{\texttt{query}, \texttt{header}, \texttt{path}\}$
        required: $r_{i,j,k} \in \{\texttt{true}, \texttt{false}\}$
        schema:
          type: The type of the parameter
      requestBody:
        content:
          $\mu_{i,j}$:
            schema:
              properties:
                $a_{i,j,s}$:
                  type: The type of the property
      responses:
        $c_{i,j,l}$:
          description: A textual description
```

Listing 6.5: Schema of an OpenAPI specification (based on SmartBear Software, 2020) to be transformed into an IDPA application model.

6.5.1. Transformation from OpenAPI Specifications

API specifications allow keeping track and using the endpoints of an application as a client comfortably. Therefore, it is recommended to use such a specification (Newman, 2015), especially in microservice environments, because others need to access the API. Because it comprises the endpoints with the respective parameters of an application (see Section 2.4), we can reuse an API specification to generate an IDPA application model. As the most prominent representative, we provide a transformation from OpenAPI specifications (OpenAPI Initiative, 2020).

Listing 6.5 provides the schema of an OpenAPI specification as expected by the transformation. It is a minimal example, not including any specifications that are not required for the transformation. For referring to the elements in the following, we label them with the following symbols:

- t: The title of the API.
- v: The version of the API.
- h: The protocol to be used with the requests. Can either be *http* or *https*.
- d: The domain name to be used.
- q: The port number to be used.
- p_{base}: The base path to be added before the path of each endpoint.
- p_i: The i-th path. There can be multiple endpoints with the same path, but different HTTP request methods.
- $m_{i,j}$: The j-th request method of the i-th path. Defines one endpoint.
- $\phi_{i,j}$: The unique ID of the endpoint represented by p_j and $m_{i,j}$.
- $n_{i,j,k}$: The name of the k-th parameter of $m_{i,j}$.
- $\tau_{i,j,k}$: The type of the parameter $n_{i,j,k}$. Currently, we only support $\tau_{i,j,k} \in \{\texttt{query}, \texttt{header}, \texttt{path}\}$ and not `cookie`.
- $r_{i,j,k}$: A Boolean flag indicating whether the parameter $n_{i,j,k}$ is required.
- $\mu_{i,j}$: The media type (IANA, 2020) of the request body. Only exists if the endpoint has a request body.
- $a_{i,j,s}$: The name of the s-th attribute of the request body. Please note that the schema focuses on the media types *multipart/form-data* and *application/x-www-form-urlencoded* and can differ for various media

types such as *application/json*. However, in this case, the precise schema is not relevant for the transformation.

- $c_{i,j,l}$: The l-th response code that can be returned by the endpoint represented by $m_{i,j}$.

Based on the OpenAPI schema, we describe the IDPA application model the transformation generates. For this, we utilize the following functions. Let S be the set of all strings. $\text{type} : S \to S$ maps an OpenAPI request type or media type x to the corresponding IDPA request type:

$$\text{type}(x) = \begin{cases} \texttt{'req-param'} & x = \texttt{'query'} \\ \texttt{'header'} & x = \texttt{'header'} \\ \texttt{'url-part'} & x = \texttt{'path'} \\ \texttt{'form'} & x \in \{\texttt{'multipart/form-data'}, \\ & \quad \texttt{'application/x-www-form-urlencoded'}\} \\ \texttt{'body'} & \text{else} \end{cases}$$

The function $\text{concat}_\sigma : S \times S \to S$ concatenates two strings with σ as a separator. As a shorthand for concatenating two strings x_1 and x_2 without separator, we use $\text{concat}(x_1, x_2)$. $\text{id} : S \to S$ transforms a string so that it can be used as an ID. Especially, it replaces all white space characters with `'_'`. Furthermore, it produces unique IDs by adding a suffix such as `'_2'`.

For one OpenAPI specification, the transformation will generate one IDPA `Application`. Listing 6.6 presents the schema of such an `Application`. The title t constitutes the ID of the `Application`, whereas we format t using the id function. For identifying the API version, we use the version attribute, which is set to v. Alternatively, we could use the timestamp of the transformation. However, we presume the version to be more accurate and required by OpenAPI in any case.

```
1  ---
   &id($t$)
3  version: $v$
   endpoints:
5  - !<http>
     &$\phi_{i,j}$
7    domain: $d$
     port: $q$
9    path: concat($p_{\texttt{base}}, p_i$)
     method: $m_{i,j}$
11   parameters:
     - &concat$_{\_}$($\phi_{i,j}, n_{i,j,k}$)
13     name: $n_{i,j,k}$
       parameter-type: type($\tau_{i,j,k}$)
15   - &concat$_{\_}$($\phi_{i,j}, a_{i,j,s}$)
       name: $a_{i,j,s}$
17     parameter-type: type($\mu_{i,j}$)
     - &concat$_{\_}$($\phi_{i,j}$, 'body')
19     parameter-type: body
     protocol: $h$
```

Listing 6.6: IDPA application model generated from an OpenAPI specification (see Listing 6.5).

For each request method $m_{i,j}$, the `Application` holds at least one corresponding `Endpoint`. If there are one or more request body media types $\mu_{i,j}$, there is one `Endpoint` per $\mu_{i,j}$. Otherwise, there is precisely one `Endpoint`. Because OpenAPI only allows describing REST APIs, every generated `Endpoint` is of type *http*. The domain name, port number, and protocol are extracted from the URL defined in the OpenAPI specification as d, q, and h. The path is constructed by concatenating the base path p_{base} and the path p_i. The request method $m_{i,j}$ can be reused as it is.

Each `Endpoint` has several `Parameters` as defined by the $n_{i,j,k}$ (line 12). Each $n_{i,j,k}$ both defines the parameter name and the ID of the `Parameter` in concatenation with the `Endpoint` ID. In doing so, we ensure that there are no ID conflicts due to similar parameters of different endpoints. The parameter type is transformed from the OpenAPI parameter type $\tau_{i,j,k}$ using the type function. For the OpenAPI parameter types *query*, *header*, and *path*, there are corresponding IDPA types *req-param*, *header*, and *url-part*. For the type *cookie*, there is currently no implementation in the IDPA.

Besides the OpenAPI parameters, the request body of the corresponding media type $\mu_{i,j}$ is transformed into `Parameters`, in case there is one. In the case of the *multipart/form-data* and *application/x-www-form-urlencoded* media types, we generate one `Parameter` per attribute $a_{i,j,s}$ (line 15). These parameters then have the type *form*, defined by type($\mu_{i,j}$). Furthermore, $a_{i,j,s}$ defines the name of the parameter. We transform all other media types into a single *body* `Parameter` (line 18). In this case, the ID is based on the `Endpoint`'s ID and the string `'body'`.

6.5.2. Transformation of Request Logs to Workload Models

The second transformation, in which the IDPA is involved, is the extraction of a workload model from recorded request logs. In this transformation, the IDPA is used to label individual requests according to the called endpoint. Then, the extraction algorithm — e.g., the WESSBAS-DSL extractor (Vögele et al., 2018, see Section 3.2.4) — uses the labels for grouping the requests to the same endpoints and for naming the elements of the extracted workload model. Hence, we can correlate IDPA `Endpoints` and workload model elements later on. In the extraction itself, the IDPA is not involved. In the following, we describe the labeling of the requests. Furthermore, we discuss how the labeling can be done in the case that the API has changed between recording the requests and extracting the workload model.

6.5.2.1. Without API Changes

Assuming there are no API changes, the IDPA is involved in the workload model extraction process as simple labeling of the individual requests. We illustrate this in Figure 6.20. Starting from the raw request logs, we use the application model of the IDPA to label each request with the ID of the corresponding `Endpoint`. In a second step, the workload model is extracted

Algorithm 6.1 Find the label for a request.

```
Let $d, q, p, m$ be the domain name, port number, path, and request method
    of one request
Let $E$ be the set of all Endpoints

function FindRequestLabel($d$, $q$, $p$, $m$, $E$)
    $\phi \leftarrow \emptyset$
    for all $e \in E$ do
        if $d = e$.domain & $q = e$.port & $m = e$.method
          & $p = e$.path then
            $\phi \leftarrow e$.id
        end if
    end for
    if $\phi = \emptyset$ then
        for all $e \in E$ do
            $r \leftarrow$ RegExReplace($e$.path, '\/', '\/')
            $r \leftarrow$ RegExReplace($r$, '\{.+\:(.+)\}', '$1')
            $r \leftarrow$ RegExReplace($r$, '\{.+\}', '.+')
            if $d = e$.domain & $q = e$.port & $m = e$.method
              & RegExMatches($r$, $p$) then
                $\phi \leftarrow e$.id
            end if
        end for
    end if
    return $\phi$
end function
```

from the labeled request logs. For this, we assume each request to hold at least the following information:

- domain name d
- port number q
- accessed path p
- HTTP request method m

Based on this information, Algorithm 6.1 defines the label for each request. First, all `Endpoints` are traversed and compared to d, q, p, and m. If all are equal, the algorithm found the correct `Endpoint` and stores its ID as the label. If it did not find an appropriate one, it checks each `Endpoint` again, but now considers URL variables such as {*sauce*} in */hot-sauces/*{*sauce*}. For this, it transforms the URL to a regular expression as follows. It adds an escape character \ to the slashes / and replaces each variable such as {*sauce*} with the expression `.+`, or the specified expression if present. The RegExReplace calls constitute these replacements. Then, we can check whether the path p matches the regular expression and store the `Endpoint`'s ID as a label if so. Non-labeled requests have no corresponding `Endpoint` and, therefore, will not be considered for the workload model extraction.

We illustrate the algorithm with an example. We consider the two `Endpoints` *hotSaucesDetails* and *addToCart* from Section 6.3.3.1 and a further `Endpoint` *hotSaucesOverview*. For the sake of simplicity, we assume all `Endpoints` and recorded requests to have the same domain name and port number. The request methods and paths of these `Endpoints` are:

- *hotSaucesDetails*: GET */hot-sauces/*{*sauce*}
- *hotSaucesOverview*: GET */hot-sauces/overview*
- *addToCart*: POST */cart/add*

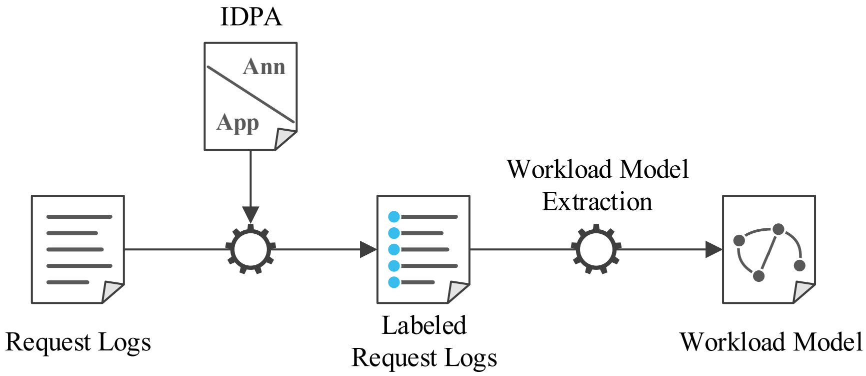


Figure 6.20.: Transformation process of request logs to a workload model using an IDPA.

Furthermore, we consider the following recorded requests:

1. GET */hot-sauces/overview*
2. GET */hot-sauces/sudden_death_sauce*
3. POST */cart/add*
4. POST */hot-sauces/add*

For the first and third requests, the `Endpoints` *hotSaucesOverview* and *addToCart* have the same request methods and path. Hence, the algorithm labels the requests with *hotSaucesOverview* and *addToCart*, respectively. The second request has a different path than all `Endpoints`. However, it matches the regular expression $\backslash/hot\text{-}sauces\backslash/.+$ generated from the path of *hot-SaucesDetails* and, therefore, is labeled accordingly. The last request matches this regular expression, as well. However, it has a different request method (POST) than the `Endpoint` (GET) and, therefore, is not labeled. This request will be ignored when extracting the workload model.

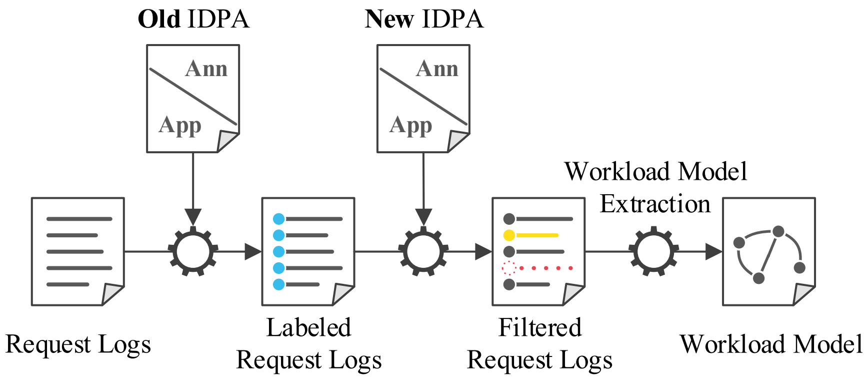


Figure 6.21.: Transformation process of request logs of an old API version to a workload model for a newer API version using IDPAs.

6.5.2.2. With API Changes

If the API of the target application changed, the IDPA changes as well (see Section 6.4). However, only request logs for the old version may be available, while we need to generate a workload model for the new version. Even though this can lead to less representative load tests, we provide extracting workload models based on old request logs as a workaround until new request logs are available. For this, we need to add an intermediate step to the transformation process, as illustrated in Figure 6.21. The transformation starts with the same step as without API changes, namely labeling the requests using the IDPA for the old API version. In a newly added next step, the labeled requests are then filtered and updated based on the introduced API changes of the new IDPA. Finally, the workload model extraction is applied to the filtered request logs.

The filtering adjusts the individual requests to the new API. For this, it performs the following actions depending on the particular API change type:

Add Endpoint If a new `Endpoint` has been added, there will be no requests targeting this `Endpoint`, because it did not exist in the old version. Therefore, we do not need to adjust anything.

Remove Endpoint In the case of removed `Endpoints`, we remove all requests to these `Endpoints`.

Add Parameter For newly added `Parameters`, we add a corresponding parameter to all requests to the corresponding `Endpoint`. Furthermore, we add a default parameter value. This value will be overwritten when the load test is parameterized (see Section 6.5.3).

Remove Parameter Similar to removed `Endpoints`, we remove all corresponding parameters from the requests.

Change Endpoint/Parameter [Property] In the case of property changes, we update the properties of all corresponding requests.

Change Input/Response Behavior Changes of these types do not affect the application model. Thus, we do not need to update or filter any requests. However, such changes can affect the annotation. Using the annotation of the new IDPA for the parameter will solve this issue.

As an example, we base on the same `Endpoints` and requests as in Section 6.5.2.1. Furthermore, we consider the following API changes:

- *hotSaucesDetails*: The path changed to */details/{sauce}*.
- *hotSaucesOverview* is removed.
- *removeFromCart*: GET */cart/remove* is added.

The resulting filtered and labeled request logs will be the following:

~~*hotSaucesOverview*: GET */hot-sauces/overview*~~

1. *hotSaucesDetails*: GET */**details**/sudden_death_sauce*
2. *addToCart*: POST */cart/add*

~~POST */hot-sauces/add*~~

The first request is labeled based on the old IDPA but removed in the filter step because the *hotSaucesOverview* `Endpoint` has been removed. The second request is labeled and kept, but the path is updated from */hot-sauces/sudden_death_sauce* to */details/sudden_death_sauce*. The third request is not changed. The last request is ignored, similar to the example without API changes before.

Because old requests can not call newly added `Endpoints` such as *removeFromCart* in the example, the generated load test will never cover these new `Endpoints`. This is a common drawback and open challenge of representative load testing that solely bases on recorded user behavior. However, testing all `Endpoints` that already existed before the API changed with a representative workload is better than not testing any `Endpoint`. For the newly added `Endpoints`, we suggest using either a manually defined load test or a generated one based on added artificial user behavior. Alternatively, request logs retrieved from production testing such as a canary release (Newman, 2015) can be used.

6.5.3. Transformation to Load Tests

The last step of the load test generation process is the transformation of the workload model and the IDPA into a load test. In this step, the workload model is transformed into a raw load test, which is parameterized with the IDPA, i.e., the manual parameterizations defined in the IDPA are merged into the load test. For the transformation of the workload model, we base on existing works such as the WESSBAS approach (Vögele et al., 2018). In this section, we describe the transformation of the IDPA into load test elements. First, we describe how we parameterize a generic load test without focusing on a specific load testing tool. Second, we provide the particular parameterization of JMeter test plans (Apache Software Foundation, 2020[a]) and BenchFlow tests (Ferme and Pautasso, 2018).

6.5.3.1. Parameterization of Generic Load Tests

Figure 6.22 provides an overview of the transformation of an IDPA to the following abstract elements of a generic load test. A `WorkloadDefinition` describes the amount and order of submitted requests. It refers to one or multiple `RequestDefinitions`, which describe the properties of one request. For instance, for HTTP requests, it defines the domain name, port number, path, and further request properties. Depending on the implementation, this reference can also be a composition. Request parameters are defined by `ParameterDefinitions`, which hold information such as the parameter name. Furthermore, a `RequestDefinition` can hold `ExtractionDefinitions`, which define the extractions of values from the responses of the submitted requests. For defining the values to be used for the parameters, each `ParameterDefinition` holds a `ValueDefinition`. For that, it can refer to an `ExtractionDefinition` of another `RequestDefinition` and also to a `GlobalDefinition`, which defines certain value sets globally. In the subsequent sections, we will illustrate the abstract elements by concrete examples of JMeter and BenchFlow.

All load test elements are generated based on the workload model and the IDPA. The `WorkloadDefinition`, `RequestDefinitions`, and `Param-`

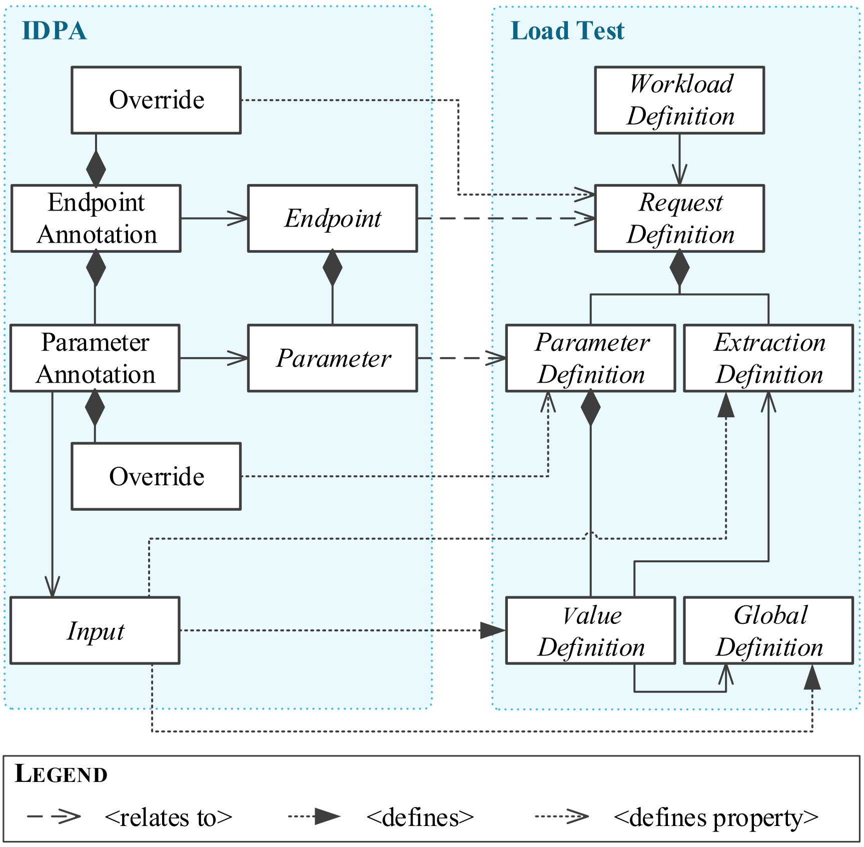


Figure 6.22.: Parameterization of a generic load test with an IDPA. For the sake of clarity, we omitted some elements of the IDPA, such as an `Application` or `ApplicationAnnotation`.

eterDefinitions constitute the raw load test, which the IDPA should parameterize. The WorkloadDefinition only relates to the workload model, e.g., the Markov chains of the WESSBAS-DSL (see Section 3.2.4.2). The RequestDefinitions and ParameterDefinitions are transformed from the workload model as well but correlate with respective Endpoints and Parameters of the IDPA. Depending on the workload model and the transformation implementation, we could also generate the RequestDefinition and ParameterDefinition based on the IDPA and correlate them with the elements of the workload model. Both ways will result in the same load test because we label the requests processed by the workload model extraction according to the IDPA (see Section 6.5.2).

When parameterizing the raw load test, properties of the RequestDefinitions and ParameterDefinitions can be changed, and ExtractionDefinitions, ValueDefinitions, and GlobalDefinitions are added. For that, we map the Endpoints and Parameters to the RequestDefinitions and ParameterDefinitions by correlating the IDs of the IDPA with the analogous concepts of the load test. As a consequence, we can correlate EndpointAnnotations and ParameterAnnotations as well. For each RequestDefinition and ParameterDefinition, we apply the Overrides of the respective EndpointAnnotation or ParameterAnnotation. Furthermore, we apply the Overrides of the higher-level scopes (not displayed in the figure). The Input of a ParameterAnnotation is transformed into several elements of the load test. First, we generate an ExtractionDefinition for the IDPA ValueExtractions of ExtractedInputs and add it to the RequestDefinition from which we want to extract the values. Second, we generate a GlobalDefinition for large input data that could be used for several parameters, such as the value list of a DirectListInput. Finally, we generate a ValueDefinition for the respective ParameterDefinition, which refers to the ExtractionDefinition or GlobalDefinition in the case we generated ones.

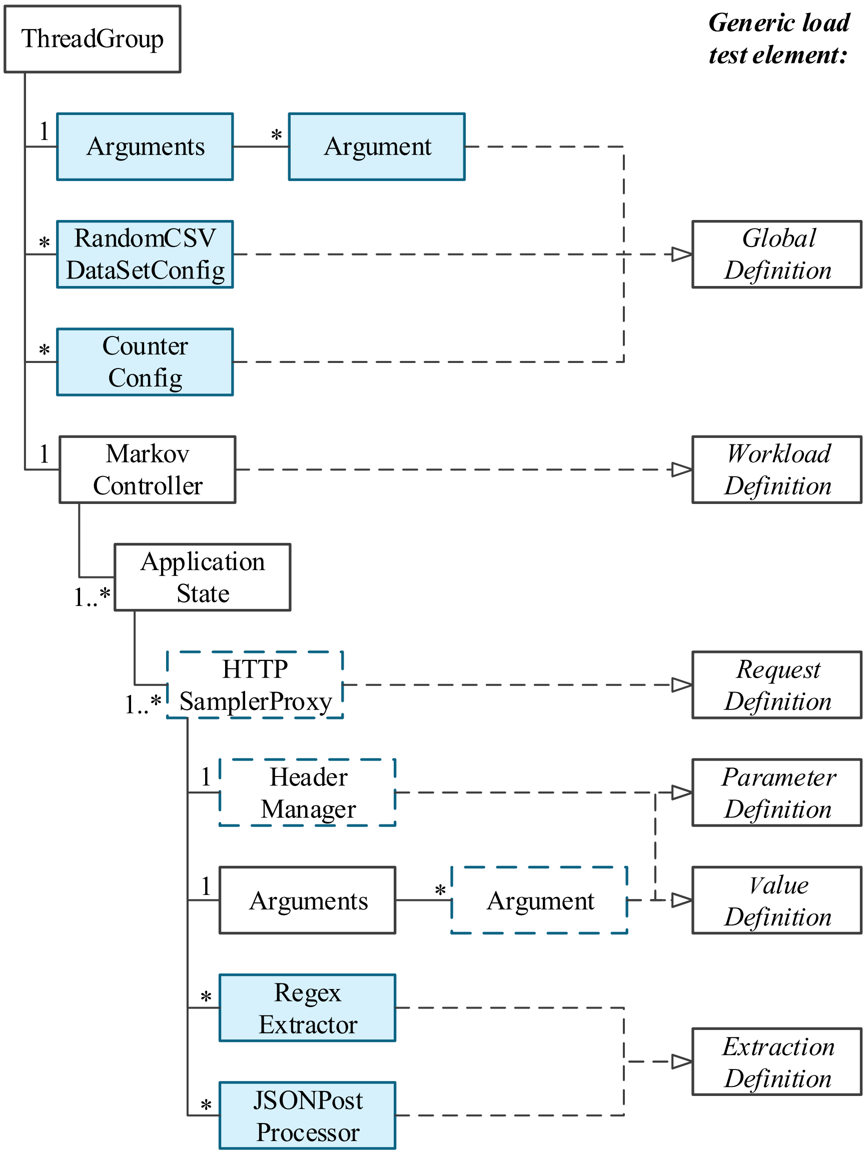


Figure 6.23.: Structure of a JMeter test plan and mapping to the generic load test elements (see Figure 6.22). We marked the entities transformed from the IDPA (filled box) and adjusted based on the IDPA (dashed border).

6.5.3.2. Parameterization of JMeter Test Plans

In this section, we describe the actual parameterization of JMeter test plans (Apache Software Foundation, 2020[a]). We base on the JMeter version 5.0 with the Markov4JMeter (van Hoorn et al., 2008) and Random CSV Data Set (Fedorov, 2017) plugins installed. First, we describe how JMeter realizes the generic concepts presented in Figure 6.22. Then, we introduce the transformation of each element of the IDPA into a JMeter element. Please note that the structure and names of elements in the JMeter UI slightly differ from the metamodel constituted by the Java API (Apache Software Foundation, 2020[b]). In the following, we refer to the Java API. As an example, Appendix B.1 provides the JMeter test plan transformed from the IDPA in Section 6.3.3.1.

Mapping to Generic Load Test Elements: JMeter test plans have a tree-based structure, whose main elements are illustrated in Figure 6.23. The root node of a JMeter test plan is a `TestPlan` entity (not displayed in the figure), which subsumes one or several subtrees with a `ThreadGroup` as the root element. The `ThreadGroup` nodes summarize a set of virtual users to be executed and are to be generated based on the workload model. For the sake of simplicity, we assume that a `TestPlan` always holds exactly one `ThreadGroup`, which is the case for the transformation of a WESSBAS-DSL instance to a JMeter test plan. However, the parameterization would work similarly with multiple `ThreadGroup` nodes. The child nodes of a `ThreadGroup` node can be elements of various other types. In the following, we limit ourselves to the types relevant for the load test parameterization.

First, nodes realize the `GlobalDefinition`, such as `Arguments`, `RandomCSVDataSetConfig`, and `CounterConfig`. They represent globally defined input data, data retrieved from CSV files, and a counter. Second, there needs to be a realization of the `WorkloadDefinition` defining the amount and order of submitted requests. For illustration purposes, we present the `MarkovController`, which defines a Markov-based user behavior and can be generated based on a WESSBAS-DSL instance. The `MarkovController` holds one or multiple ApplicationStates, which each constitute a state of the represented Markov chains and subsume the requests to be submitted as part of the state. For that, an ApplicationState holds one or multiple HTTPSamplerProxy nodes constituting the RequestDefinition of HTTP requests. For other request types, different but semantically equal nodes are used. The child nodes of an HTTPSamplerProxy can be a HeaderManager, Arguments, multiple RegexExtractors, and multiple JSONPostProcessors. The HeaderManager and the Argument elements of the Arguments define all headers and parameters to be sent with the request. These nodes include the header and parameter values as strings, which can also include function calls (see below). Hence, these two entities realize the ParameterDefinition and the ValueDefinition at the same time. The RegexExtractors and JSONPostProcessors realize the ExtractionDefinition by defining value extractions based on regular expressions or JSON paths.

The entities with the filled boxes are transformed from an IDPA as parameterizations. In contrast to that, the non-filled entities are generated by the workload model transformation. However, our IDPA transformation will adjust the properties of generated HTTPSamplerProxy nodes and its grandchild nodes of type HeaderManager and Argument. For that, we assume the *name* property of each HTTPSamplerProxy node to correspond with an ID of an IDPA Endpoint, which we achieve by labeling the requests of the request logs accordingly (see Section 6.5.2). In the following, we describe the parameterizations, which are defined by the Overrides and Inputs of the IDPA annotation.

Parameterization with Overrides: Overrides — regardless if specified under an ApplicationAnnotation, EndpointAnnotation, or ParameterAnnotation — change individual properties of either an HTTPSamplerProxy or one of its Argument grandchild nodes. These changes are summarized in Table 6.6. Overrides with a key of type HttpEndpointOverrideKey affect HTTPSamplerProxy properties. Precisely, the *domain*, *port*, *protocol*, and *base-path* keys affect the properties *domain*, *port*, *protocol*, and

Table 6.6.: Parameterization of JMeter Entities with an `Override`

Override		**JMeter**	
Key Type	Key	Entity Type	Property
`HTTPEndpoint-OverrideKey`	domain	`HTTP-SamplerProxy`	domain
	port		port
	protocol		protocol
	base-path		path
`HTTPParameter-OverrideKey`	encoded	`Argument`	always_encode

path. While `Overrides` with one of the first three keys completely override the old value of the respective property, the *base-path* only overrides the first *n* segments of the path (see Section 6.3.2.2). The *encoded* key of type `HttpParameterOverrideKey` affects the `Argument` nodes. `Overrides` with this key change the *always_encode* property of an `Argument` to *false* or *true* — depending on its value.

Parameterization with Inputs: We transform `Inputs` into the generic elements `ValueDefinition`, `ExtractionDefinition`, and `GlobalDefinition`. In JMeter, a `ValueDefinition` is realized by a string possibly including calls to predefined functions added to the *path* property of an `HTTPSamplerProxy`, the header definition of a `HeaderManager`, or the *value* property of an `Argument` — depending whether the type of the parameter the `Input` is used for is *url-part, header,* or any other type. For instance, `Inputs` used for the *sauce url-part* `Parameter` of the *hotSaucesDetails* `Endpoint` (see Section 6.3.3.1) will be transformed into a string that is added to the *path* property of the corresponding `HTTPSamplerProxy`, such that it is */hot-sauces/[string]*, whereas *[string]* denotes the value string. JMeter will evaluate the contained functions when retrieving the value. The string can also read dynamically set variables using the `${·}` operator. An `ExtractionDefinition` or `GlobalDefinition` is realized by a global `Argument`, a `RandomCSVDataSetConfig`, `CounterConfig`, `Regex-`

Extractor, or JSONPostProcessor, which will be created depending on the Input subtype and can be accessed as variables with the ${·} operator.

The transformations of all Input subtypes are summarized in Table 6.7. It shows the transformed Input subtype, the value string, and potentially additionally generated elements. We use ϕ as a symbol for the ID of the transformed Input. Furthermore, regular letters denote letters that are contained in the value strings as they are while *italic* letters refer to properties of the Input. The $|_.$ operator denotes replacing certain parts of a property with another string.

The string generated based on an ExtractedValue directly reads the ID ϕ. For the *initial value*, a global Argument is generated, setting the variable ϕ initially. For each RegExExtraction or JsonPathExtraction of the ExtractedInput, a RegexExtractor or JSONPostProcessor will be generated, which will overwrite the variable ϕ. These elements are added as child nodes of the HTTPSamplerProxies from which the values are to be extracted.

Table 6.7.: JMeter input strings for an Input with ID ϕ.

Input Type	**Value String**	**Additional Elements**
ExtractedInput	${$\phi$}	A, R, J
CsvInput	${$\phi$}	CSV
DirectListInput	${__GetRandomString(${ϕ},;)}	A
RandomNumberInput	${__Random(*lower*, *upper*,)}	–
RandomStringInput	$\mathit{template}\|_{[A]\{n\}\mapsto\$\{\_\_\mathrm{RandomString}(n,A,)\}}$	–
CounterInput	${$\phi$}	C
DatetimeInput	${__timeShift(*format*,, *offset*,,)}	–
CombinedInput	$\mathit{format}\|_{I_i\mapsto\mathrm{string}(I_i)}$	–
EnvironmentInput	${__P(*property*)}	–
JsonInput	in JSON format	–
ConciseJsonInput	in JSON format	–

A = Global Argument, CSV = RandomCSVDataSetConfig, C = CounterConfig, R = RegexExtractor, J = JSONPostProcessor

A CsvInput is also transformed into a string simply reading the variable ϕ. Furthermore, a RandomCSVDataSetConfig, which reads the input data from the CSV file and stores it in the variable ϕ, is generated and added to the ThreadGroup. Multiple CsvInputs that are connected via the *associated* attribute will be subsumed in a single RandomCSVDataSetConfig, for ensuring the values of the different columns are retrieved from the same row index. A CsvInputGroup is treated similarly as a collection of associated CsvInputs.

For a DirectListInput, a global Argument is generated, which holds the whole list of values separated by commas (,). We use ϕ as the variable name of the Argument. The value string refers to the Argument via ϕ and by calling the GetRandomString function, which randomly retrieves a value from the list.

Functions defined in the value strings realize the RandomNumberInput and RandomStringInput without additional elements. For the former type, the Random function is called to generate a random number between the defined *lower* and *upper* bounds. For the latter type, the RandomString function is used. We assume that the *template* of the RandomStringInput consists of one or multiple regular expressions in the form of [A]{n}, e.g., *[0-9A-D]{8}\-[0-9A-D]{4}\-[0-9A-D]{4}\-[0-9A-D]{4}\-[0-9A-D]{12}* for a UUID. The template is used as the value string, but each [A]{n} is replaced by a call to RandomString, which will generate a random string with n characters from the alphabet A. Because RandomString expects the alphabet as a list of characters, we enroll A. As an example, the first part of the UUID definition is replaced with ${__RandomString(8, 0123456789ABCD,)}.

A CounterInput is transformed into a CounterConfig setting the variable ϕ and a value string that reads this variable.

A DatetimeInput is transformed into a call to the timeShift function. The *format* defines the format of the generated date and time, and the *offset* defines the time to be added. We realize the EnvironmentInput as a parameter passed to the JMeter process, which can then be accessed via the __P function. The parameter — e.g., a password — can then be passed to the load test as *-Jpassword=mypassword*.

The string generated for a `CombinedInput` holds the defined *format* where each reference to another Input I_i — e.g., *(2)* — is replaced by the string generated from the corresponding `Input` from the `Input` list — e.g., the second `Input`. Finally, the `JsonInput` and `ConciseJsonInput` are transformed into a string in JSON format, i.e., to the represented JSON tree. The value strings for these `Inputs` replace references to other `Inputs`.

6.5.3.3. Parameterization of BenchFlow Tests

The second load testing tool for which we provide a transformation of the IDPA is BenchFlow (Ferme and Pautasso, 2018). We base on the BenchFlow version introduced by Palenga (2018), which uses JMeter as a test execution engine. Similar to the previous section, we describe the relation of BenchFlow test elements to the generic concepts presented in Figure 6.22 and the precise transformation of each IDPA element to a BenchFlow element. An example based on the IDPA example in Section 6.3.3.1 can be found in Appendix B.2.

Mapping to Generic Load Test Elements: BenchFlow tests are defined in the YAML format (*YAML* 2020) and, thus, have a tree-based layout. The main elements of a test that are relevant for the parameterization with an IDPA are illustrated in Figure 6.24. The central element is a `Test` (not displayed in the figure), which holds several other elements such as an `Sut` and a workload definition, e.g., an `HttpWorkload` for HTTP endpoints. The `Sut` holds a `TargetService`, which defines the domain name and base path of all called endpoints. A limitation of the BenchFlow version we use is that only one `TargetService` can be defined, i.e., it is not possible to include multiple services with different domains in one test. Hence, it is a `GlobalDefinition`.

The `HttpWorkload` subsumes the remainder of the relevant parts of the `Test`. The second type of `GlobalDefinitions` are `DataSources` denoting a set of values read from CSV files. One or several `WorkloadItems` realize the `WorkloadDefinition`. Each holds a set of `Operations` defining the requests to be submitted against the individual endpoints and a Markov-

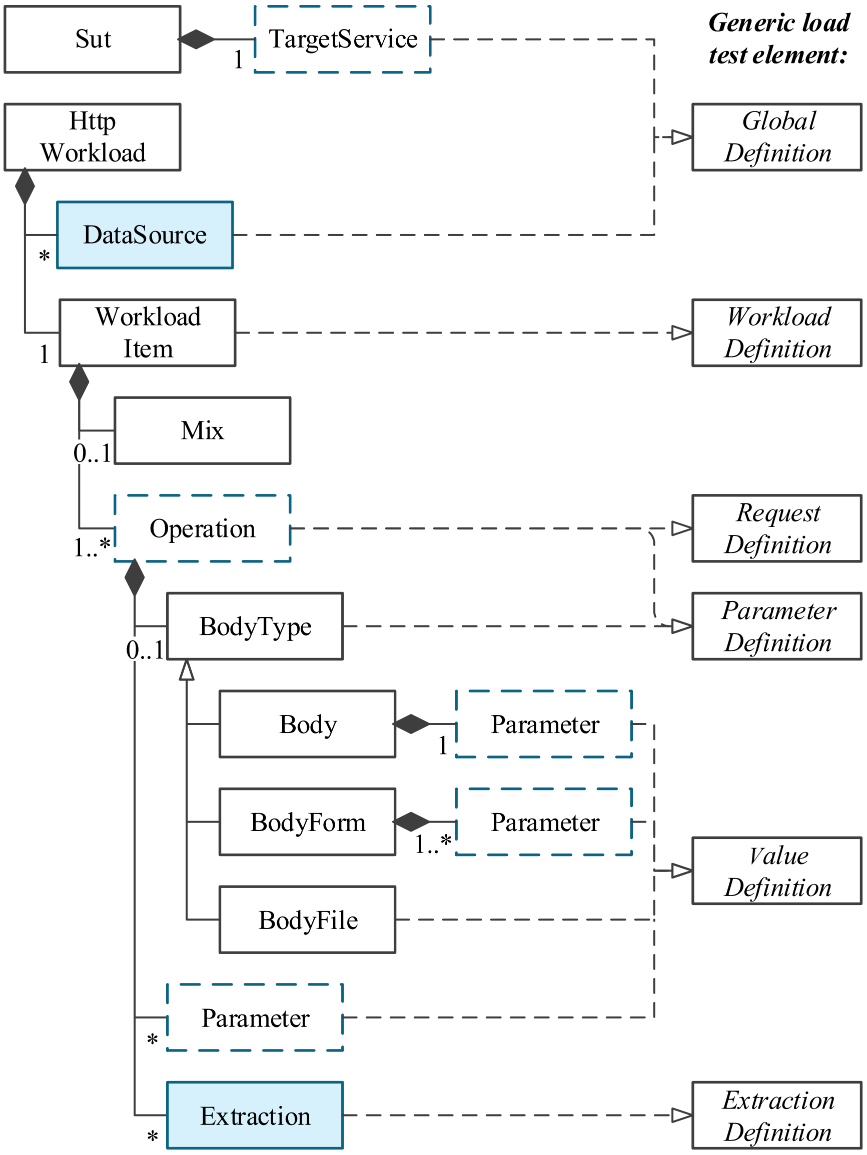


Figure 6.24.: Structure of a BenchFlow test and mapping to the generic load test elements (see Figure 6.22). We marked the entities transformed from the IDPA (filled box) and adjusted based on the IDPA (dashed border).

chain-based `Mix` defining the order of the `Operations` and the think times between the `Operations`. At the same time, each `Operation` realizes a `RequestDefinition`.

Parameters (`ParameterDefinition`) are specified as maps of `Parameters` for the IDPA parameter types *req-param* and *url-part* or a `BodyType` for body parameters. In the former case, the keys of the maps define the parameter names. In the latter case, BenchFlow differentiates between a `Body` defining a single body value, a `BodyForm` defining one or multiple *form* parameters, and a `BodyFile` defining a body value read from a file. The `Body` and `BodyForm` refer to `Parameters` for specifying the parameter values. Each `Parameter` defines a single value, list of values, or reference to a `DataSource` or `Extraction` in the *items* attribute and hence, realizes the `ValueDefinition`. An `Extraction`, of which an `Operation` can hold several, realizes the `ExtractionDefinition` and allows to specify value extractions based on regular expressions or JSON paths.

The `DataSources` and `Extractions` of a BenchFlow test will be generated based on an IDPA as a parameterization. All other entities can be transformed from a workload model. However, the parameterization with an IDPA can change certain properties of the `TargetService`, the `Operations`, and `Parameters`. For that, we assume the *id* property of an `Operation` is equal to the ID of the corresponding IDPA `Endpoint`. BenchFlow `Parameters` can be correlated with IDPA `Parameters` based on the parameter name.

Parameterization with Overrides: IDPA `Overrides` are applied to a BenchFlow test as summarized in Table 6.8. `Overrides` using the *domain*, *port*, and *base-path* `HTTPEndpointOverrideKeys` change the *endpoint* of the `TargetService`. For instance, the domain name *localhost*, the port number *8080*, and the base path */test/stage* will change the *endpoint* to *localhost:8080/test/stage*. In addition, the *base-path* key can change the first segments of the *path* of each `Operation` in the case of different base paths for different `Operations`. The *protocol* key refers to the *protocol* property of each *Operation*. In the used BenchFlow version, there is no concept for ex-

Table 6.8.: Parameterization of BenchFlow Entities with an `Override`

Override		**BenchFlow**	
Key Type	Key	Entity Type	Property
	domain		
`HTTPEndpoint-OverrideKey`	port	`TargetService`	endpoint
	base-path		
	protocol	`Operation`	protocol
`HTTPParameter-OverrideKey`	encoded	`Parameter`	items
		CSV file	content

Table 6.9.: BenchFlow `Parameter` *Items* for an `Input` with ID ϕ

Input Type	**Item(s)**	**Additional Elements**
`ExtractedInput`	$\$\{\phi\}$	`Extraction`
`CsvInput`	$\$\{\phi\}$	`DataSource`
`DirectListInput`	*data*	–
`RandomNumberInput`	$\$\{\_\_\text{Random}(lower, upper,)\}$	–
`RandomStringInput`	$template\|_{[A]\{n\}\mapsto\$\{\_\_\text{RandomString}(n, A,)\}}$	–
`CounterInput`	$\$\{\phi\}$	`DataSource`
`DatetimeInput`	similar to JMeter	–
`EnvironmentInput`	similar to JMeter	–
`CombinedInput`	similar to JMeter	–
`JsonInput`	similar to JMeter	–
`ConciseJsonInput`	similar to JMeter	–

plicitly encoding an already defined parameter value. Therefore, the *encoded* `HTTPParameterOverrideKey` entails encoding the defined values directly in the BenchFlow test and the accompanying CSV files. Values extracted by an `Extraction` cannot be encoded.

Parameterization with Inputs: Because the used BenchFlow version utilizes JMeter as an execution engine, we can transform IDPA `Inputs` very similarly to JMeter. In particular, we can reuse the JMeter functions and the `${·}` operator for accessing a variable. For each `Input`, the *items* attribute of the

BenchFlow Parameters the Input is to be used for is adjusted. Table 6.9 summarizes the transformed *items* attribute and potentially additionally generated elements for an Input with ID ϕ.

The DirectListInput is the only Input for which the BenchFlow Parameter holds multiple *items*. It holds the defined *data* list. For all other Inputs, only one *item* is generated. For an ExtractedInput, the generated *item* refers to the Input ID ϕ. Furthermore, Extractions are added to all Operations from which a value is to be extracted. Depending on the used type of ValueExtraction, the Extraction will hold a regular expression or a JSON path.

A CsvInput is transformed into ${$\phi$} and a DataSource defining how to read the CSV file. In the case of multiple CsvInputs that are connected via the *associated* property, only one DataSource will be generated. The same applies to a CsvInputGroup. Because BenchFlow reads the variable name for each CSV column from the column header, we add ϕ as a header of the respective column in the file.

The RandomNumberInput and RandomStringInput are transformed into the same function calls as in JMeter. As there is no corresponding entity for a CounterInput, we transform it to a DataSource. We generate all values the CounterInput can generate and store it into a CSV file. Then, we handle it similarly to a CsvInput. Finally, the DatetimeInput, EnvironmentInput, CombinedInput, JsonInput, and ConciseJsonInput are transformed similarly to JMeter and do not require additional elements.

6.6. Integration into Continuous Software Engineering

IDPAs are designed for use in CSE and, therefore, in CI/CD pipelines. Precisely, IDPAs are to be used to generate a load test based on spontaneously selected user behavior — e.g., the most recent one — automatically within a CI/CD pipeline. In the previous sections, we already described how to evolve an IDPA and automatically generate a load test, which are fundamental requirements. In the following, we describe the integration into the CSE life cycle. The integration includes evolving the IDPA in the local development environment, persisting in a repository, and generating and executing a load test as part of a CI/CD pipeline.

Figure 6.25 illustrates the intended use of IDPAs in CSE. In the local development environment, a developer develops, extends, and maintains the program code ①. Hence, all API changes are introduced by the developer in this environment. Therefore, the developer is responsible for creating and evolving the IDPA based on the introduced API changes. For this, they rely on the evolution strategies described in Section 6.4.

After the developer implemented a code change and evolved the IDPA accordingly, both are committed to the central code repository ②. We recommend storing the IDPA in the code repository, because it is close to the program code and hence, close to possible API changes. The same principle is applied for unit tests, which are typically stored next to the program code as well. Because the program code and the IDPA are committed simultaneously, we ensure that the IDPA in the code repository always fits the developed application's API.

Generating representative load tests based on recorded user behavior requires the application to run and be used in a production environment ③. From this execution, request logs representing the user behavior are to be collected ④, which can be used to generate a load test.

Based on the up-to-date IDPA in the code repository and the recorded request logs, we can automatically generate a new load test in the CI/CD pipeline and execute it ⑤. In this way, we allow using spontaneously selected user behavior in the CI/CD pipeline.

Because newly introduced API changes are always first tested in the CI/CD pipeline before being deployed to the production environment, the described procedure has the main drawback that after an API change, no request logs are fitting the new API. In such a case, we apply the labeling approach described in Section 6.5.2.2, which maps the requests of the old API to the new one. For this, we need to use the two IDPA versions corresponding to the API versions, which we can easily retrieve from the code repository. This approach entails the drawback that newly introduced endpoints are not tested. However, as canary releases or other production testing approaches are common in CSE (Newman, 2015), they can be utilized to collect request logs for the new API before the actual release. These logs can be used for generating a load test covering the whole API.

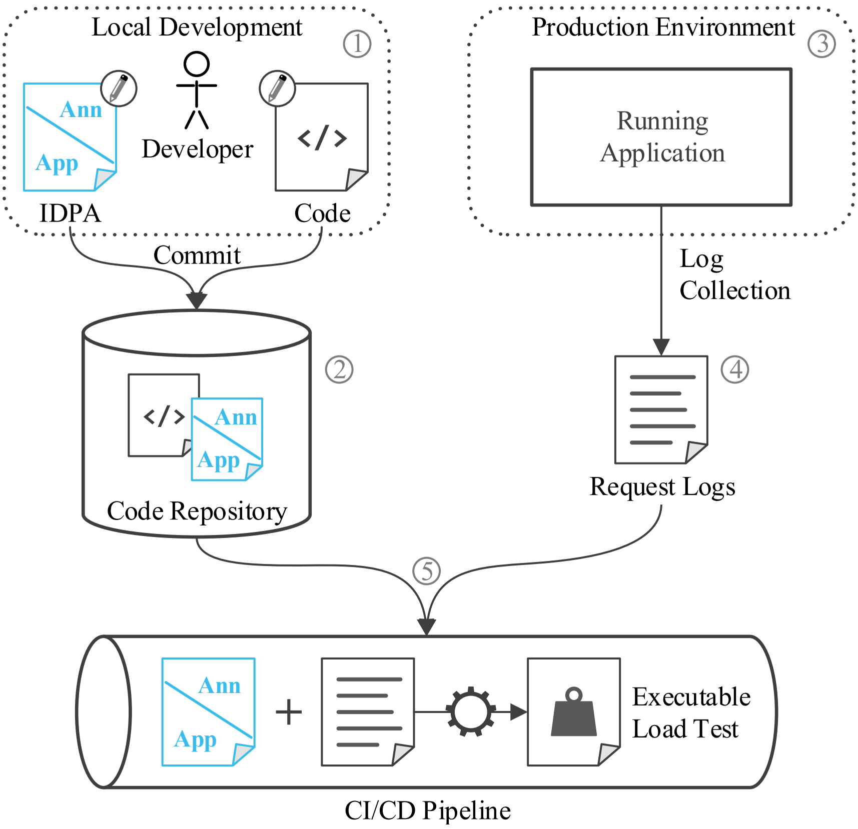


Figure 6.25.: Use of an IDPA in a CI/CD pipeline.

6.7. Summary

In this chapter, we introduced our approach to the automated parameterization of generated load tests, addressing RQ1: *How can load test parameterizations be evolved without manual intervention at test generation or execution time?*

We store the parameterizations an expert needs to do manually in a separate model — the Input Data and Properties Annotation (IDPA) — and merge them into the load test when it is generated. Instead of overwriting the parameterizations, this separation preserves them, also if the workload changes. Furthermore, it allows us to evolve the parameterizations over typical API changes, as recorded in the literature, based on an expert's feedback. Various transformations — e.g., from or to API specifications, workload models, and load tests — and the use of the YAML format allow a neat integration into CSE. In Chapter 12, we provide an evaluation of the approach.

The automated parameterization forms the basis of our overall approach. It allows us to generate tailored load tests without manual intervention. Hence, we can integrate it into a CI/CD pipeline. For instance, a user only needs to describe the load test, and our approach can generate the test when it is required, based on the most recent production workload. The spontaneous load test generation is furthermore helpful because the space of possibly generated load tests is vast. As an example, load tests can target different sets of services of a distributed application and can represent the workload scenarios of different contexts. In the next two chapters, we will introduce two approaches generating corresponding load tests and making use of the automated parameterization. First, we introduce tailoring to a given set of services. Second, we present an approach to generating context-tailored load tests.

Chapter 7

Tailoring Load Tests to Microservices

Generating a representative load test for a session-based microservice application requires a focus on the service level. As multiple teams are typically responsible for developing, testing, deploying, and operating one service of the application independently following DevOps practices (Bass et al., 2015; Newman, 2015), each team requires its load tests tailored to the developed service. Alternatively, for performing integration tests between services, they require load tests tailored to the set of tested services. Untailored tests targeting the whole system would require deploying the complete application for the test execution, which results in unnecessary resource consumption. Notably, as each team uses its independent continuous integration and delivery (CI/CD) pipeline, system-level tests can result in deploying the application multiple times in parallel. Furthermore, such a procedure contradicts the individuality of the teams, as they would need to deal with foreign services. At the same time, a tailored representative load test needs to preserve the inter-relations of the requests of each user session, which are only explicitly present at the user-faced services.

Several works targeting the described challenge exist. Different approaches allow extracting representative load tests from user sessions, focussing on the tested application as a whole (Barros et al., 2007; Cai et al., 2007; Krishnamurthy et al., 2006; Lutteroth and Weber, 2008; Menascé and Almeida, 2002; Ruffo et al., 2004; Vögele et al., 2018). Approaches using request-based workload models (Barford and Crovella, 1998) can extract workload models from requests missing a session context. Finally, approaches translating workload models from higher-level to lower-level constructs exist (Graf, 1987; Koziolek, 2008; Koziolek and Reussner, 2008). However, these approaches lack in generating session-based workload models tailored to specific services. The load test extraction approaches either cannot consider the services or the session context. The last mentioned approaches comprise concepts helping to tailor system-level workload models to service-level ones but are not adopted in load testing, yet.

Therefore, we aim at extending the existing approaches to the generation of session-based workload models, introducing means for tailoring the workload models to specific services. We want to reduce the overall used resources for load testing and enable DevOps teams to focus on their services. In doing so, we focus on Markov-chain-based workload models such as the WESSBAS-DSL (Vögele et al., 2018). We address the research question RQ2 defined in Section 5.1: *How can representative load tests be tailored to specific services of a session-based application?*

For identifying possible extensions of the existing approaches, we analyze the typical load test extraction process. For that, we base on the WESSBAS approach (Vögele et al., 2018). Then, we introduce two algorithms modifying the artifacts of certain stages of the process. *Log-based tailoring* adjusts the collected request or session logs before aggregating them to a workload model. *Model-based tailoring* modifies the workload model generated by the existing approaches. Both algorithms change the process such that the finally generated load tests are tailored to a set of desired services. In our evaluation, we will compare the quality of the models produced.

The remainder of the chapter is structured as follows. Section 7.1 provides a motivating example and puts our approach into context. Section 7.2

summarizes the notations used in this chapter. In Section 7.3, we explain the load test extraction process we extend and motivate the choice of algorithms. In Sections 7.4 and 7.5, we introduce the log-based and model-based tailoring algorithms. Section 7.6 illustrates the integration of the approach described in this chapter with the automated load test parameterization described in Chapter 6. Finally, we summarize the chapter in Section 7.7.

This chapter is a revised and extended version of Section 4 of our below publication. Besides the integration with the automated load test parameterization (Section 7.6), all extensions are for explanation or (formal) concretization purposes only and do not add additional research results.

- H. Schulz, T. Angerstein, D. Okanović, and A. van Hoorn (2019a). “Microservice-tailored Generation of Session-based Workload Models for Representative Load Testing.” In: *Proceedings of the 27th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2019)*. IEEE Computer Society, pp. 323–335

7.1. Motivating Example

For motivating the need for tailored load tests and putting them into context, we introduce the Sock Shop Microservices Demo (Weaveworks, Inc., 2020), which we also use in the evaluation (see Chapter 13). Consisting of the services shown in Figure 7.1, it constitutes a webshop for socks with functionalities such as browsing products, adding them to a cart, and purchasing. Aderaldo et al. (2017) have assessed the Sock Shop to be a representative microservice application; hence, we use it as a representative example.

Assuming the services are developed following DevOps practices, they are to be load-tested individually. As each team is responsible for one service, they need to focus on only the developed one. As an example, the team responsible for the *orders* service needs a load test directly targeting *orders*. Besides, it needs stubs mimicking the functional and performance behavior of the dependent services, for being able to test orders in isolation.

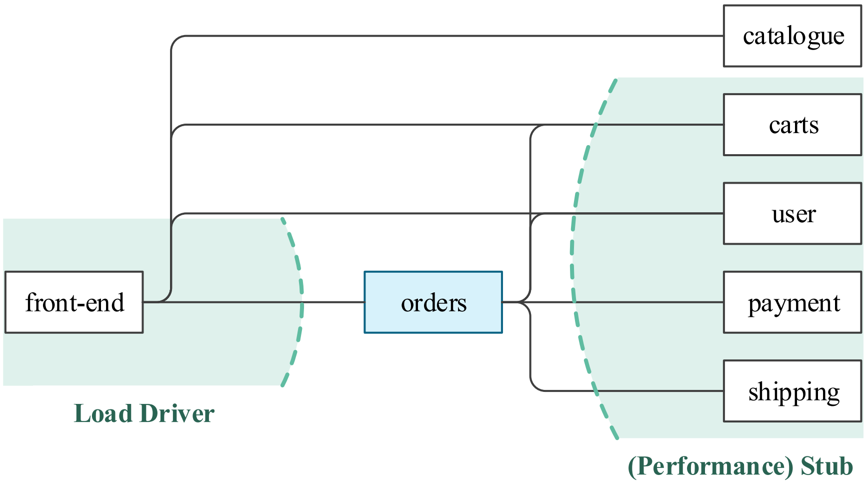


Figure 7.1.: Services of the Sock Shop and illustration of load drivers and stubs required for testing the *orders* service in isolation.

Integration testing (Myers, 2004) uses similar techniques. There are two basic approaches: top-down testing starts with the top-most service (left-most in Figure 7.1), replacing dependent ones with stubs; bottom-up testing starts with the bottom-most services (right-most in the figure), replacing calling ones with drivers. For full isolation of, e.g., the *orders* service, we need to apply both approaches at the same time. In this case, we need to replace the *front-end* with a load driver and the *carts*, *user*, *payment*, and *shipping* services with stubs.

In this dissertation, we only focus on the bottom-up approach. For top-down testing, i.e., replacing dependent services with stubs, we refer to existing work (Baltas and Field, 2012; Becker et al., 2008; Field et al., 2018; Versteeg et al., 2016). However, generating session-based representative load tests serving as drivers in this scenario is missing in the literature. Therefore, we introduce two algorithms extending existing approaches for tailoring a load test to individual services such as *orders* or a set of services such as *orders* in combination with *catalogue*.

The major challenge when generating service-tailored load tests is the sessions, in which the end-users interact with the application. The application identifies a session by a session ID, which it sends to the end-users, who reuse the ID in subsequent requests. The services responsible for the IDs are the user-faced ones—*front-end* in the example. For computing the response to a request, the *front-end* calls further services, including *orders*. However, such sent requests do not have a session ID. Hence, only focusing on orders misses this information. As an example, if multiple users are checking the status of their orders, which, for instance, includes two subsequent requests to the *orders* service, this service misses a relation of these requests to the users and the request sequence. Simply replaying the requests at the observed rates in a load test will most likely result in different inter-request time distributions and disallows for user-based parameterization, such as using one order per user.

Therefore, we require algorithms that restore the session context. Precisely, the algorithms need to assign session IDs to requests of non-user-faced (backend) services. In the next two sections, we introduce the notations used and provide an overview of the algorithms, which act at different stages of the load test extraction process.

7.2. Notations

In this chapter, we use several notations, which we summarize in this section. Table 7.1 provides an overview. In the next section, we will detail the relation of the notations to the load test extraction process. We consider an application with n (micro) services, e.g., the Sock Shop, with its seven services. Each service m_i has a set of endpoints E_i, e.g., *POST /carts/{id}/items*. M and E denote the entirety of all existing services and endpoints. We use the script font $\mathcal{M}, \mathcal{E}$ for denoting selected subsets used for tailoring a specific workload model.

Request logs are designated as $\mathcal{R}$ and comprise all end-user requests that have been observed at the entry points of the application, e.g., the endpoints of the *front-end* service. We characterize each request $r \in \mathcal{R}$ by the called endpoint $\varepsilon(r)$, a user session ID $\phi(r)$, a start timestamp $t(r)$, and a duration $\delta(r)$. Trace logs $\mathcal{T}$ add to $\mathcal{R}$ the additional dimension of the requests internally made by the services to process the end-user request. We define each trace $\tau \in \mathcal{T}$ as a directed tree consisting of the root request $r_\tau \in \mathcal{R}$, further requests (nodes) R_τ to the different services, and the call relations (edges) C_τ. For convenience, we also use ϕ, t, and δ for traces: $\phi(\tau = (r_\tau, \cdot, \cdot)) := \phi(r_\tau)$. Furthermore, we use $r \in \tau$ as a shorthand for $\exists (r_1, r_2) \in C_\tau : r = r_1 \vee r = r_2$.

Session logs $\mathcal{S}$ are a different log format, which we obtain by grouping the requests in $\mathcal{R}$ by the session ID:

$$\forall s \in \mathcal{S}\ \forall r_1, r_2 \in s : \phi(r_1) = \phi(r_2)$$

and by sorting them according to the time stamp:

$$\forall s \in \mathcal{S}\ \forall r_1, r_2 \in s : r_1 \leq r_2 \iff t(r_1) \leq t(r_2)$$

A workload model $\mathcal{W}$ aggregates session logs $\mathcal{S}$ clustered by similar user behavior. A workload model consists of one Markov chain W_j — corresponding to a WESSBAS behavior model — per session cluster and a weighting function f defining the workload mix. Each Markov chain consists of states Σ_j, a mapping of the states to endpoints ε_j, a transition probability function p_j, and a think time distribution function Δ_j. Furthermore, we record the session cluster $\mathcal{S}_j$. Σ_j allows for multiple states having the same endpoint. That will be relevant for the model-based tailoring algorithm. The think times follow a normal distribution. While in general, WESSBAS and other approaches allow for different distributions as well, we chose the normal distribution in this work and leave further ones for future work. Finally, we denote the linear combination (convolution; Montgomery and Runger, 2003) of two normal distributions Δ_1, Δ_2, as follows:

$$\alpha\Delta_1 * \beta\Delta_2 \sim \mathcal{N}(\alpha\mu_1 + \beta\mu_2, \alpha^2\sigma_1^2 + \beta^2\sigma_2^2)$$

Table 7.1.: Table of Notations

Notation	**Explanation**
$M = \{m_1, \ldots, m_n\}$	existing (micro) services, $\lvert M\rvert = n$
$\mathcal{M} \subset M$	(micro) services of a tailored workload model
$E_i = \{e_1, \ldots, e_{k_i}\}$	endpoints of m_i, $\lvert E_i\rvert = k_i$
$E = \bigcup_{i=1}^{n} E_i$	endpoints of all services
$\mathcal{E} \subset E$	endpoints of a tailored workload model
$\mathcal{R}$	request logs
$\varepsilon : \mathcal{R} \to E$	called endpoint of request
$\phi : \mathcal{R} \to \mathbb{R}$	session ID of request
$t : \mathcal{R} \to \mathbb{R}$	time stamp of request
$\delta : \mathcal{R} \to \mathbb{R}$	duration of request
$\mathcal{T}$	trace logs
$\tau = (r_\tau, R_\tau, C_\tau) \in \mathcal{T}$	trace tree with root request r_τ and vertices R_τ
$C_\tau \subset R_\tau \times R_\tau$	call relations/edges of a trace τ
$\mathcal{S} \subset \mathcal{R}^*$	session logs ($\mathcal{R}^* = \bigcup_{i=1}^{\infty} \mathcal{R}^i$)
$\mathcal{W} = (\{W_1, \ldots, W_q\}, f)$	workload model with q behavior models
$f : \{W_1, \ldots, W_q\} \to [0,1]$	relative frequency of behavior model
$W_j = (\Sigma_j, \varepsilon_j, p_j, \Delta_j, \mathcal{S}_j)$	behavior model with states Σ_j †
$\varepsilon_j : \Sigma_j \to \mathcal{E} \cup \{I, \$\}$	mapping of states to endpoints†
$I, \$$	initial and final state of a behavior model
$p_j : \Sigma_j \times \Sigma_j \to [0,1]$	state transition probability†
$\Delta_j : \Sigma_j \times \Sigma_j \to \mathcal{N}(\mu, \sigma)$	think time (normal) distribution†
$\mathcal{S}_j \subset \mathcal{S}$	sessions aggregated in W_j
$\alpha\Delta_1 * \beta\Delta_2$	convolution of normal distributions Δ_1, Δ_2

† In H. Schulz et al. (2019a), we modeled multiple states with the same endpoint implicitly. Here, Σ_j and ε_j allow for this explicitly.

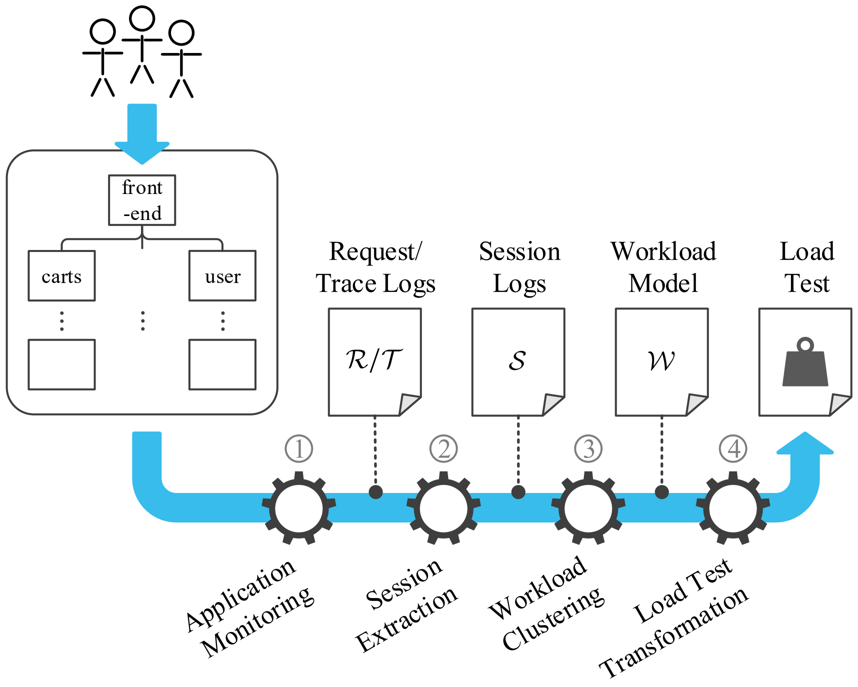


Figure 7.2.: Load test extraction process.

7.3. Overview of Tailoring Algorithms

In this section, we motivate the choice of algorithms. First, we define the process of extracting representative session-based load tests from monitored user behavior, related to the process used in previous chapters and using the notations introduced in the previous section. Figure 7.2 illustrates it. We presume a microservice application is running in production, used by end-users. As a first step, application monitoring collects request logs $\mathcal{R}$, containing the end-users' requests ①. For the algorithms introduced in this chapter, we additionally collect the trace logs, $\mathcal{T}$. Next, session logs $\mathcal{S}$ are extracted from $\mathcal{R}$ ②. In a third step, the workload clustering extracts a workload model $\mathcal{W}$ from $\mathcal{S}$ ③. Finally, the workload model is transformed

]]>

into a load test and parameterized to be executable (see Chapter 6) ④. Without modification, the described process generates untailored load tests targetting the whole application.

For tailoring a generated load test, we introduce algorithms modifying the process at different stages. For that, we need the set of target services $\mathcal{M}$ and the corresponding endpoints $\mathcal{E}$ as additional inputs to the process. All algorithms have in common that they use the trace logs $\mathcal{T}$ for tailoring. The output of such a modified process is a load test tailored to the endpoints $\mathcal{E}$.

Based on the load test extraction process, we identify four approaches to reach this goal. The simplest one is to record only the requests arriving at $\mathcal{E}$ — i.e., the endpoints of the tested services — and to specify the rate at which the end-users request each endpoint. As motivated previously, such a request-based workload model will be able to replay the mean request rates correctly, but misses relevant session information, which is required for preserving the request orders and inter-request time distributions. Therefore, we aim at developing more elaborate algorithms but use the request-based workload model as a baseline in our evaluation.

The first proposed algorithm modifies the request logs $\mathcal{R}$, measured at the endpoints requested by the end-users and, thus, containing the session IDs $\phi(\cdot)$. By using the trace corresponding to each request $r \in \mathcal{R}$, we can replace r by all requests targeting $\mathcal{E}$ while preserving the session ID $\phi(r)$. Such modified request logs only target $\mathcal{E}$ and allow to reuse the remaining steps of Figure 7.2. We denote this algorithm as *log-based tailoring*.

A second algorithm could address the session logs $\mathcal{S}$ and replace each request similar to the first algorithm. However, this would lead to exactly the same result as the first algorithm does. Therefore, we do not differentiate between these two algorithms and only focus on the first one.

Finally, an algorithm can modify the workload model $\mathcal{W}$, which we refer to as *model-based tailoring*. In doing so, the algorithm needs to replace each state — which aggregates several requests to the respective endpoint — in each Markov chain W_j by the aggregate control flow caused by such a request at the endpoints $\mathcal{E}$. For instance, an end-user request *POST /orders* to the *front-end* can cause a *POST /orders* request to the *orders* service and several requests to the *user* service, in potentially varying sequences. The replacement of the original Markov state needs to reflect this calling behavior.

In the next two sections, we introduce the log-based and model-based tailoring algorithms in detail.

7.4. Log-based Tailoring

This section introduces the log-based tailoring algorithm. It operates on the request logs $\mathcal{R}$. Essentially, the request logs are the only artifact of the load test extraction process that the algorithm modifies. Steps ② to ④ of Figure 7.2 remain unchanged. As described earlier, it is also possible to use the session logs $\mathcal{S}$ and run the procedure on them, with the same results.

In the following, we describe the inputs and outputs of the algorithm, a helper algorithm, and the algorithm itself.

7.4.1. Input and Output

The log-based tailoring algorithm takes the following inputs:

- the recorded trace logs $\mathcal{T}$, which implicitly contain $\mathcal{R}$ as root requests, i.e., $\mathcal{R} = \{r_\tau \mid \tau \in \mathcal{T}\}$,
- the mapping of requests to session IDs: $\phi : \mathcal{R} \to \mathbb{R}$,
- the endpoints $\mathcal{E}$ of a set of services $\mathcal{M} = \{m_1, \dots, m_n\}$, which the resulting load test should target, i.e., $\mathcal{E} = \bigcup_{i=1}^{n} E_i$ for endpoints E_i of m_i.

Based on the inputs, the algorithm calculates the following outputs:

- request logs $\mathcal{R}'$, which directly target the endpoints $\mathcal{E}$,
- a new mapping of requests to session IDs: $\phi' : \mathcal{R}' \to \mathbb{R}$.

For defining the intended output precisely, we define two postconditions. In Chapter 13, we will use them to validate the proposed algorithm. First, the tailored request logs $\mathcal{R}'$ need to contain those requests that have an endpoint in $\mathcal{E}$ and for which no other request (transitively) calling it has an endpoint in $\mathcal{E}$. Only considering these top-most requests is essential. During the load test execution, the services will send the lower-level requests as a reaction to the test's top-level requests. Hence, the load test should not duplicate the lower-level ones. Let $P_{\mathcal{E}}(\tau, r)$ be the predicate whether a trace τ has a path (potentially of length 0) to a request r where all nodes (requests) except for r are not in $\mathcal{E}$:

$$P_{\mathcal{E}}(\tau, r) \equiv \exists r_1, \dots r_l : r_1 = r_\tau \land r_l = r \land \big(\forall i < l : (r_i, r_{i+1}) \in C_\tau \land \varepsilon(r_i) \notin \mathcal{E}\big)$$

Then, we formally define the postcondition $A^{(\log)}_{\text{req}}$ as follows:

$$A^{(\log)}_{\text{req}} \equiv \forall r \in R : \big(r \in \mathcal{R}' \leftrightarrow \exists \tau \in \mathcal{T} : \varepsilon(r) \in \mathcal{E} \land P_{\mathcal{E}}(\tau, r)\big) \tag{7.1}$$

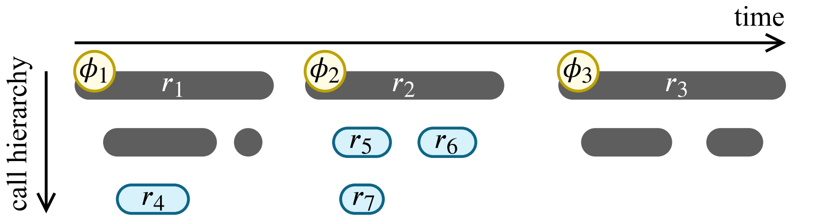
The second postcondition $A^{(\log)}_{\text{ID}}$ considers the new session ID $\phi'(r)$ of each request in $\mathcal{R}'$, which needs to equal the corresponding trace's one:

$$A^{(\log)}_{\text{ID}} \equiv \forall r \in \mathcal{R}' \; \forall \tau = (r_\tau, R_\tau, C_\tau) \in \mathcal{T} : r \in \tau \rightarrow \phi'(r) = \phi(r_\tau) \tag{7.2}$$

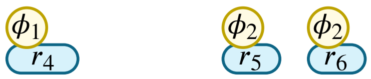
7.4.2. Example

We illustrate the intended behavior of the algorithm in an example. As shown in Figure 7.3, we consider trace logs $\mathcal{T}$ consisting of three traces with root requests r_1 to r_3, which we have recorded from the production system. The requests have different session IDs ϕ_1 to ϕ_3; hence, they belong to different sessions. The root requests made further requests, which are displayed below the root ones, e.g., r_2 made requests r_5 and r_6. Requests targeting an endpoint in $\mathcal{E}$ are highlighted.

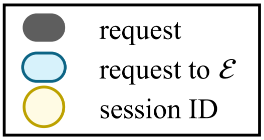
The output consists of three requests replacing r_1 to r_3, which would form untailored request logs. In the trace of r_1, there is one request targeting $\mathcal{E}$: r_4. Hence, r_4 replaces r_1. Notably, the tailored request logs reflect the start time and duration of r_4, which differ from the ones of r_1. Furthermore, r_4 has the session ID ϕ_1 of r_1. Hence, if there were further requests inside the session with ID ϕ_1, their child requests would have ϕ_1, too.



(a) Input: trace logs



(b) Output: tailored request logs



(c) Legend

Figure 7.3.: An example of tailoring requests r_1 to r_3 to $\mathcal{E}$.

In the trace of r_2, there are three requests r_5 to r_7 that target $\mathcal{E}$. However, the tailored request logs only contain r_5 and r_6. Because r_7 is a child request of r_5 and r_5 targets $\mathcal{E}$, too, only r_5 as the most top-level request is part of the output. Similar to r_4, r_5 and r_6 have the session ID of their root request, which is ϕ_2.

Finally, the trace of r_3 does not have any requests targetting $\mathcal{E}$. Therefore, there is no request in the output.

We split the algorithm for the log-based tailoring into two. TAILORREQUEST (Algorithm 7.1) takes a single trace as input and extracts all requests from it that should be part of the tailored request logs. For instance, it would extract r_5 and r_6 from r_2. TAILORREQUESTLOGS (Algorithm 7.2) uses TAILORREQUEST for tailoring a set of traces — i.e., trace logs — and assigns the correct session IDs to each extracted request. In the next two sections, we introduce both algorithms.

Algorithm 7.1 Extract sub-requests targeting endpoints $\mathcal{E}$ from trace τ

```
function TAILORREQUEST($\tau$, $\mathcal{E}$)
    $\mathcal{R}' \leftarrow \emptyset, \quad \overline{T} \leftarrow \{\tau\}$
    while $\overline{T} \neq \emptyset$ do
        $\overline{\tau} \leftarrow (\overline{r}, \overline{R}, \overline{C}) \in \overline{T}$                ▷ arbitrarily selected
        $\overline{T} \leftarrow \overline{T} \setminus \{\overline{\tau}\}$
        if $\varepsilon(\overline{r}) \in \mathcal{E}$ then
            $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{\overline{r}\}$
        else
            $\overline{T} \leftarrow \overline{T} \cup \{(r', \overline{R}, \overline{C}) \mid r' \in \overline{R} \wedge (\overline{r}, r') \in \overline{C}\}$
        end if
    end while
    return $\mathcal{R}'$
end function
```

7.4.3. Algorithm for Tailoring a Single Request

Before we introduce the actual tailoring algorithm, we provide a helper algorithm tailoring a single request to the endpoints $\mathcal{E}$ — TAILORREQUEST (Algorithm 7.1). It takes one trace $\tau \in \mathcal{T}$ and the endpoints $\mathcal{E}$ as inputs and performs a breadth-first search for collecting all relevant requests from τ. The algorithm processes a set $\overline{T}$ of sub traces, initialized with τ (line 2), and continues until $\overline{T}$ is empty. In each iteration, the algorithm selects one trace $\overline{\tau}$ and removes it from $\overline{T}$ (lines 4 and 5). Then, it checks whether the root request $\overline{r}$ of $\overline{\tau}$ targets one of the endpoints in $\mathcal{E}$ (line 6). If so, the algorithm adds $\overline{r}$ to the resulting set of request logs (line 7). Otherwise, it extracts all sub traces from $\overline{\tau}$ and adds them to $\overline{T}$ (line 9).

As an example, we consider the algorithm processes the trace with root request r_2 from our example (Figure 7.1). In the first iteration, it processes the whole trace. As r_2 does not target $\mathcal{E}$, TAILORREQUEST adds the two sub traces with roots r_5 and r_6 to $\overline{T}$. Then, it processes them in arbitrary order. As both of them target $\mathcal{E}$, the algorithm adds them to $\mathcal{R}'$ and finishes, because $\overline{T}$ is now empty. The breadth-first search ensures that only the highest-level requests that target $\mathcal{E}$ are collected, e.g., r_7 is not part of $\mathcal{R}'$ because r_5 is already.

Algorithm 7.2 Extract request logs tailored to endpoints $\mathcal{E}$ from traces $\mathcal{T}$[*]

```
1: function TAILORREQUESTLOGS($\mathcal{T}$, $\phi$, $\mathcal{E}$)
2:     $\mathcal{R}' \leftarrow \emptyset, \quad \phi' \leftarrow \phi$
3:     for all $\tau \in \mathcal{T}$ do
4:         $\mathcal{R}'' \leftarrow$ TAILORREQUEST($\tau$, $\mathcal{E}$)
5:         $\phi'(\mathcal{R}'') \leftarrow \phi(\tau)$
6:         $\mathcal{R}' \leftarrow \mathcal{R}' \cup \mathcal{R}''$
7:     end for
8:     return $\mathcal{R}'$, $\phi'$
9: end function
```

[*] Opposed to H. Schulz et al. (2019a), the algorithm returns a new session ID mapping ϕ' instead of adjusting ϕ.

7.4.4. Log-based Tailoring Algorithm

The log-based tailoring algorithm takes as input recorded trace logs $\mathcal{T}$— which implicitly contain $\mathcal{R}$ as root requests—, the session ID mapping ϕ, and the endpoints $\mathcal{E}$. The outputs are the request logs $\mathcal{R}'$ tailored to $\mathcal{E}$ and a new session ID mapping ϕ'. As described earlier, it is also possible to use the session logs $\mathcal{S}$ and run the procedure on them, with the same results.

We name the algorithm TAILORREQUESTLOGS and present it in Algorithm 7.2. Initially, it starts with empty tailored request logs $\mathcal{R}'$ and an unchained session ID mapping ϕ'. Then, it iterates over all traces τ in $\overline{T}$ and calls TAILORREQUEST for getting the relevant requests from τ (line 4). It sets the session IDs of all these requests to $\phi(\tau)$ (line 5) and appends the requests to $\mathcal{R}'$ (line 6).

Again, we consider the example from Figure 7.1. Given TAILORREQUEST-LOGS iterates over the traces from left to right, it first uses TAILORREQUEST to extract r_4 as a sub-request of r_1. Then, it sets $\phi'(r_4) := \phi(r_1) = \phi_1$ and adds r_4 to $\mathcal{R}'$. Next, it repeats the same for the second trace, resulting in r_5 and r_6 added to $\mathcal{R}'$ and $\phi'(r_5) = \phi'(r_6) = \phi_2$. Finally, TAILORREQUEST returns an empty set for the third trace; hence, $\mathcal{R}'$ remains unchanged. The result is the output illustrated in Figure 7.3b.

7.5. Model-based Tailoring

Instead of performing tailoring on the collected traces, the second algorithm targets the workload model $\mathcal{W}$. While there are several workload modeling formalisms in the literature (Calzarossa et al., 2016; Draheim et al., 2006), we chose to use the WESSBAS-DSL (Vögele et al., 2018), which models $\mathcal{W}$ as multiple Markov-chain-based behavior models W_j with a relative frequency $f(W_j)$. In addition to the Markov chains, the DSL allows modeling so-called guards and actions, which we exclude from this work. The algorithm modifies each behavior model individually and replaces or removes states that are not in the set of targeted endpoints $\mathcal{E}$. Hence, it adjusts step ③ of the load test extraction process (Figure 7.2), while the others remain unchanged.

In the following, we describe the inputs and outputs of the algorithm, two helper algorithms, and the algorithm itself.

7.5.1. Input and Output

The model-based tailoring algorithm takes the following inputs:

- the workload model $\mathcal{W}$ as generated by the original extraction process, consisting of one or several Markov chains W_1 to W_q, with $W_j = (\Sigma_j, \varepsilon_j, p_j, \Delta_j, S_j)$,
- the recorded trace logs $\mathcal{T}$,
- the endpoints $\mathcal{E}$ of the targeted Services $\mathcal{M}$.

The algorithm processes the inputs and generates a modified workload model $\mathcal{W}' = (\{W_1', \dots, W_q'\}, f)$. The Markov chains W_j' will be modified while the workload mix f remains unchanged. Precisely, the algorithm replaces states with sub-Markov chains that model the request order and timings caused by the user-faced states or removes states that do not entail any further requests. In the following, we define this behavior in three postconditions the algorithm should meet. Again, we will use them to validate the proposed algorithm in Chapter 13.

For a formal specification of the postconditions, we consider each $W'_j = (\Sigma'_j, \varepsilon'_j, p'_j, \Delta'_j, S'_j)$ individually in relation to the original W_j. We define the set of traces corresponding to the sessions of the Markov chain:

$$T_j := \left\{\tau \in \mathcal{T} \mid \ \exists s \in S_j : r_\tau \in s\right\}$$

Besides, we define the *trace session*, consisting of the requests submitted within one trace τ. Even though there is a similarity to the session definition for the log-based algorithm (see Equation (7.1) in Section 7.4.1), a trace session does not consider the session ID ϕ. Instead, it describes the requests the application did internally for calculating the response to the user request:

$$s_\tau := \{r \in \tau \mid \ \varepsilon(r) \in \mathcal{E} \land P_{\mathcal{E}}(\tau, r)\}$$

With these definitions, we can specify the first postcondition $A^{(\text{model})}_{\text{states}}$ concerning the states Σ'_j of the tailored Markov chain. We require that every request of every s_τ has a corresponding state:

$$A^{(\text{model})}_{\text{states}} \equiv \forall \tau \in T_j \ \forall r \in s_\tau \ \exists \tilde{e} \in \Sigma'_j : \varepsilon(r) = \varepsilon'_j(\tilde{e}) \tag{7.3}$$

With a second postcondition $A^{(\text{model})}_{\text{prob}}$, we describe the proper transition probabilities defined by p'_j. With the algorithm, we will replace several states of Σ_j with a set of new states. For describing the relation between the new states and the original state — which uniquely maps to one endpoint —, we define the *root endpoint* as a function $\text{root}_j : \Sigma_j \to \mathcal{E}^{(\text{orig})}$:

$$\text{root}_j(\tilde{e}) = e \longleftrightarrow \big(\exists \tau \in T_j \ \exists r \in s_\tau : \varepsilon(\tau) = e \land \varepsilon(r) = \varepsilon_j(\tilde{e})\big)$$

Notably, root_j implies that for each request in $\mathcal{E}$, there can be multiple states in Σ'_j. sub_j summarizes these states, i.e., it is the inverse of root_j:

$$\tilde{e} \in \text{sub}_j(e) \longleftrightarrow \exists e \in \mathcal{E}^{(\text{orig})} : e = \text{root}_j(\tilde{e})$$

We base the postcondition on the probability to reach one state from another, regardless of the number of steps required. That is, the tailored Markov chain must submit the same requests as the untailored one does in combination with the services calling each other. Remarkably, we cannot compare individual transitions, due to potential state removals.

Let $p_j^{\infty}(e_1, e_2)$ be the probability to reach e_2 from e_1 in any number of steps and $p_j'^{\infty}(\tilde{e}_1, \tilde{e}_2)$ the corresponding function for the tailored Markov chain. We state that the corresponding replacing states should preserve the $p_j^{\infty}(e_1, e_2)$ in the original Markov chain. That means for every state $\tilde{e}_1$ that replaces e_1, the probability to reach any of the replacing states of e_2 should be equal to $p_j^{\infty}(e_1, e_2)$. Let $p_j'^{\infty}(e, X)$ be the probability to reach any state in a set X from e. Then, the postcondition is the following:

$$
\begin{aligned}
A_{\text{prob}}^{(\text{model})} \equiv \forall \tilde{e}_1 \in \Sigma_j' \; \forall e_2 \in \mathcal{E}^{(\text{orig})} : \big(\text{sub}_j(e_2) \neq \emptyset \wedge \tilde{e}_1 \notin \text{sub}_j(e_2)\big) \\
\rightarrow \big(p_j'^{\infty}(\tilde{e}_1, \text{sub}_j(e_2)) = p_j^{\infty}(\text{root}_j(\tilde{e}_1), e_2)\big)
\end{aligned}
\tag{7.4}
$$

Finally, we define a postcondition $A_{\text{time}}^{(\text{model})}$ describing the timing behavior of the tailored Markov chain. Because the algorithm changes the structure of the Markov chain, we cannot compare individual transitions between the untailored and tailored ones. Instead, we focus on the highest possible granularity. We define the time needed on average to execute the whole Markov chain as $\text{time}_j(I, \$)$ (untailored chain) and $\text{time}_j'(I, \$)$ (tailored chain). The execution includes the think times and the time waiting for the response of the application after submitting a request. The execution time should remain the same after the tailoring:

$$
A_{\text{time}}^{(\text{model})} \equiv \text{time}_j'(I, \$) = \text{time}_j(I, \$) \tag{7.5}
$$

7.5.2. Example

For illustrating the tailoring of a Markov chain, we use an example. Figure 7.4 shows a Markov chain before and after the tailoring. The original Markov chain W has five states, e_1 to e_5, plus the initial and final states. The states e_1

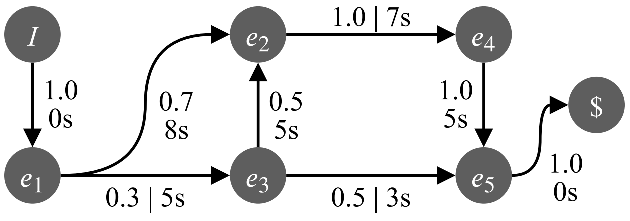
to e_5 correspond to one endpoint each; hence, we can assume $\Sigma_j = \mathcal{E}^{(orig)}$ and $\varepsilon_j = \text{id}$. For illustration purposes, we assume each request made in one of the states to take one second and each transition think time to have a variance of zero. W' should target a different but overlapping set of endpoints, which is e_1 plus e_5 to e_7. That is, we need to replace or remove states e_2 to e_4.

For e_2, we assume there are two endpoints e_6 and e_7 that the application requests as a reaction to a request to e_2—i.e., $\text{sub}_j(e_2) = \{e_{6,1}, e_7\}$ with $\varepsilon'_j(e_{6,1}) = e_6$ and $\varepsilon'_j(e_7) = e_7$. The differentiation between two states $e_{6,\cdot}$ is required because they request the same endpoint (see below). We need to replace e_2 with the aggregate request behavior, as illustrated in Figure 7.4b. In doing so, we have to divide the transitions entering into e_2 into the states of $\text{sub}_j(e_2)$. Here, e_6 is called more frequently than e_7; hence, $p'_j(e_1, e_6) > p'_j(e_1, e_7)$. Furthermore, we need to adjust the think times. For example, the transition from e_7 to e_4 respects that the replacement takes less time than the original e_2. Thus, the think time is 0.2 seconds longer.

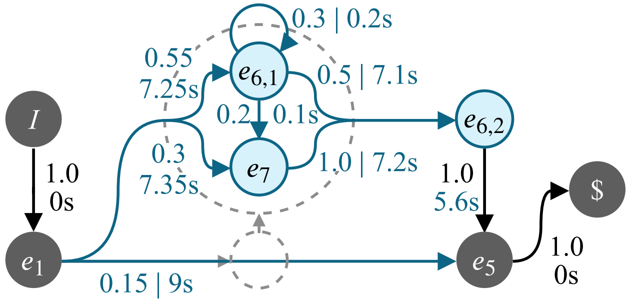
For e_3, the application does not request any further endpoints in $\mathcal{E}$. Therefore, we remove the state. As a consequence, we need to concatenate the incoming and outgoing transitions. For instance, we add a new transition from e_1 to e_5 with a transition probability of $p'_j(e_1, e_3) * p'_j(e_3, e_5) = 0.15$. Duplicate transitions resulting from the concatenation—such as from e_1 to e_2—, we need to merge by adding the transition probabilities.

Furthermore, we need to compute the correct think times. For (e_1, e_5), the think time is the sum of the two former transitions and the duration of the state $\delta_3 \sim \mathcal{N}(1,0)$: $\Delta'_j(e_1, e_5) = \Delta'_j(e_1, e_3) * \delta_3 * \Delta'_j(e_3, e_5) \sim \mathcal{N}(9,0)$. For (e_1, e_2), the merge of two parallel transitions requires modeling an “either-or” think time. Notably, that is not possible when sticking to normal distributions. Therefore, we will attempt the best solution using normal distributions, which preserves at least the correct mean.

The final adjustment of the original Markov chain is the replacement of state e_4. We assume it needs to be replaced by a single request to e_6. Hence, the structure remains unchanged, as a state $e_{6,2}$ with $\varepsilon'_j(e_{6,2}) = e_6$ replaces e_4. Only the think times are affected; in this case, the transition to e_5 takes 0.6 seconds longer, due to the shorter duration of the replacement.



(a) Original Markov chain W_j.



(b) Tailored Markov chain W'_j.

Figure 7.4.: Example of tailoring a Markov chain W_j to endpoints $\mathcal{E} = \{e_1, e_5, e_6, e_7\}$. The transition are labeled with the transition probability and the average think time.

The resulting Markov chain conforms to the postconditions formulated in the previous section. $A^{(\text{model})}_{\text{states}}$ holds because there is at least one state per endpoint. For e_6, there are even two states, because the replacements of both e_2 and e_4 request the endpoint.

$A^{(\text{model})}_{\text{prob}}$ holds due to the probability adjustments we did. As an example, we consider the reachability of e_2 or its replacement from e_1. In the original Markov chain, there are two ways to reach e_2 from e_1: via the direct transition or e_3. In sum, the probability of reaching e_2 is 0.85. In the tailored Markov chain, there are also two ways to reach the replacement of e_2 from e_1: the direct transitions to $e_{6,1}$ and e_7. In sum, the probability of reaching $\text{sub}_j(e_2)$ from e_1 is 0.85, too.

Finally, we preserved the think times, as stated by $A^{(\text{model})}_{\text{time}}$, by merging the think times of merged or concatenated transitions, also considering the duration of states removed or replaced by shorter-lasting ones.

In the following, we introduce three algorithms for the operations illustrated above: state removals, state replacements with sub-Markov chains, and orchestration of the two.

7.5.3. Algorithms for Modifying Markov States

We introduce two helper algorithms for the model-based tailoring, namely for removing and replacing individual states in a Markov chain.

Algorithm 7.3 Remove state e with normally distributed duration δ from Markov chain W_j*

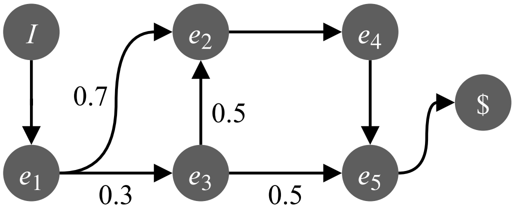
```
1:  function REMOVESTATE($W_j = (\Sigma_j, p_j, \Delta_j, \mathcal{S}_j), e, \delta$)
2:      $\alpha \leftarrow \frac{1}{1-p_j(e,e)} - 1$                ▷ geometric series (Hildebrandt, 2006)
3:      for all $e' \in \Sigma_j$ do
4:          $p_j(e,e') \leftarrow \frac{p_j(e,e')}{1-p_j(e,e)}$
5:          $\Delta_j(e,e') \leftarrow \Delta_j(e,e') * \alpha \cdot \big(\Delta_j(e,e) * \delta\big)$          ▷ correction†
6:      end for
7:      for all $e', e'' \in \Sigma_j$ do
8:          $p' \leftarrow p_j(e',e'') + p_j(e',e) \cdot p_j(e,e'')$
9:          $\Delta_j(e',e'') \leftarrow \left[\frac{p_j(e',e'')}{p'} \cdot \Delta_j(e',e'')\right]$
                $* \left[\frac{p_j(e',e) \cdot p_j(e,e'')}{p'} \cdot \big(\Delta_j(e',e) * \Delta_j(e,e'') * \delta\big)\right]$
10:         $p_j(e',e'') \leftarrow p'$
11:     end for
12:     $\Sigma_j \leftarrow \Sigma_j \setminus \{e\}$
13: end function
```

* Opposed to H. Schulz et al. (2019a), we use the revised notation of states and endpoints (see Section 7.2).

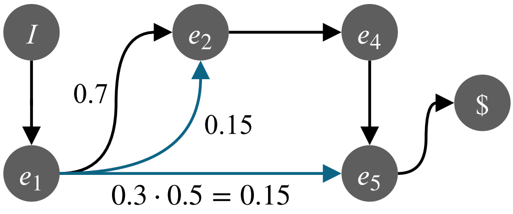
† In the previously published version, we missed including the duration δ, which we fix here.

7.5.3.1. Remove State

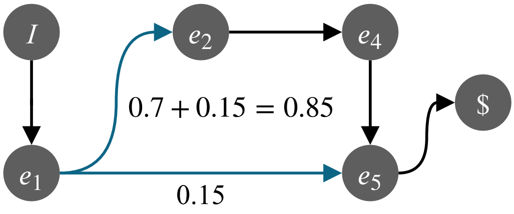
Removing a state e with duration distribution δ from a Markov chain W_j is implemented in Algorithm 7.3 and illustrated in Figure 7.5. In this example, we reuse the Markov chain from Figure 7.4, and we want to remove state e_3. In a first step, the algorithm calculates the expected number of steps the Markov chain loops in the state e — given that such a cycle exists —, based on the geometric series $\sum_{i=1}^{\infty} p_j(e,e)$ (line 2). Because we will remove e, we need to ensure that the Markov chain preserves the time spent in the loop.



(a) Input: Markov chain W_j

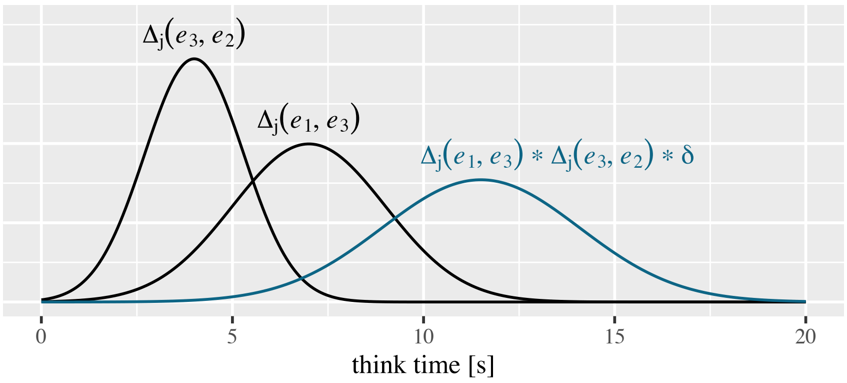


(b) Step 1: concatenate transitions via e_3

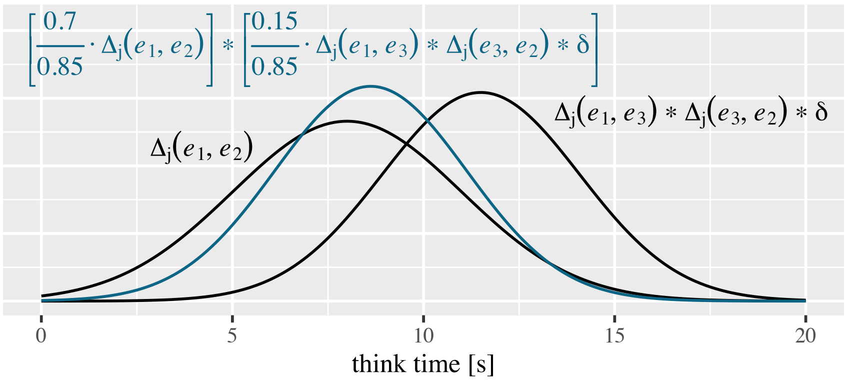


(c) Step 2: merge duplicate transitions

Figure 7.5.: Illustration of REMOVESTATE(W_j, e_3, δ) (Algorithm 7.3).



(a) Step 1: concatenate (e_1, e_3) and (e_3, e_2)



(b) Step 2: merge the two transitions (e_1, e_2)

Figure 7.6.: Illustration of the think time calculation in $\textsc{removeState}(W_j, e_3, \delta \sim \mathcal{N}(0.5, 1))$ (Algorithm 7.3).

For that, we remove the cycle by appending the think time plus the duration of e to all outgoing transitions from e and by normalizing the probabilities such that they sum to 1 (lines 3-6).

Then, we concatenate all incoming and outgoing transitions to/from e (lines 7-11). We do this in two logical steps. First, we concatenate the transitions, e.g., (e_1, e_3) and (e_3, e_2) to (e_1, e_2), and (e_1, e_3) and (e_3, e_5) to (e_1, e_5) (Figure 7.5b). The think times we convolve accordingly, as illustrated in Figure 7.6a, furthermore including the original state's duration δ. In the second step, we merge potential duplicate transitions, e.g., between e_1 and e_2 (Figure 7.5c). The think times we convolve again, weighted by the relative transition probabilities (Figure 7.6b). Please note that such a resulting think time deviation is not a stochastically valid junction of the original think times. However, as the proper junction is not a normal distribution, but the workload model requires it, this is the closest we can achieve. At least, the mean think time will be correct.

7.5.3.2. Replace State

When a state is to be replaced, a new (sub-) Markov chain is to be inserted instead of the original state. The replacement is implemented in Algorithm 7.4 and illustrated in Figure 7.7 at the example of replacing state e_2 of our example Markov chain (see Figure 7.4a). The algorithm takes as input the Markov chain W_j, the state e to be replaced, and the tailored request logs $\mathcal{R}'$ representing the request behavior that should replace e. The algorithm interprets the requests in $\mathcal{R}'$ as multiple sessions and, besides, assumes it contains placeholders I' and $\$'$, indicating the start and end of each session. These placeholders correspond to the start and end of the root request. Hence, it allows the algorithm to respect the difference in the duration of the replacement and the original state.

First, the algorithm groups the request logs into session logs — which correspond to the s_τ (see Section 7.5.1) — and aggregates them into a new Markov chain W' (lines 2 and 3). For the aggregation, we use the existent means of the WESSBAS approach. As an example, W_j' can appear as

Algorithm 7.4 Replace state e of Markov chain W_j with a Markov chain derived from tailored requests $\mathcal{R}'$*

```
function REPLACESTATE($W_j = (\Sigma_j, p_j, \Delta_j, \mathcal{S}_j), e, \mathcal{R}'$)
    $\mathcal{S}' \leftarrow$ GROUPTOSESSIONS($\mathcal{R}'$)
    $W' = (\Sigma', \cdot, \cdot, \cdot) \leftarrow$ AGGREGATE($\mathcal{S}'$)
    $\Sigma_j \leftarrow \Sigma_j \cup \Sigma'$
    for all $e' \in \Sigma_j$ do
        $p_j(e', I') \leftarrow p_j(e', e)$
        $\Delta_j(e', I') \leftarrow \Delta_j(e', e)$
    end for
    for all $e' \in \Sigma_j$ do
        $p_j(\$', e') \leftarrow p_j(e, e')$
        $\Delta_j(\$', e') \leftarrow \Delta_j(e, e')$
    end for
    $\Sigma_j \leftarrow \Sigma_j \setminus \{e\}$
    REMOVESTATE($W_j, I', 0$)
    REMOVESTATE($W_j, \$', 0$)
end function
```

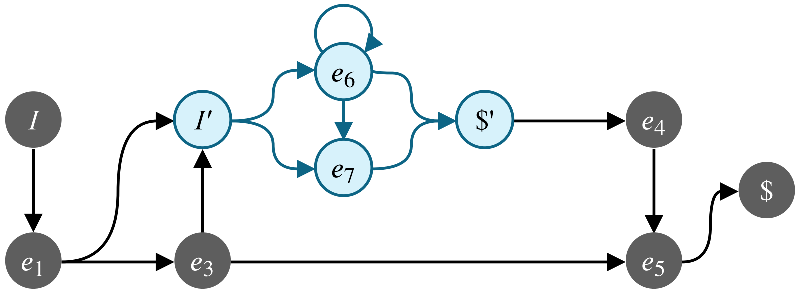
* Opposed to H. Schulz et al. (2019a), we use the revised notation of states and endpoints (see Section 7.2).

illustrated in Figure 7.7a. Here, it contains two states e_6 and e_7, targeting corresponding endpoints, plus the initial and final states.

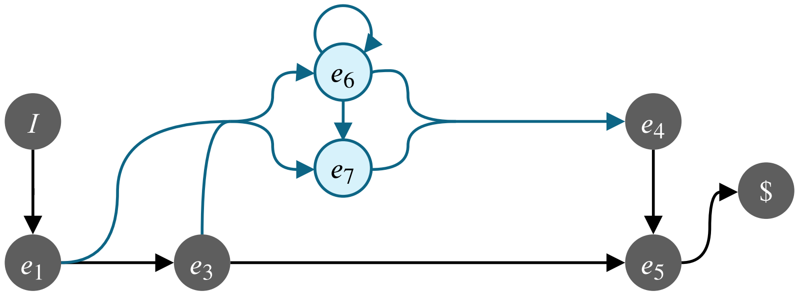
Then, this chain replaces e in two steps. In the first step, we replace e (e_2 in the example) with the new chain by changing the target and source states of the transitions of e (lines 4-8; Figure 7.7a). In the second step, we remove the artificially inserted states I' and $\$'$ using Algorithm 7.3 (lines 13-15; Figure 7.7b). Because I' and $\$'$ mark the start and end of the original state, the incoming and outgoing transitions of the new chain will contain the duration of the original request as think times.

7.5.4. Model-based Tailoring Algorithm

In this section, we present the actual tailoring algorithm, which utilizes the two algorithms we introduced before. Algorithm 7.5 implements the



(a) Step 1: replace e_2 with Markov chain



(b) Step 2: remove I' and $\$'$ using REMOVESTATE

Figure 7.7.: Illustration of REPLACESTATE($W_j, e_2, \mathcal{R}'$) (Algorithm 7.4).

model-based tailoring. It takes the workload model $\mathcal{W}$, the collected traces $\mathcal{T}$, and the set of target endpoints $\mathcal{E}$ as input and modifies each Markov chain W_j of $\mathcal{W}$. Based on the traces and the session logs $\mathcal{S}_j$ corresponding to the Markov chain, it replaces all states that are not contained in $\mathcal{E}$ by new endpoints from $\mathcal{E}$. A replacement needs to model the application's control flow that is caused by a request of the original state.

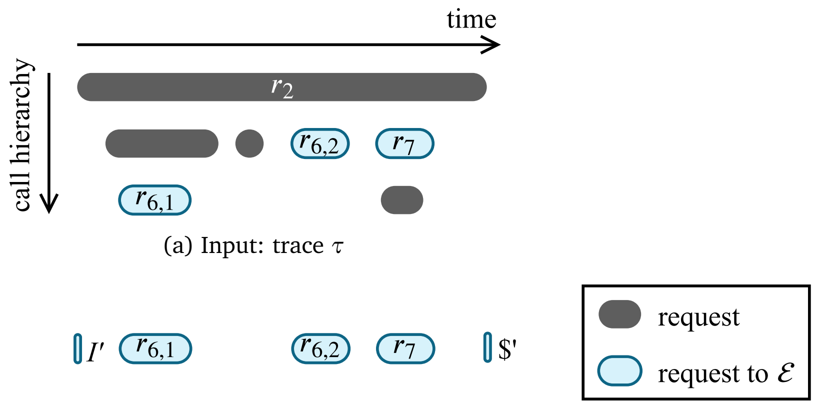
For that, the algorithm iterates over all such states (line 4) and collects all traces from $\mathcal{T}$ whose root request is contained in a session $\mathcal{S}_j$ (line 5). Then, it sets the session IDs of these traces to unique values (line 6) and reuses Algorithm 7.2 for extracting request logs tailored to $\mathcal{E}$, as illustrated in Figure 7.8. Given a trace with root request r_2 is processed and there are nested requests $r_{6,1}, r_{6,2}, r_7$ to endpoints in $\mathcal{E}$, the resulting request logs will consist of $\{r_{6,1}, r_{6,2}, r_7\}$. Because of the changed session IDs, $\mathcal{R}'$ represents the control flow at the endpoints in $\mathcal{E}$ from the perspective of the original state rather than an end user's perspective.

Depending on $\mathcal{R}'$, the original state is either removed if $\mathcal{R}'$ does not cause any requests on $\mathcal{E}$ (line 9), or replaced by a Markov chain extracted from $\mathcal{R}'$ (line 13). In the former case, the aggregated duration of the requests to the original state — represented by a normal distribution $\delta(\mathcal{T}')$ — is passed as a parameter for preserving the delay introduced by the state. In the latter case, placeholder requests I' and $\$'$ are added (lines 11 and 12; Figure 7.8b), which will ensure that the new Markov chain will take the same time as the original state does. Both are important for preserving the overall timing of W_j.

Algorithm 7.5 Tailor workload model $\mathcal{W}$ to endpoints $\mathcal{E}$ using traces $\mathcal{T}$*

```
1:  **function** tailorWorkloadModel($\mathcal{W}, \mathcal{T}, \mathcal{E}$)
2:      $\mathcal{W}' \leftarrow \mathcal{W}$
3:      **for all** $W_j' = (\Sigma_j', p_j', \Delta_j', \mathcal{S}_j') \in \mathcal{W}'$ **do**
4:          **for all** $e \in \Sigma_j, \varepsilon_j(e) \notin \mathcal{E}$ **do**
5:              $\mathcal{T}' \leftarrow \{(r_\tau, R_\tau, C_\tau) \in \mathcal{T} \mid \exists s \in \mathcal{S}_j : r_\tau \in s \wedge \varepsilon(r_\tau) = \varepsilon_j(e)\}$
6:              $\phi(\mathcal{T}') \leftarrow \{1, \dots, |\mathcal{T}'|\}$
7:              $\mathcal{R}' \leftarrow$ tailorRequestLogs($\mathcal{T}', \phi, \mathcal{E}$)
8:              **if** $\mathcal{R}' = \emptyset$ **then**
9:                  removeState($W_j', e, \delta(\mathcal{T}')$)
10:             **else**
11:                 $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{(I', \phi(\tau), t(\tau), 0) \mid \tau \in \mathcal{T}'\}$
12:                 $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{(\$', \phi(\tau), t(\tau) + \delta(\tau), 0) \mid \tau \in \mathcal{T}'\}$
13:                 replaceState($W_j', e, \mathcal{R}'$)
14:             **end if**
15:         **end for**
16:     **end for**
17:     **return** $\mathcal{W}'$
18: **end function**
```

* Opposed to H. Schulz et al. (2019a), the algorithm creates a new workload model $\mathcal{W}'$ instead of changing $\mathcal{W}$. Furthermore, we use the revised notation of states and endpoints (see Section 7.2).



(a) Input: trace τ

(b) Output: request logs with I' and $\$'$

(c) Legend

Figure 7.8.: Extracting tailored request logs from a trace in Algorithm 7.5.

7.6. Integration with Automated Parameterization

In the previous chapter, we introduced an approach to the automated parameterization of load tests using Input Data and Properties Annotations (IDPAs). The algorithms we proposed in this chapter integrate with IDPAs, as illustrated in Figure 7.9. We consider multiple teams each developing a service of the application. To each team, we assign a name tag for unique identification. Furthermore, each team has its IDPA, identified by the same name.

The IDPA of a team consists of an application and an annotation model. The application model describes the endpoints of the team's service. Hence, it describes the set $\mathcal{E}$ of endpoints to which the load test should be tailored. The annotation model holds the parameterizations to be applied to the load test. Following DevOps practices (Bass et al., 2015), each team has to specify its parameterizaion individually.

For generating a load test, e.g., inside a CI/CD pipeline, all teams can use the same trace logs. However, they will use different IDPAs, resulting in load tests appropriately tailored to the respective service. In the case

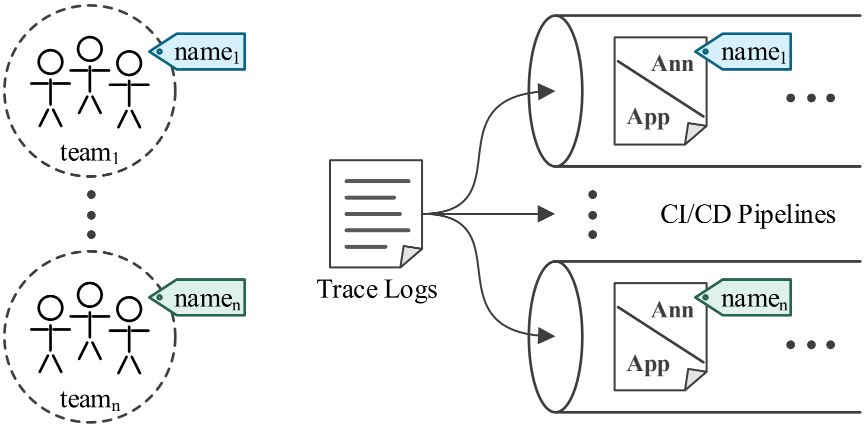


Figure 7.9.: Integration of tailoring to services with IDPAs for multiple teams.

that a load test should include multiple services as an integration test, we can use the respective IDPAs, merge the application models, and apply the parameterizations defined in the annotations subsequently. The name tags allow the teams to specify which IDPAs and, thus, tailoring to use only by the names of the services involved.

7.7. Summary

In this chapter, we introduced an approach for tailoring load tests to a specified set of (micro) services. In doing so, we addressed RQ2: *How can representative load tests be tailored to specific services of a session-based application?*

We introduced two algorithms — log-based and model-based tailoring —, which modify certain artifacts of the load test extraction process. As a result, the load tests generated by the modified pipeline directly target the services to be tested. Hence, DevOps teams do not need to integrate calling services when applying representative load testing.

Tailored load tests complement existing approaches. In combination with performance stubs (Baltas and Field, 2012; Becker et al., 2008; Field et al., 2018; Versteeg et al., 2016), they allow isolating a service completely during the load test. Furthermore, our approach to the automated parameterization of load tests integrates with the tailoring approach.

For each tailoring algorithm, we formally specified requirements to the artifacts produced. In Chapter 13, we will verify the algorithms to fulfill these requirements. Furthermore, we will compare the representativeness and other quality attributes of the load tests tailored by the two algorithms in an experimental study.

In the next chapter, we introduce tailoring of load tests in a different dimension, namely the workload context. This will increase the suitability of the tests and the level of automation. Still, context and service-tailoring can be combined. In Chapter 9, we present a load test description language using the capabilities of the service-tailoring approach.

Chapter 8

Tailoring Load Tests to Workload Contexts

The workloads of modern session-based applications undergo a significant variation (Herbst et al., 2013), which needs to be considered when load-testing the application. Known from detecting contextual anomalies (Chandola et al., 2009), contexts such as special offers, incidents, or weather conditions can influence the workload, resulting in different workload scenarios. For instance, while the normal workload of a webshop might comprise 1000 concurrent users of a certain mix, a special offer could cause a spike to up to 5000 users of a different mix. As contexts do not only influence the number of users but also the workload mix, there is no workload scenario we can consider to be most demanding. Consequently, when load-testing the application, we need to consider all scenarios that have already been observed or might occur in the future. However, continuous software engineering (CSE) has fast release cycles (Humble and Farley, 2010), which do not allow for such extensive and time-consuming testing.

Existing approaches (Barros et al., 2007; Cai et al., 2007; Krishnamurthy et al., 2006; Lutteroth and Weber, 2008; Menascé and Almeida, 2002; Ruffo et al., 2004; Vögele et al., 2018) can extract a load test from recorded user sessions representing a workload scenario that occurred in the past. They generate a workload model — e.g., based on Markov chains — that represents the behavior of different user groups with a particular mix. Hence, they reconstruct the past workload scenario. However, past scenarios are not sufficient for preparing for future scenarios. As most workloads follow a global trend and seasonal variations (Herbst et al., 2013), they need to be integrated into the load test. For that, approaches for time series modeling and forecasting (Bauer et al., 2020; Herbst et al., 2013; von Kistowski et al., 2014b; Taylor and Letham, 2018) could be used, which, however, mainly focus on the overall intensity, disregarding a varying workload mix. Also, they lack support for selecting the relevant scenarios and incorporating qualitative forecasting (Menascé and Almeida, 2002), e.g., for estimating unseen future scenarios. Finally, CSE requires automation.

Therefore, we target an approach that automatically generates load tests tailored to the contexts a user specifies. In doing so, they can express the scenarios relevant to them and save the time needed for testing less relevant scenarios. As an example, the operator of a webshop will be interested in the Black Friday spike in early November, which is, however, less relevant in December. Also, the user should be able to add external knowledge as a qualitative forecast, e.g., planned changes to the platform. Our approach should then automatically generate a load test tailored to the user's specification. Hence, we address RQ3: *How can representative load tests automatically be tailored to the contexts of a session-based workload?*

For this purpose, we extend the WESSBAS approach (Vögele et al., 2018) for learning user groups and their mix incrementally and leverage existing time series forecasting approaches for predicting the future behavior of these groups. Furthermore, we introduce the Load Test Context-tailoring Language (LCtL) for describing a workload scenario based on its context. LCtL instances serve as input to the forecasting approach, specify how to extract a workload scenario from a forecast or past data, and allow changing the scenario for qualitative forecasting. For integration with CSE, the generation of a context-tailored load test based on an LCtL description is fully automated.

The remainder of this chapter is structured as follows. Section 8.1 defines the fundamental notions used in this chapter. As further motivation, Section 8.2 provides examples for workload-influencing contexts. In Section 8.3, we present the suggested context-tailoring process, whose individual steps we detail in the subsequent sections. First, in Section 8.4, we introduce an approach for learning a workload model incrementally and enriching it with contexts. Section 8.5 presents the LCtL. In Section 8.6, we describe the realization of the LCtL concepts. Section 8.7 depicts the integration with the approaches introduced in Chapters 6 and 7. We conclude the chapter with a summary in Section 8.8.

This chapter is an extended version of Section 4 of the manuscript below. Apart from minor details for comprehensibility, we have added a second incremental clustering algorithm in Section 8.4.1 and describe the integration with the remainder of our approach in Section 8.7.

- H. Schulz, D. Okanović, A. van Hoorn, and P. Tůma (2021). "Context-tailored Workload Model Generation for Continuous Representative Load Testing." In: *Proceedings of the 12th ACM/SPEC International Conference on Performance Engineering (ICPE 2021)*. To appear. ACM

8.1. Definitions

Before presenting and classifying examples of workload contexts and the corresponding workload scenarios, we detail the definitions of these two notions.

Definition 8.1 (Workload Context)
A workload context is a set of workload context facets. *A workload context facet is a self-contained circumstance present to a significant number of users of the application. It has a well-defined state at each point in time. Examples are special sales offers (which can be present or not) or weather conditions (whose state set is continuous).*

Context facets that have only two states (present or not) we also denote to *occur* or not. Notably, also contexts of such facets that all do not occur at a certain point in time can exist. The workload at this point in time could be considered to be normal.

A context-tailored load test aims at replaying a particular workload scenario, which we define as below. Naturally, the context influences a workload scenario.

Definition 8.2 (Workload Scenario)
A workload scenario is a workload segment of a fixed length with specific characteristics to be replayed in a load test, e.g., the constant workload of a specific mix or a workload spike. Workload scenarios can have occurred in the past, be expected in the future, or be hypothetical, e.g., for what-if analyses.

8.2. Examples of Workload-influencing Contexts

The Web collects several examples of contexts that influenced an application's workload. Particularly, influences that caused a high workload, which, in turn, lead to downtimes are reported in blog posts and news articles. Below, we provide a collection of such contexts, including seven different context facets (see highlights in the text).

Soper (2012): high workload due to *special offers* (Boxing Day)

Billington (2014): high workload and resulting downtime due to *special incident* (prominent posting)

WeatherAds (2014): varying workloads due to *weather condition*

Weise (2016): high workload and resulting downtime due to *release of new product* (TV series) and *weather condition* (rainy)

Bradley Web Group (2016): varying workloads depending on the *weekday and holidays*

Godlewski (2017): high workload and resulting downtime due to *release of new product* (sneakers) and *special offers* (Black Friday)

Popa (2018): high workload due to *special incident* (acquisition of competitor)

Mills (2018): high workload and resulting downtime due to *special offers* (Prime Day)

H. Schulz et al. (2019c) (1): increased workload due to planned *changes to the platform* (new devices added)[1]

H. Schulz et al. (2019c) (2): workload spike due to *special incident* (recovery from message endpoint outage)[1]

Heckmann (2020): suddenly decreased workload due to *current event* (new year's countdown)

Wikipedia (2020): increased focus on a particular subject influenced by *current events*

We classify the context facet examples for developing a meta-model of contexts and the LCtL. The classification bases upon two attributes: the predictability of the facet's state and the (quantitative) predictability of the influence on the workload. Table 8.1 summarizes the classification and additionally highlights the regularity of state changes.

The largest category is context facets that recur, i.e., whose state and influence are well predictable. These context facets comprise special offers — as reported by Soper, Weise, and Mills —, which a company plans. Some special offers even belong to world-wide sales events such as Black Friday. Furthermore, the related facet type of product releases, as reported by Weise and Godlewski, falls into this category. Finally, there are public holidays (Bradley Web Group, 2016) and public events (Heckmann, 2020; Wikipedia, 2020). WeatherAds and Weise report about continuous context facets, namely weather conditions. These are related to the first category, but with the limitation that the future weather conditions rely on a forecast with a limited range and accuracy.

Significantly different categories are irregular and singleton context facets. Irregular means that future states are unknown in advance, but the former state has been observed and can be used to predict the influence. An example

[1]These examples can also be found in the evaluation presented in Chapter 15.

Table 8.1.: Classification of the Collected Workload Context Facets

state	influence	
	predictable	**unpredictable**
predictable	***recurring*** Soper (2012) Weise (2016) Bradley Web Group (2016) Godlewski (2017) Mills (2018) Heckmann (2020) Wikipedia (2020)	***singleton*** H. Schulz et al. (2019c) (1)
semi-predictable	***continuous*** WeatherAds (2014) Weise (2016)	
unpredictable	***irregular*** H. Schulz et al. (2019c) (2)	***singleton*** Billington (2014) Popa (2018)

is from our work (H. Schulz et al., 2019c (2)), where outage recoveries occurred and will occur on a random basis.

The influence of singleton context facets (H. Schulz et al., 2019c (1); Billington, 2014; Popa, 2018) cannot be quantitatively predicted based on past states. As the name indicates, these facets occur only once. Here, qualitative forecasting methods are required for predicting the influence. Especially in the case of known future occurrences, e.g., a planned change to the platform (H. Schulz et al., 2019c (1)), qualitative forecasting can be applied.

Besides the differences in predictability, contexts also differ in the type of workload scenario. While some cause increased but generally steady workload — e.g., Soper (2012) —, others entail a workload spike — e.g., H. Schulz et al. (2019c) (1). Notably, contexts do not only influence the workload intensity, but also the mix. For instance, Wikipedia reports changing focus on different subjects depending on current events. As a consequence, there is no most-demanding workload we could use in a load test for covering all scenarios. Instead, we need load tests tailored to those contexts and scenarios that are currently relevant.

8.3. Context-tailoring Process Overview

Figure 8.1 illustrates the process of tailoring a load test to a context. It consists of two major parts: a continuously repeated one for filling a workload model repository (WMR) and one executed on demand for extracting load tests from the WMR. In this section, we provide an overview of the individual steps of both parts. In the following sections, we will detail the steps.

The continuously repeated part consists of two streams of data. First, session logs are clustered incrementally ①. Here, we base on the WESSBAS approach (Vögele et al., 2018) and extend the clustering algorithm. The result is a workload model consisting of multiple Markov-chain-based behavior models — each corresponding to one user group — and the intensity (number of users) per group over time. Concurrently to the session clustering, we store all relevant context facets into the WMR ②. They can be used for querying the stored workload model and intensity and for influencing a workload forecast. Hence, the WMR constitutes the single knowledge base for generating context-tailored load tests.

The part executed on demand always starts with a user's input, who describes a context-tailoring using the Load Test Context-tailoring Language (LCtL) ③. For integration into CSE, the processing of this description is fully automated. As a first step, we use the description for querying all relevant behavior models, intensities, and context data from the WMR ④. Also, the description can state to modify particular context facets for what-if analyses. The intensities and context are input to a time series forecasting approach ⑤. Because we forecast the intensities of each user group separately, we can

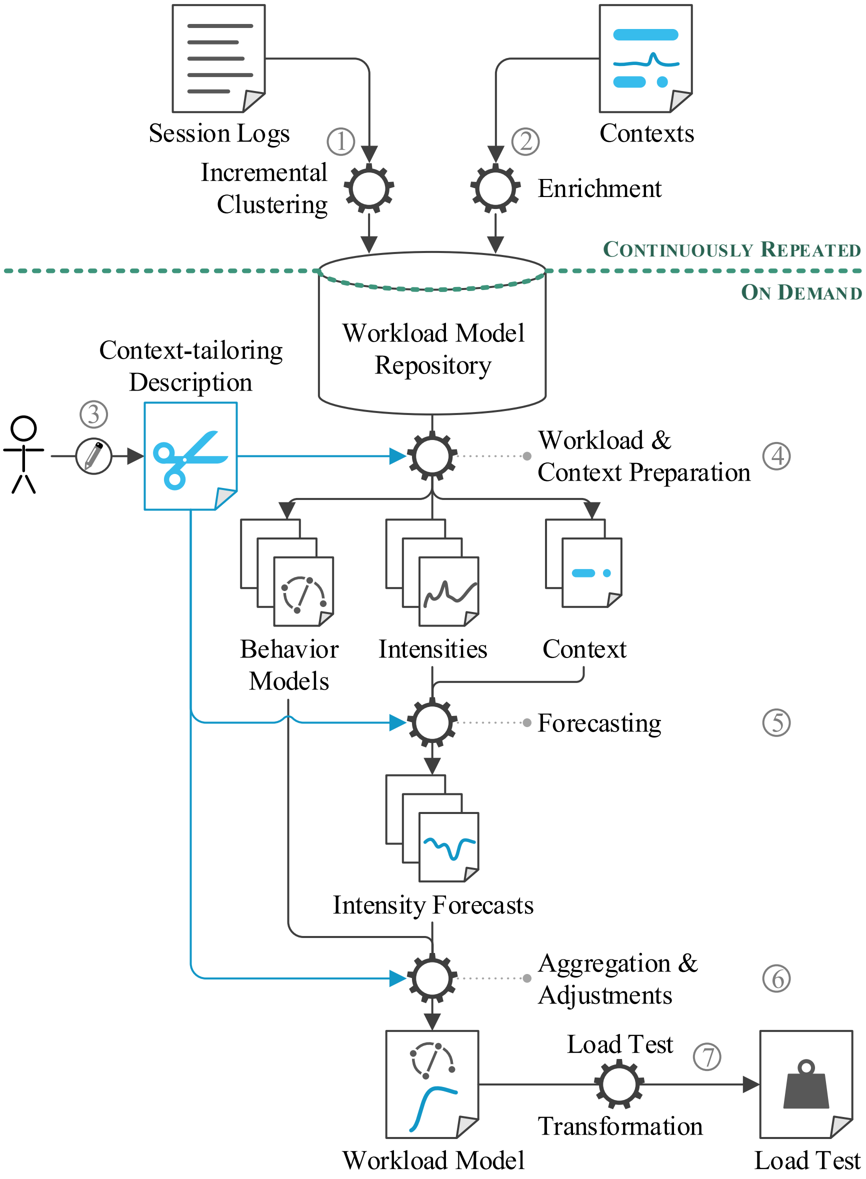


Figure 8.1.: Overview of the context-tailoring process.

respect varying workload mixes. Then, we apply an aggregation and an optional set of adjustments to the intensity forecasts and behavior models ⑥. The aggregation extracts individual workload scenarios from the forecast. The adjustments can be used to incorporate qualitative forecasts by changing the scenario, e.g., by increasing the intensity of a particular user group.

The result is a WESSBAS workload model with potentially varying intensities per user group. As a final step, our automated parameterization approach (Chapter 6) transforms the workload model into a load test ⑦.

8.4. Continuous Learning of the Workload Model

The learning of the workload model comprises the incremental clustering of the user sessions into behavior models and intensities and the enrichment with contexts. In Sections 8.4.1 and 8.4.2, we introduce our approaches to the two aspects.

8.4.1. Incremental Session Clustering

Existing approaches, such as WESSBAS (Vögele et al., 2018), extract a Markov-chain-based workload model from a set of sessions by clustering. For that, each of the sessions is encoded as a matrix consisting of the number of transitions between two endpoints (see Section 3.2.4) and treated as one data point. Each cluster is transformed into a Markov-chain-based behavior model representing a user group by determining a representative, e.g., the mean, and relativizing the absolute transition frequencies to probabilities. The workload model then comprises the behavior models and relative frequencies (mix) of them. The used clustering algorithms are k-means (Menascé et al., 1999) or its enhanced version X-means (Vögele et al., 2018).

However, these approaches are not designed for updating a once extracted workload model with newer sessions. Still, we require exactly this feature for building the workload knowledge base in the WMR. This requirement corresponds to RQ3.1: *How can we incrementally learn the workload models from observed user sessions for predicting future workload scenarios?*

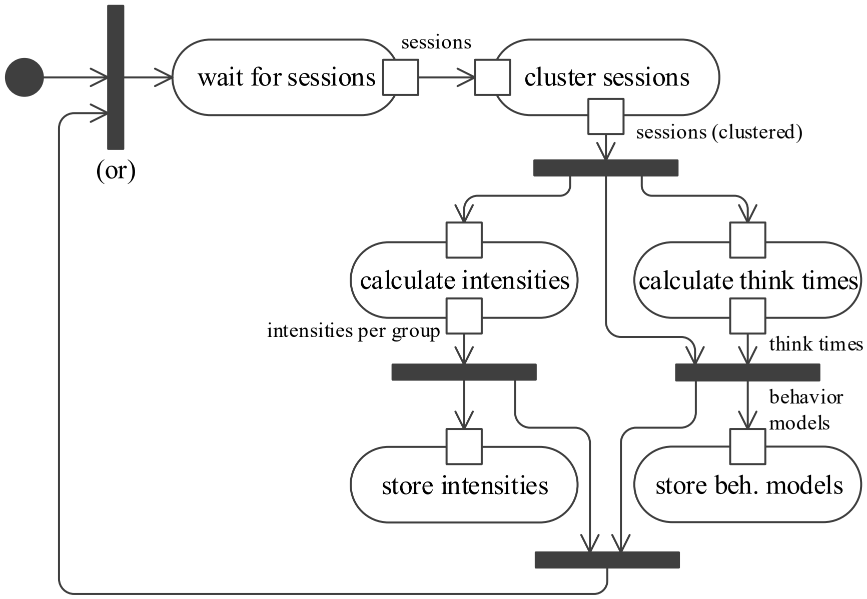


Figure 8.2.: Activity diagram of incremental session clustering.

We can extend the clustering for incremental updating by either modifying the clustering based on *k*-means or by choosing a different clustering algorithm. One such candidate is DBSCAN (Ester et al., 1996), which has a promising set of parameters different from *k*-means. Remarkably, neither the number of clusters nor a range needs to be defined. However, DBSCAN also entails several drawbacks over algorithms based on *k*-means. In a nutshell, DBSCAN can efficiently identify new clusters after several iterations, while *k*-means is better suited for selecting a cluster representative, which also eases incremental updating of existing clusters. After weighing all the arguments, we chose using *k*-means. In the following sections, we introduce two incremental clustering algorithms — one based on *k*-means and one using DBSCAN — and discuss the differences and our choice in detail.

Figure 8.2 illustrates a common framework we apply for both algorithms. We process the sessions in batches of a manageable size, e.g., one day or week. We wait until a new batch is ready and cluster it using one of the mentioned algorithms. Then, we calculate the think times per cluster and store the resulting behavior models — i.e., a representative per cluster with the calculated think times — into the WMR. Concurrently, we calculate the per-cluster intensities. Finally, we store the behavior models and intensities and wait for the next batch. Accounting for changes in the API, e.g., additions of endpoints, we re-encode the sessions before each clustering iteration.

In the following, we present the clustering algorithms and our algorithms for calculating the think times and intensities per session cluster, which are independent of the clustering algorithm.

8.4.1.1. Incremental Session Clustering based on k-means

k-means or related algorithms aim at finding a predefined number (or range of numbers) of clusters with *centroids* that minimize the *inertia*. The centroid μ_i of each cluster C_i is the mean of all data points S — sessions in our case — of the cluster:

$$\mu_i = \frac{1}{|C_i|} \sum_{s \in C_i} s$$

The inertia is the sum of squared distances between each session and the closest centroid:

$$\sum_{s \in S} \min_{C_i}(||s - \mu_i||^2)$$

An important characteristic of k-means is that all clusters are convex.

When clustering the sessions incrementally, i.e., updating the behavior models with new sessions, it is crucial that once classified sessions remain in the same cluster. As we need to calculate the per-cluster intensity, changing a session's cluster would require updating the already calculated intensity. As a consequence, the complexity of the clustering and intensity calculation would increase over time, until it will necessarily exceed an acceptable limit, e.g., new sessions arrive more frequently than they are clustered.

Algorithm 8.1 Initial session clustering using k-means++.

```
function initialKmeans($S$, $k$, $\eta$, $\alpha$)
    $d(s) \leftarrow \min_{s' \in S \setminus \{s\}} \{\|s - s'\|\}$ for $s \in S$
    $D \leftarrow q_{\frac{1+\alpha}{2}}(d(S)) + 1.5 \cdot \left(q_{\frac{1+\alpha}{2}}(d(S)) - q_{\frac{1-\alpha}{2}}(d(S))\right)$
    $C_{-1} \leftarrow \{s \in S \mid d(s) > D\}$
    $\{C_1, \dots, C_k\} \leftarrow$ kmeans++($\{s \in S \mid d(s) \leq D\}$, $k$, $\eta$)
    for $i \in \{-1, 1, \dots, k\}$ do
        $\mu_i \leftarrow \frac{1}{|C_i|} \sum_{s \in C_i} s$
        $r_i \leftarrow \max_{s \in C_i} \{\|s - \mu_i\|\}$
        $\hat{\mathbf{r}}_i \leftarrow$ transitionRadius($C_i$, $\mu_i$)
    end for
    return $\{C_{-1}, \dots, C_k\}$, $\{\mu_{-1}, \dots, \mu_k\}$, $\{r_{-1}, \dots, r_k\}$, $\{\hat{\mathbf{r}}_{-1}, \dots, \hat{\mathbf{r}}_k\}$
end function

function transitionRadius($C$, $\mu$)
    $\hat{\mathbf{r}} \leftarrow \infty^{\dim(\mu)}$
    for $j \in \{1, \dots, \dim(\mu)\}$, $\mu[j] > 0$ do
        $\hat{\mathbf{r}}[j] \leftarrow \max_{s \in C} \{|s[j] - \mu[j]|\}$
    end for
    return $\hat{\mathbf{r}}$
end function
```

Therefore, we process the sessions in batches and apply the framework from Figure 8.2. For the first clustering, we apply Algorithm 8.1 based on k-means++ (Arthur and Vassilvitskii, 2007) — an improved version of k-means reducing the risk of reaching local optima — for determining the initial clusters C_i, their centroids μ_i, and radiuses r_i and $\hat{\mathbf{r}}_i$. As the inertia is very sensitive to outliers, we first remove them from the set of sessions applying a generalization of the inter-quartile range (IQR) method. First, we calculate the distance $d(s)$ of each session to its closest neighbor (we interpret s as a vector with each entry corresponding to one transition). Then, we move all sessions to a separate noise cluster C_{-1} if $d(s)$ exceeds the range of the two centered quantiles with distance α ($\alpha = 0.5$ corresponds to the IQR method). We call k-means++ on the remaining sessions with η repetitions. The resulting centroids μ_i are the basis for the behavior models.

Algorithm 8.2 Incremental cluster update based on minimum distance.

```
1: $\mathcal{M} := \{\mu_i\}$, $\mathcal{R} := \{r_i\}$, $\hat{\mathcal{R}} := \{\hat{\mathbf{r}}_i\}$, $\mathcal{N} := \{n_i\}$ for $i = -1, 1, \ldots, k$
2: function incrementalUpdate($S$, $\mathcal{M}$, $\mathcal{R}$, $\hat{\mathcal{R}}$, $\mathcal{N}$, $\beta$, $\eta$, $m$)
3:     $C'_i \leftarrow \emptyset$ for $i \in \{-1, 1, \ldots, k\}$
4:     $m' \leftarrow 0$, $\gamma \leftarrow 2t$
5:     for $s \in S$ do
6:         $i \leftarrow \arg\min_{i \in \{1,\ldots,k\}} \left\{\xi_\beta(s, \mu_i, r_i, \hat{\mathbf{r}}_i)\right\}$          ▷ see Equation (8.1)
7:         if $i \neq \emptyset$ then          ▷ $\infty$ is never considered to be minimum
8:             $C'_i \leftarrow C'_i \cup \{s\}$
9:         end if
10:    end for
11:    $\mathcal{C}' \leftarrow \{C'_1, \ldots, C'_k\}$, $\mathcal{R}' \leftarrow \mathcal{R}$, $\hat{\mathcal{R}}' \leftarrow \hat{\mathcal{R}}$
12:    $C'_{k+1} \leftarrow$ findLargestCluster($S \setminus \bigcup_{C \in \mathcal{C}'} C$, $\beta \cdot \max_{i \in \{1,\ldots,k\}}\{r_i\}$, $\eta$)
13:    if $|C'_{k+1}| \geq m$ then
14:        $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{C'_{k+1}\}$, $\mu'_{k+1} \leftarrow \frac{1}{|C'_{k+1}|} \sum_{s \in C'_{k+1}} s$
15:        $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{\max_{s \in C'_{k+1}}\{\|s - \mu'_{k+1}\|\}\}$
16:        $\hat{\mathcal{R}}' \leftarrow \hat{\mathcal{R}}' \cup \{$ transitionRadius($C'_{k+1}$, $\mu'_{k+1}$) $\}$
17:    end if
18:    $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{C'_{-1}\}$ for $C'_{-1} = S \setminus \bigcup_{C \in \mathcal{C}'} C$
19:    $\mathcal{M}' \leftarrow \bigcup_{C'_i \in \mathcal{C}'} \left\{\frac{1}{n_i + |C'_i|}\left(n_i \cdot \mu_i + \sum_{s \in C'_i} s\right)\right\}$
20:    return $\mathcal{C}'$, $\mathcal{M}'$, $\mathcal{R}'$, $\hat{\mathcal{R}}'$
21: end function
22:
23: function findLargestCluster($\overline{S}$, $r_{\max}$, $\eta$)
24:    for $j \in \{1, \ldots, \eta\}$ do
25:        $\overline{\mu} \leftarrow 0$, $\mu^{(j)} \leftarrow$ random element from $\overline{S}$
26:        while $\overline{\mu} \neq \mu^{(j)}$ do
27:            $C^{(j)} \leftarrow \left\{s \in \overline{S} \mid \|s - \mu^{(j)}\| < r_{\max}\right\}$
28:            $\overline{\mu} \leftarrow \mu^{(j)}$, $\mu^{(j)} \leftarrow \frac{1}{|C^{(j)}|} \cdot \sum_{s \in C^{(j)}} s$
29:        end while
30:    end for
31:    return $\arg\max_{C \in \{C^{(1)},\ldots,C^{(\eta)}\}}\{|C|\}$
32: end function
```

In further clusterings, we use radiuses to determine whether a new session belongs to an existing cluster. We consider both the total radius r_i, which we calculate as the maximum distance of a session to the corresponding centroid, and a per-transition radius $\hat{\mathbf{r}}_i$, which we calculate analogously but elementwise. Here, each entry of the vector $\hat{\mathbf{r}}_i$ is the radius of the respective transition. If an entry of the mean μ_i is zero, i.e., none of the sessions of cluster C_i includes such a transition, we set the per-transition radius to ∞. This allows us to assign sessions to the cluster C_i, also if they differ in, e.g., few requests to a new endpoint. The total radius r_i will ensure that the difference between new and previously clustered sessions is within an acceptable range. As the initial clustering builds the basis for incremental updates, it is preferable to cluster a range of sessions as large as possible.

For incremental updates, Algorithm 8.2 assigns new sessions S to the existing clusters or identifies a new one. It does not process the previously clustered sessions again but operates on the existing centroids μ_i, total radiuses r_i, per-transition radiuses $\hat{\mathbf{r}}_i$, and number of sessions per cluster n_i. Hence, its processing time is independent of the number of clustering iterations. The algorithm is based on the *radius criterion* ξ_β with *tolerance factor* $\beta \geq 1$, which defines whether a session s belongs to the i-th cluster (with $\leq_e$ being the elementwise comparison of two vectors):

$$\xi_\beta(s, \mu_i, r_i, \hat{\mathbf{r}}_i) := \begin{cases} \|s - \mu_i\|, & (\|s - \mu_i\| \leq \beta \cdot r_i) \wedge (s - \mu_i \leq_e \beta \cdot \hat{\mathbf{r}}_i) \\ \infty, & \text{else} \end{cases} \tag{8.1}$$

According to ξ_β, some sessions might not belong to any cluster ($\xi_\beta = \infty$ for all clusters). Among these sessions, we try to identify a new cluster (FINDLARGESTCLUSTER, line 23). We loosely base upon Lloyd's Algorithm (Lloyd, 1982) for finding the largest agglomeration of sessions that has a radius of at most the largest radius of an existing cluster. If this agglomeration contains at least m sessions, we handle it as a new cluster (lines 13 to 17). Finally, we assign the remaining sessions to the noise cluster C_{-1} and update the centroids (lines 18 and 19). To avoid exponential growth, we leave the radiuses of the previous clusters unchanged.

The algorithm is capable of identifying new clusters if they have a non-greater radius than the existing ones. Also, it handles sessions that do not fit into any cluster. Future works might improve it by detecting multiple new clusters in one iteration.

8.4.1.2. Incremental Session Clustering using DBSCAN

DBSCAN follows a different approach for clustering a set of data points S. It interprets dense areas as clusters, separated by sparse areas. To identify these, it takes as argument a positive value ε and a minimum number of samples m. It detects all data points $s \in S$ as *core samples* if the number of other samples in the ε-range of s is greater or equal to m. Then, it incrementally constructs clusters by starting with a core sample and adding further data points lying inside the ε-range of the current cluster. Hence, the algorithm can identify the number of clusters without a user's input; for finding appropriate values of ε and m, the literature suggests several heuristics, e.g., based on the distances to the k-th neighbors for a certain k (Schubert et al., 2017). Furthermore, DBSCAN will detect outliers outside the neighborhood of any clusters as noise. Finally, it also can detect clusters that are not convex, which is a limitation of k-means.

Even though these attributes of DBSCAN are superior over k-means, incrementally clustering sessions with DBSCAN bears the challenge that the algorithm bases upon the sessions' neighborhood rather than the minimum distance to a centroid. Thus, we cannot follow a similar approach as with k-means. Furthermore, using the existing clusters for assigning new sessions would require comparing the sessions with increasingly larger clusters, and, thus, increasing clustering durations.

Therefore, we apply the framework from Figure 8.2 and additionally introduce a sliding window w_1 larger than the batch size. Each clustering of one sliding window results in one workload model. Of these models, we consider the last p instances within a second sliding window $w_2 >> w_1$, for calculating a mapping of the new clusters. We formalize this in Algorithm 8.3. It takes as input the sessions S to be clustered, the p previous workload models — represented by sets $\mathcal{M}^{(1)}$ to $\mathcal{M}^{(p)}$ of behavior models (cluster means) —, a mapping f of behavior models to labels, and the DBSCAN parameters ε and m. It clusters S using DBSCAN and calculates the mean of each cluster. For the first cluster, the algorithm assigns each of the means a unique label and returns them as the behavior models.

When clustering a second or further batch of sessions, we map the newly calculated cluster means to the p previous workload models based on the neighborhood (labelClusterMeans). Due to the sliding window w_1 and the resulting overlap of sessions between two subsequent clusterings, the likelihood that the clusterings are related is high. Therefore, we can sensibly map the identified clusters to previous ones. Precisely, we compare each mean with all behavior models of all p workload models and select a label by a majority vote among the p closest ones. However, if the distance between a cluster mean and the behavior models with the chosen label is too high, we consider it a new cluster. Several cluster means may be assigned to the same label. In that case, we merge them. Finally, the algorithm returns the (potentially merged) cluster means as behavior models and the calculated label mapping.

The algorithm clusters the sessions without a predefined number of clusters. Each iteration may result in a different number. Still, the sliding window w_1 allows a sensible comparison with previous clusterings. At the same time, w_1 and w_2 limit the duration of the algorithm. Besides, it can detect clusters that are not convex, and the parameters ε and m can be defined with the support of heuristics. Hence, DBSCAN appears to be a suitable replacement of k-means for session clustering. However, as we discuss in the next section, the absent guarantee for convexity constitutes a significant limitation rather than a benefit, at least for Markov-chain-based behavior models.

Algorithm 8.3 Incremental session clustering using DBSCAN.

```
function incrementalDBSCAN($S$, $\{\mathcal{M}^{(1)},\dots,\mathcal{M}^{(p)}\}$, $f$, $\varepsilon$, $m$)
    $\{C_1,\dots,C_k\} \leftarrow$ DBSCAN($S$, $\varepsilon$, $m$)
    $\mu_i' \leftarrow \frac{1}{|C_i|}\sum_{s\in C_i} s$ for $i = 1,\dots,k$
    $f' \leftarrow$ labelClusterMeans($\{\mu_1',\dots,\mu_k'\}$, $\{\mathcal{M}^{(1)},\dots,\mathcal{M}^{(p)}\}$, $f$, $\varepsilon$)
    $C' \leftarrow \emptyset$, $\mathcal{M}' \leftarrow \emptyset$
    for $\lambda \in f'(\{\mu_1,\dots,\mu_k\})$ do
        $C_\lambda \leftarrow \bigcup_{i\in\{1,\dots,k\} \mid f'(\mu_i')=\lambda} C_i$
        $C' \leftarrow C' \cup \{C_\lambda\}$
        $\mu_\lambda' \leftarrow \frac{1}{|C_\lambda|}\sum_{s\in C_\lambda} s$, $\mathcal{M}' \leftarrow \mathcal{M}' \cup \left\{\mu_\lambda'\right\}$
        $f'(C_\lambda) \leftarrow \lambda$
    end for
    return $C', \mathcal{M}', f'$
end function

function labelClusterMeans($\{\mu_1',\dots,\mu_k'\}$, $\{\mathcal{M}^{(1)},\dots,\mathcal{M}^{(p)}\}$, $f$, $\varepsilon$)
    $f'(\mu_i') \leftarrow$ fresh label for $i = 1,\dots,k$
    $\mathcal{M} \leftarrow \bigcup_{j=1}^{p} \mathcal{M}^{(j)}$
    if $\mathcal{M} = \emptyset$ then
        return $f'$
    end if
    $g \leftarrow$ neighborClassification($\{\mu_1',\dots,\mu_k'\}$, $\mathcal{M}$, $f$, $p$)
    for $i \in \{1,\dots,k\}$ do
        $\lambda_i \leftarrow g(\mu_i')$
        $\mathcal{M}^{(\lambda_i)} \leftarrow \{\mu \in \mathcal{M} \mid f(\mu) = \lambda_i\}$
        $\mu^{(\lambda_i)} \leftarrow \frac{1}{|\mathcal{M}^{(\lambda_i)}|}\sum_{\mu\in\mathcal{M}^{(\lambda_i)}} \mu$
        if $\|\mu_i' - \mu^{(\lambda_i)}\| \leq \max_{\mu\in\mathcal{M}^{(\lambda_i)}}\{\|\mu - \mu^{(\lambda_i)}\|\} + \varepsilon \wedge$
          $\left(|\mathcal{M}^{(\lambda_i)}| > 1 \vee \{\mu_j' \mid j \in \{1,\dots,k\} \wedge g(\mu_j') = \lambda_i\} = \{\mu_i'\}\right)$ then
            $f'(\mu_i') \leftarrow \lambda_i$
        end if
    end for
    return $f'$
end function
```

Table 8.2.: Comparison of Incremental Session Clustering Algorithms

attribute	***k*-means**	**DBSCAN**
parameters	initial clusters (k), repetitions (η), quantile range (α), radius factor (β), min. sessions per new cluster (m)	ε, min. sessions per cluster (m), clustering window (w_1), mapping window (w_2)
num. initial clusters	fixed	determined
detects new clusters	no	yes
cluster shape	convex	various
cluster representative	mean	?

8.4.1.3. Discussion and Choice of Clustering Algorithm

We chose to use the incremental session clustering based on *k*-means. In the following, we discuss this choice. Both proposed algorithms aim at resolving the drawbacks of *k*-means or DBSCAN for incremental session clustering. Based on *k*-means, we added mechanisms for incrementally assigning sessions to existing clusters and detecting new clusters. Using DBSCAN, we rely on the built-in detection of new clusters and map them to previous ones for incremental processing. The *k*-means-based algorithm is preferable because it produces convex clusters while we cannot make such assumptions for DBSCAN. In the following, we compare the two algorithms more detailed and explain why the missing convexness is critical.

Table 8.2 summarizes the essential attributes of the algorithms. Regarding the parameters a user needs to specify, DBSCAN appears superior over *k*-means. While *k*-means requires the hardly estimable number of (initial) clusters, we can use heuristics for estimating the DBSCAN parameters. As a result, DBSCAN determines the number of initial clusters on its own. Also, the DBSCAN-based algorithm has fewer parameters in general. Furthermore, it detects any number of new clusters arising in a later clustering iteration, while we only detect a single one with *k*-means.

Another advantage of DBSCAN over k-means is that it can detect non-convex clusters. That is, given a set of sessions arranged in several strongly skewed agglomerations, it will be able to classify each of them as one cluster. In contrast, k-means attempts to find convex clusters, which will likely differ from the actual agglomerations. At the same time, the lacking assumption of convexness is critical for identifying cluster representatives, which serve as behavior models. With k-means, we can use the centroids. With DBSCAN, the mean of a cluster can lie outside the cluster's outer shape, e.g., in case it resembles a banana. In such a case, the mean is not representative of the cluster. Consequently, we cannot rely on the representativeness of the behavior models extracted by Algorithm 8.3.

Therefore, we use the k-means-based algorithm, but also publish the DBSCAN-based one here. Future works might improve both of them by either supporting estimating the number of initial clusters for k-means or by using a different representative of a DBSCAN cluster. Here, the challenge will be to ensure the comparability of existing representatives and new clusters. Besides, using our k-means-based algorithm in different scenarios might help to find parameter value recommendations, which ease the use of the algorithm. Also, further clustering algorithms could be investigated. A promising type of algorithm is interactive clustering (Bae et al., 2020), which incorporates a user's interaction for, e.g., deriving explainable behavior models. However, it has to be ensured that humans do not block the incremental clustering and, thus, the automation of our approach.

8.4.1.4. Think Time Calculation

Regardless of the clustering algorithm, after we have clustered a new batch of sessions, we calculate the think times per cluster. Here, we base on the previous work (Vögele et al., 2018) for extracting the accumulated think times per transition of the cluster's Markov chain. Hence, there is one think time specification per pair of Markov states. Furthermore, we use the provided encoding as a normal distribution but suggest investigating other distributions in future work.

For ensuring a non-increasing calculation duration, we only use the newly clustered sessions for calculating the think times and merge these with the previously calculated ones. For each Markov transition, we calculate the resulting think time distribution based on the previously calculated one $\Delta \sim \mathcal{N}(\mu_\Delta, \sigma_\Delta^2)$, the newly calculated one $\Delta' \sim \mathcal{N}(\mu'_\Delta, \sigma'^2_\Delta)$, and the respective numbers n and n' of sessions from which we have calculated the distributions:

$$\Delta * \Delta' \sim \mathcal{N}(n \cdot \mu_\Delta + n' \cdot \mu'_\Delta, n^2 \cdot \sigma_\Delta^2 + n'^2 \cdot \sigma'^2_\Delta)$$

Please note, similar to the model-based service-tailoring presented in Section 7.5, this operation does not result in the correct merging of the normal distributions. Instead, it would need to result in a different type of distribution. Therefore, future work should investigate other think time specifications that accurately allow for merging.

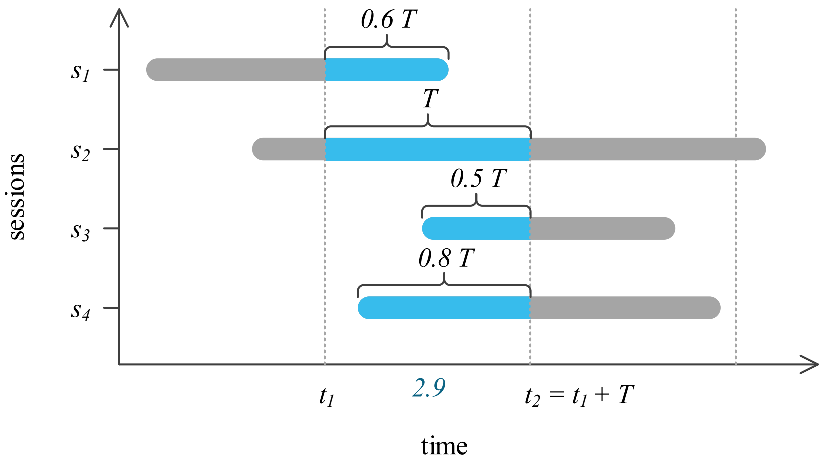


Figure 8.3.: Illustration of the calculation of the intensity from sessions.

8.4.1.5. Intensity Calculation

As a last step of the incremental clustering, we calculate the varying intensity — i.e., the number of concurrent user sessions — per cluster. For that, we consider the sessions of each new cluster separately and calculate the intensity per time interval T, e.g., $T = 1$ minute. For that, we apply an algorithm based on Hidiroglu (2019), which we illustrate in Figure 8.3.

First, we discretize the time into timestamps t_i with $t_{i+1} - t_i = T$. Then, we calculate the intensity value per t_i by summing the time sessions spent between t_i and t_{i+1}. For instance, in the figure, session s_1 started before t_1 and ended at $t_1 + 0.6\,T$. Therefore, we add $0.6T$. s_2 completely overlapped t_1 to t_2; thus, we add T. s_3 started at $t_2 - 0.5\,T$ and ended after t_2, which adds another $0.5\,T$. Similarly, s_4 adds $0.8\,T$. The sum of the time sessions spent between t_1 and t_2 is $2.9\,T$. The intensity value results from the sum divided by the time interval T, i.e., 2.9.

8.4.2. Enrichment with Contexts

Concurrently to the incremental session clustering, our approach aims at continuously annotating the clustering results — i.e., the behavior models and intensities — with the relevant context facets. Such annotation, which includes both past observations and known future states, has two benefits: (1) a user can describe a time range, from which they want to extract a load test, based on the context facets' states; (2) our approach can use the future facet states for improving the intensity forecast. The behavior models, intensities, and stored context information build the knowledge base for the context-tailored load test generation. For proper storing and utilization, we derive a context facet schema. Furthermore, we allow users to register extensions that implement custom handling of certain context facets.

The types of context facets we are interested in are those whose state or influence (or both) is predictable (see Section 8.2). Context facets with unpredictable states and influence can neither be used for describing the time range nor for improving the forecast. For the remaining ones, we store at least the past states we have observed; for those with a predictable state, we also store the future states. For storing the context facets to the WMR, we provide a generic endpoint that can be used for syncing with various data sources, e.g., calendars for sales events, corresponding platforms for weather forecasts, or application monitoring tools for platform incidents.

To find an appropriate schema for these types of context facets, we consider the scales of the collected examples (Section 8.2):

- Many facets, such as special offers (with different types of offers) or product releases (with different products), have a nominal scale. A special case is facets that can either be true or false, e.g., the occurrence of a special event or a recovery from an outage. Also, special offers could be encoded as separate facets with true or false states.
- Other facets, such as weather conditions, can be seen to have an ordinal scale, e.g., rainy, cloudy, and sunny weather in an order.
- Finally, facets are having an interval or ratio scale, such as the temperature. Also, the recovery from an outage could be encoded with different strengths or outage durations.

Given we aim at using the facets as an input to quantitative forecasting tools, and these require numerical inputs, we do not consider ordinal scales. As the distance between two values of such a scale is undefined, we cannot encode it as numerical values. Therefore, we require users to encode facets either in numerical (interval or ratio) scales or to drop the ordering and use a nominal scale. Furthermore, we differentiate between multi-valued nominal scales and boolean scales. Thus, we allow a user or application to define context facets by specifying a context record in JavaScript Object Notation (JSON) — for compatibility with common CSE tooling — per timestamp using the JSON schema (Wright, 2019) in Listing 8.1. As illustrated in Listing 8.2, boolean facets can be specified by adding their name to the respective array. Multi-valued nominal-scaled facets are specified as key-value pairs with string values. Interval- or ratio-scaled facets can be described similarly, but with number values.

```json
{
    "$schema": "http://json-schema.org/draft-04/
       schema#",
    "title": "Workload Context Record",
    "type": "object",
    "additionalProperties": false,
    "properties": {
        "boolean": {
            "type": "array",
            "items": { "type": "string" }
        },
        "string": {
            "type": "array",
            "items": {
                "type": "object",
                "additionalProperties": {
                     "type": "string"
                 }
            }
        },
        "numeric": {
            "type": "array",
            "items": {
                "type": "object",
                "additionalProperties": {
                    "type": "number"
                }
            }
        }
    }
}
```

Listing 8.1: Workload context record schema.

```
{
    "boolean": [ "black_friday" ],
    "string": {
        "product_release": "sneakers"
    },
    "numeric": {
        "temperature": 21
    }
}
```

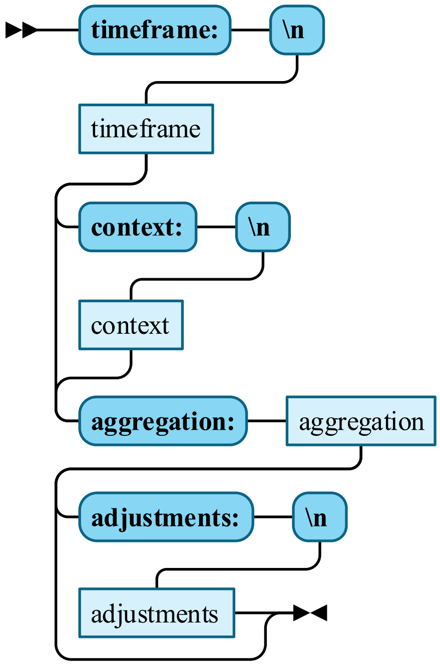
Listing 8.2: Example of a workload context record.

When we receive a new context record, we extract the names and types (boolean, string, or numeric) from the specified facets. For new facets, we register the type. For previously seen facets, we compare the specified type with the stored type and reject the record if they differ. Furthermore, a user can define an optional extension per context facet. For that, they need to implement a code snippet in the R programming language (R Core Team, 2019), which will be called prior to the workload forecasting. The snippet can then change the context facets based on their original states and the intensities. An example where such an extension was required is the recovery from an outage of the message endpoint (H. Schulz et al., 2019c, also see Chapter 15). For achieving accurate forecasts, we had to specify the recovery severity — i.e., an approximation of the number of buffered messages. For that, we based on the duration of the preceding outage, calculated the severity, and set the corresponding facet's state.

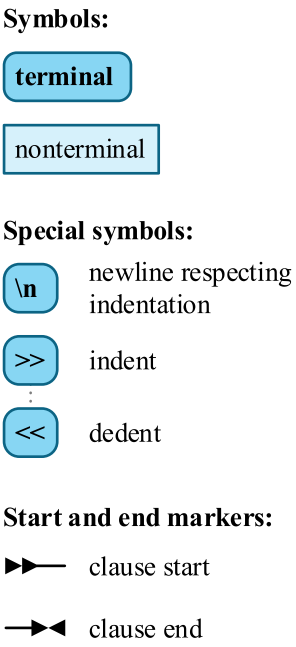
Concluding, we internally register each context facet with its type out of boolean, string, and numeric, and potentially store an extension for custom handling of the facet. Also, we will refer to the context types in the LCtL.

8.5. Load Test Context-tailoring Language

When a user wants to extract a load test from the WMR, they need to describe the workload scenario the test will simulate. For that, we provide the Load



(a) Top-level `scenario` clause.



(b) Legend.

Figure 8.4.: Meta-model of the Load Test Context-tailoring Language (LCtL).

Test Context-tailoring Language (LCtL), which uses the YAML format (*YAML* 2020) for viable integration with CSE. In the following, we provide the meta-model of the language as an extended Backus–Naur form (EBNF), visualized as syntax diagrams (Jensen and Wirth, 1975) and formatted to highlight the YAML structure. In Appendix C, we provide the pure EBNF.

The root clause of an LCtL instance describes a workload scenario (Figure 8.4a) based on the contexts stored in the WMR in up to four sections:

- The `timeframe` section specifies a time frame from which the workload model should be extracted. It can be in the past — then, the load test will replay observed workload — or in the future — for predicting the future workload using quantitative forecasting methods.

- Users can use the `context` section to influence the quantitative forecast by qualitative information. Instead of only relying on the context facet stored in the WMR, they can add additional facet states — e.g., for those with unpredictable states — or change the stored states. Such changes are particularly useful for what-if analyses. The `context` section is optional.
- The `aggregation` section specifies how to extract a workload scenario from the (forecasted) workload of the specified time frame. We allow for replaying the whole time frame, selecting parts of it, or extracting steady-state workloads.
- The final `adjustments` section is optional and allows a user to change the extracted workload scenario. They can use this section for qualitative forecasting exceeding the changed context facets.

Using the four sections, a user can specify all kinds of workload scenarios we classified based on collected examples (see Section 8.6). Scenarios based on recurring or continuous context facets they can define by referring to the stored contexts in the `timeframe` section. Irregular context facets they can specify in the `context` section, for performing what-if analyses. In this case, the workload forecasting will use the past states of the facets and the specified ones for predicting the workload. Finally, for singleton facets, a user can add qualitative forecasts — which they need to do manually — in the `adjustments` section. Notably, entirely unpredictable singleton context facets — such as the extraordinarily prominent posting on a social network (Billington, 2014) — can only be integrated by making assumptions. This is a natural limitation of such facets, which we cannot bypass.

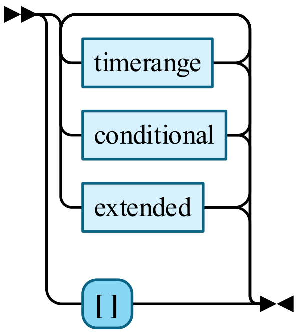
In the following, we introduce the LCtL sections and provide examples.

8.5.1. Timeframe Section

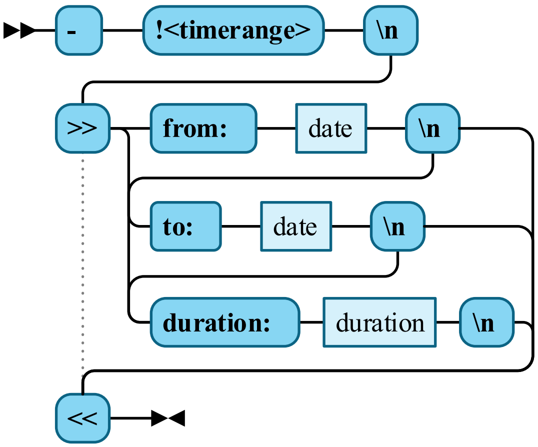
In the `timeframe` section, a user can specify the time frame from which the workload model of the load test will be extracted. Hence, it acts as a query language for selecting data from the WMR. As illustrated in Figure 8.5a, we provide several sub-clauses — `timerange`, `conditional`, and `extended` —, which define sequential restrictions or extensions to an unbound time frame. That is, we start with the unbound time frame and process the sub-clauses from top to bottom, whereas each restricts or extends the time frame specified by the above ones. Alternatively, the time frame can remain unrestricted, expressed by the empty list symbol *[ ]*.

```
timeframe:
- !<timerange>
  from: 2020-04-28T00:00:00
  duration: P3M
- !<conditional>
  product_release:
    is: sneakers
- !<extended>
  beginning: P1D
  end: P1D
```

Listing 8.3: Exemplary `timeframe` section.



(a) `timeframe` clause.



(b) `timerange` clause.

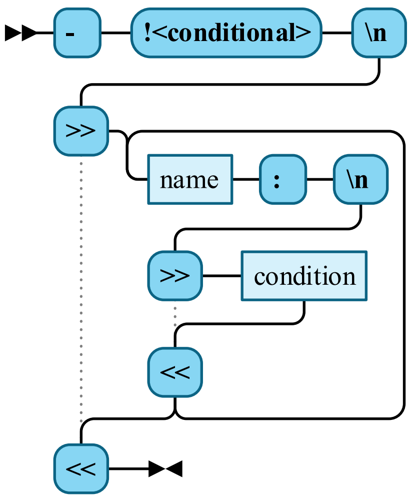
Figure 8.5.: Elements of the `timeframe` section (I).

Listing 8.3 provides a `timeframe` example using all sub-clause types. Below, we explain the meta-models of these clauses.

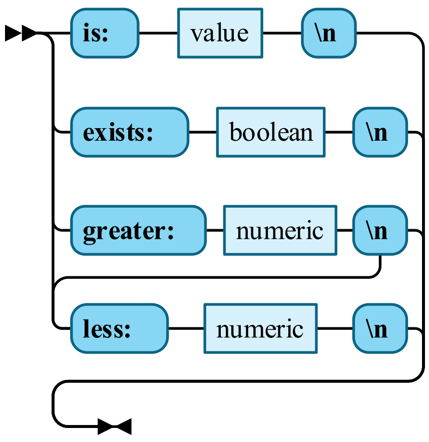
Timerange (Figure 8.5b): A `timerange` defines a time range starting at a defined date and time and having a potentially unbound duration. For that, users can define a *from* date defining the start and a *to* date defining the end. Alternatively, they can specify a *duration* starting from *from* or ending at *to*. For the date and duration specifications, we utilize the ISO 8601 format. If only the *from* or *to* date is specified, the time range is unbound. A *duration* without specified *from* or *to* date we interpret to start at the current date and time. In the example, we define a time range starting from midnight on April 28, 2020, and lasting three months.

Conditional (Figure 8.6a): For easing time frame specification, we allow users to refer to the context facets stored in the WMR. For that, we provide the `conditional` clause. It maps the name of a facet (*name* element) to a `condition` (Figure 8.6b). The allowed conditions then depend on the type of facet. For all types (boolean, string, and numeric), the *is* keyword can be used. It means that only those time frames should be selected where the facet's state equals the specified *value*. Notably, we check whether the value type fits the facet's type, e.g., we reject a boolean value if the facet is numeric. For string facets, users can also restrict to time ranges where the facet *exists*, i.e., has a non-empty state. For numeric facets, we allow comparisons, e.g., whether the temperature is in a range of 20 to 25° C. Here, we treat absent values as 0. In the example, we require the *product_release* facet to equal the string *sneakers*, related to the example by Godlewski (2017). Hence, we restrict the time frame to the range between April 28 and July 28, 2020, and those dates where new sneakers are released. Remarkably, these can be multiple periods.

Extended (Figure 8.7): While the `timerange` and `conditional` clauses restrict the time frame, we also allow extending it with the `extended` clause. It allows specifying a *beginning* and *end* duration, which extend the current



(a) `conditional` clause.



(b) `condition` clause.*

*The `exists` keyword was added as part of the evaluation (Chapter 14).

Figure 8.6.: Elements of the `timeframe` section (II).

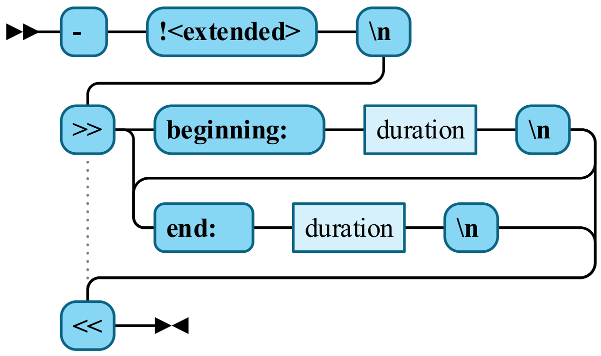


Figure 8.7.: Elements of the `timeframe` section (III): `extended` clause.

time frame by these durations. This is especially useful if users are interested in the time before or after a specific state of a facet. For instance, in the example, we extend the time range to include one day before and after each *sneakers* release.

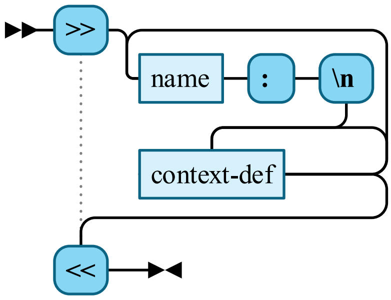
8.5.2. Context Section

While the `timeframe` section only queries data from the WMR, we allow users to modify the queried data in the `context` section. Principally, they can change the stored states of the facets for performing what-if analyses, e.g., simulating an outage recovery. The modification is only for the specific load test extraction and does not change the values in the WMR. Figure 8.8a shows the structure of the clause, which is a simple mapping of facets — identified by their name — to one or multiple `context-def` clauses. Each of these clauses defines how to change the state of that facet. We provide an example in Listing 8.4.

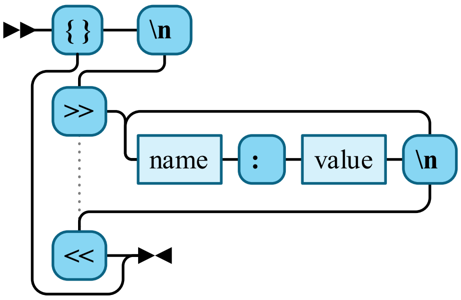
For changing the facet's state, the `context-def` clause (Figure 8.8c) provides three keywords, whose usage again depends on the type of facet. For all types, the *is* keyword can be used to set the state to a specific value. For numeric facets, we additionally provide the keywords *multiplied* and *added*, which can be used individually or in combination. They base on the stored numeric states and multiply these or add a number. A use case for

```
context:
  temperature:
  - added: 5
  outage:
  - is: false
  - is: true
    during:
    - !<timerange>
      to: 2020-04-28T16:00:00
      duration: PT3H
```

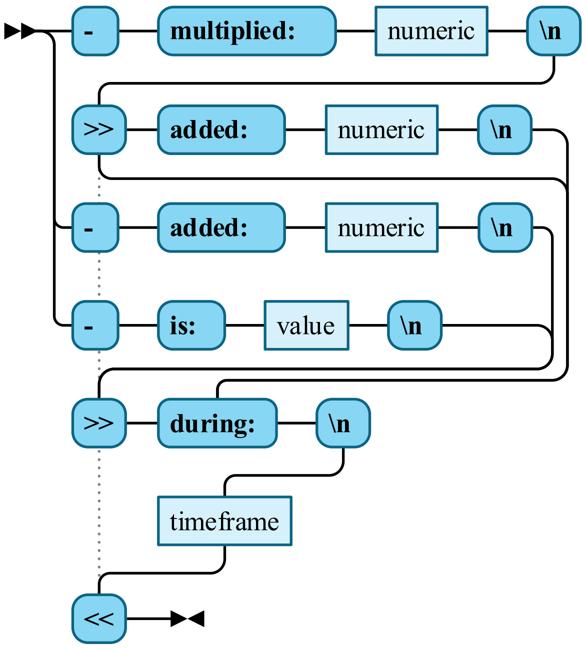
Listing 8.4: Exemplary `context` section.



(a) `context` clause.



(b) `properties` clause.



(c) `context-def` clause.

Figure 8.8.: Elements for context definition and properties.

such modification from the examples is the temperature, which could be adjusted to be, e.g., 5° C warmer.

Besides *is*, *added*, and *multiplied*, the `context-def` clause also has a *during* keyword. It allows a user to restrict the state modification of the facet to a time frame. For specifying these, we reuse the `timeframe` clause. If no *during* is specified, we apply the `context-def` to the whole time frame of the `timeframe` clause. In the example, we make use of *during* for setting the state of an outage facet, which belongs to the recovery example (see Section 8.4.2). It is a boolean facet, so we first set it to false, meaning no outage happens. Remarkably, this will likely be the default state in the WMR, as we cannot foresee outages in the future, but we make it explicit here. Then, we add another `context-def` that overrides the state with true from 1 to 4 pm on April 28, 2020. Hence, the forecasting tool will incorporate the effects of an outage during that time frame, which the load test might — depending on the aggregation — simulate.

8.5.3. Aggregation Section

The third section of the LCtL is `aggregation`, which is mandatory again. It defines how to extract a workload scenario from the selected or forecasted per-group intensities. As the defined time frame can be large, a feasible aggregation is crucial. As shown in Figure 8.9, the `aggregation` section consists of a single clause, which we design to be natively extensible. For that, we proceed similar to context facet extensions (see Section 8.4.2). Users can

```
aggregation: !<percentile>
  p: 95
```

Listing 8.5: Exemplary `aggregation` section.

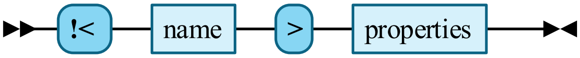


Figure 8.9.: `aggregation` clause.

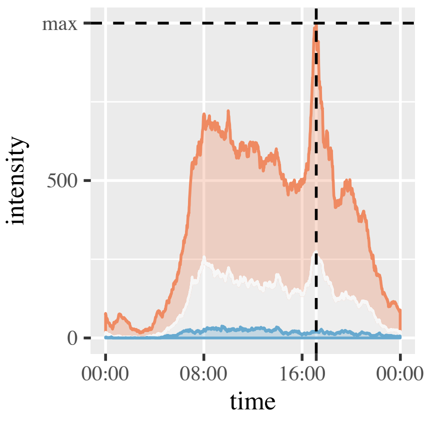
register code snippets implemented in the R programming language (R Core Team, 2019) with unique names and refer to them in the aggregation section. A code snippet gets as input the (potentially forecasted) intensity values per group for the specified time range and needs to extract parts of it. Furthermore, the snippets can be parameterized with properties (Figure 8.8b), which are either empty (`{ }`) or plain key-value maps. As an example, in Listing 8.5, we specify an aggregation with the name `percentile` and a property *p* defining the percentile.

While users can define various aggregations, we introduce four provided by default — `as-is`, `maximum`, `percentile`, and `shapest-spike` —, which we explain in the following.

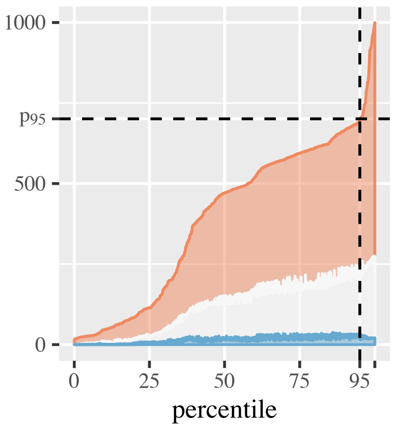
As-is Aggregation: The `as-is` aggregation is the simplest conceivable: it returns the input it receives. It is useful if small time frames are specified or for long-lasting tests, which should replay, e.g., one day. However, it should be used with care, as a load test will replay the whole time frame.

Maximum Aggregation: In contrast to the `as-is` aggregation, which returns a varying workload, the `maximum` aggregation extracts a workload with a stable intensity. As illustrated in Figure 8.10a, it selects the point of time from the input intensities with the maximum total intensities, i.e., the sum of all per-group intensities. Then, it returns the per-group intensity of that point of time as a stable workload. This aggregation is particularly useful for testing whether the system under test (SUT) can withstand the maximum expected workload.

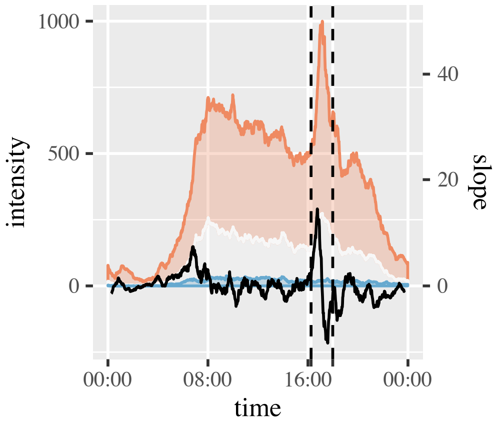
Percentile Aggregation: Because the maximum also includes rare anomalies, which might be significantly higher than the regular intensities, we provide the `percentile` aggregation. As shown in Listing 8.5, it takes as property the percentile number *p*. As illustrated in Figure 8.10b, it selects the intensity vector — with one entry per group — with the total intensity closest to the *p*-th percentile. Similar to the maximum aggregation, it returns a stable workload.



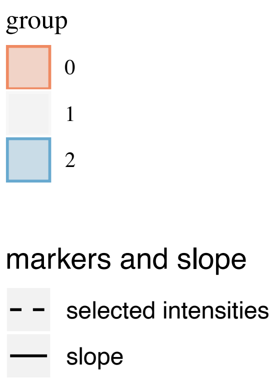
(a) `maximum`.



(b) `percentile`.



(c) `sharpest-spike`.



(d) Legend.

Figure 8.10.: Illustration of extracting workload scenarios from an excerpt of the intensities from our evaluation (Chapter 14) using different aggregations.

Sharpest-spike Aggregation: Another aggregation returning a varying workload is `sharpest-spike`. It identifies the sharpest increase in the input intensities and returns a subset around it, representing a spike. For that, it takes a *window* property as additional input and processes the input intensities, as illustrated in Figure 8.10c. First, it computes the rolling averages with the *window* size of the total intensities. Then, it calculates the slope per time point from the resulting values. From the slope, the aggregation selects the highest value, which represents the sharpest increase. Starting from the corresponding point of time, it takes the latest earlier point of time that has a slope of less than 10 % of the sharpest increase. Furthermore, it identifies the earliest point of time afterwards that is higher than −10 % but later than the first negative slope after the sharpest increase. Then, it returns the intensities within these two points of time.

The rolling average removes local extrema, which would be too short for a load test. Hence, the window property controls the sensitivity of spike detection. It is optional, with a default value of 31 minutes, which is a good value from our experience. Also, we always ensure using a non-even window size, such that the same number of values left and right to each intensity value is aggregated into the rolling average.

```
adjustments:
- !<users-multiplied>
  factor: 1.2
- !<users-added>
  amount: 200
  group: 1
```

Listing 8.6: Exemplary `adjustments` section.

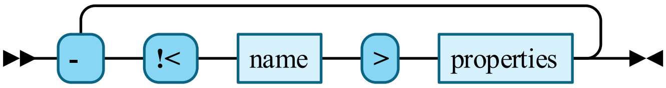


Figure 8.11.: `adjustments` clause.

8.5.4. Adjustments Section

The final LCtL section is `adjustments`, which is optional. Here, a user can incorporate a qualitative forecast into the workload scenario extracted by the `aggregation`. Similar to the `aggregation`, `adjustments` are natively extensible. Figure 8.11 shows the meta-model. Again, users can register R code snippets, refer to them via their registered name, and pass properties. A difference to the `aggregation` is that multiple `adjustments` can be specified, which our approach will execute sequentially. In Listing 8.6, we specify two adjustments — `users-multiplied` and `users-added` —, which apply the respective operations to the intensities. Besides, adjustments can modify the per-group intensities and also the behavior models per group.

By default, we provide two types of adjustments, namely those from Listing 8.6, which modify the intensities of all or individual groups. Hence, they can both adjust the total intensity and the workload mix.

Users-multiplied Adjustment: With `users-multiplied`, a user can specify to multiply the per-group intensities with a factor. For that, it takes the properties *factor* and *group*. The latter property specifies the group whose intensity should be adjusted. It can be left out; in that case, all intensities are adjusted. In Listing 8.6, we specify to add 20 % to all intensities. This relates to the example of adding new devices to the platform at a given date (Section 8.2). Hence, the workload change cannot be predicted by quantitative forecasting, but easily added using this adjustment.

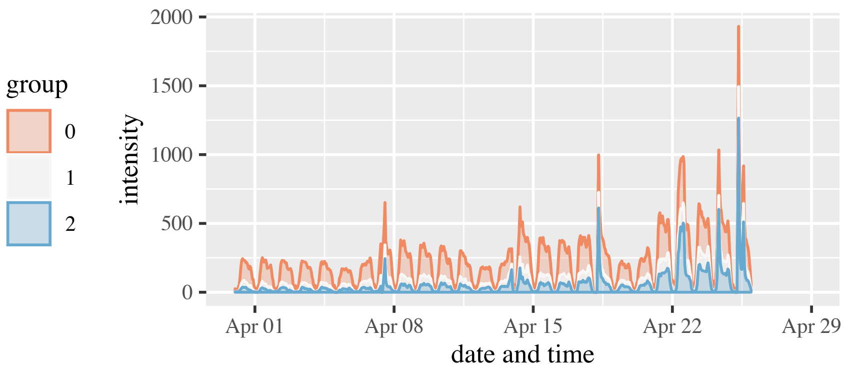
Users-added Adjustment: Besides `users-multiplied`, we provide `users-added`. It works similarly but adds a fixed *amount* to the intensities instead of multiplying the original values. In the example, we specify to add 200 users to the group with ID 1. Please note, as we specified two adjustments, the order is relevant. In this case, our approach will first multiply the intensities of all groups with 1.2 and then add 200 users to group 1. Specifying the adjustments in reverse order will fist add the 200 users to group 1 and then multiply all intensities, resulting in different values.

8.6. Context-tailored Load Test Extraction

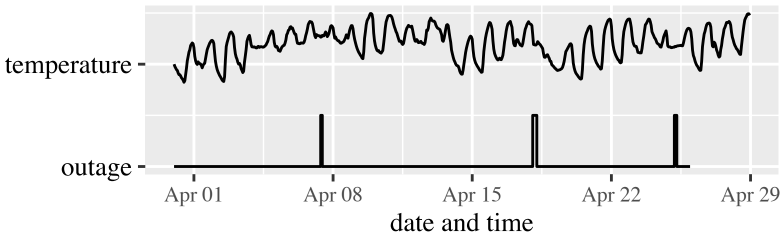
In this section, we detail the part of the context-tailoring process executed on demand (see Section 8.3) as a realization of the LCtL sections. We explain the workload and context preparation, workload forecasting, aggregation, and adjustments. For that, we introduce an exemplary LCtL instance in Listing 8.7, which is related to the previously shown LCtL section examples but simplified. We consider we have collected request logs from 26 days until April 25. Thus, the LCtL instance demands to forecast the workload to April 28. Furthermore, we adjust the context facet with the name *outage* for conducting a what-if analysis simulating the recovery after the outage (see Section 8.4.2). Then, we apply the `sharpest-spike` aggregation and adjust the number of users of group 1. In the following, we explain how an LCtL instance, such as the described one, influences the on-demand load test extraction process.

```yaml
timeframe:
- !<timerange>
  from: 2020-04-28T00:00:00
  duration: P1D
context:
  outage:
  - is: true
    during:
    - !<timerange>
      to: 2020-04-28T16:00:00
      duration: PT3H
aggregation: !<sharpest-skipe> {}
adjustments:
- !<users-added>
  amount: 200
  group: 1
```

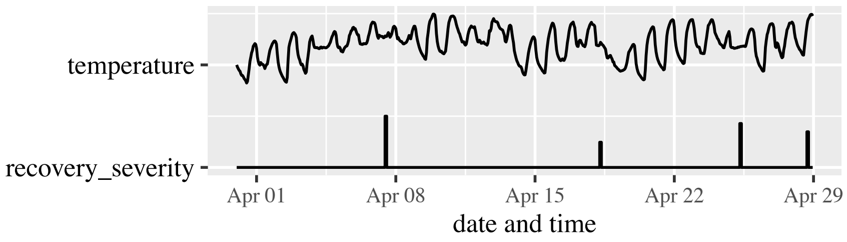
Listing 8.7: Exemplary LCtL instance.



(a) Intensities retrieved from the WMR.



(b) Context facets retrieved from the WMR.



(c) Context facets modified by the `context` section and extensions.

Figure 8.12.: Illustration of intensities and context retrieved from the WMR based on a modified excerpt from our evaluation (Chapter 14). The temperature values are available online.[1]

[1] `https://www1.ncdc.noaa.gov/pub/data/uscrn/products/hourly02/2019/CRNH0203-2019-AL_Clanton_2_NE.txt` (visited on 07/16/2020)

8.6.1. Workload and Context Preparation

In this step, we select all relevant data from the WMR based on the `timeframe` section and potentially adjust the retrieved context based on the `context` section. Furthermore, we apply the extensions registered per context facet. Below, we provide detailed explanations of these three sub-steps.

Workload and Context Selection: First, we select all relevant data from the WMR based on the time frame specified in the `timeframe` section. These are the latest behavior models per group that are before the start of the time frame, the past per-group intensities and context, and the future context until the end of the time frame. The behavior models and intensities constitute the latest state of the incrementally learned workload model. The past and future context we will use for forecasting the workload. We illustrate the intensities and context in Figures 8.12a and 8.12b. We show the context facets temperature and outage retrieved from the WMR. For the temperature, future states are present, while for the outage, only past states are known.

As we aim at using the context facet's states for influencing the workload forecast, we need to transform them into a representation that is usable for the forecasting approach. That is, we need to transform each facet into one or several variables that have numeric values per point of time. We distinguish between the context facet type and proceed as follows. Numeric facets already have an appropriate format. Therefore, they remain unchanged. A boolean facet we transform into one variable by setting its value to 0 if the facet does not occur and to 1 if it occurs. We illustrate this in Figure 8.12b, which represents the boolean outage facet as a variable with values out of 0 and 1. For string facets, we treat each of the states as a boolean facet, i.e., we transform one string facet into one $\{0, 1\}$-valued variable per state. Regardless of the facet type, we treat missing facet states as 0.

Context Modification by Context Section: If the LCtL instance contains a context section, we modify the context retrieved from the WMR. Section 8.5.2 describes the precise operations applied. In the example, we specified to modify the state of the outage facet from 1 to 4 pm on April 28. Hence, for further processing, the outage facet occurs as stored in the WMR and at the specified time frame as well.

Context Modification by Extensions: Finally, we apply all extensions registered per context facet. These extensions may adjust the values of the variable belonging to the facet or generate new variables. For that, extensions get as input the context and intensities retrieved from the WMR, modified and transformed as described above.

As described previously, the outage context facet is an example that requires customized handling by an extension (H. Schulz et al., 2019c). Here, we conceive an extension that transforms the outage into a *recovery severity*. This variable describes the severity of the recovery spike, which will occur after the end of the outage. The higher the value of is, the sharper and higher the recovery spike will be. The forecasting approach will calculate the actual spike steepness and height. In Figure 8.12c, we illustrate the recovery severity calculated based on the outage length and intensity during the outage.

8.6.2. Workload Forecasting

If the LCtL instance specifies a `timeframe` in the future, such as the example in Listing 8.7, we need to forecast the future workload. For that, we base on the data prepared in the previous step, i.e., the per-group behavior models, intensities, and context. Then, we apply forecasting approaches to the intensities of each group, also considering the contexts. As a result, we yield the future intensities, which, in combination with the behavior models, build the total workload mix. This procedure is superior over a pure forecast of the total intensity, as it also can predict the mix.

As forecasting approaches, we utilize Telescope (Bauer et al., 2020), Prophet (Taylor and Letham, 2018), and a perfect forecast. The latter one is only relevant for evaluation purposes, as it returns the actual intensities that lie in the future from a defined perspective. Hence, we can evaluate the quality of the incrementally learned workload model without the effect of potentially inaccurate forecasts. However, in real scenarios, future intensities will be unknown, and we have to rely on actual forecasting. In the following, we describe the forecasting of the workload using Telescope and Prophet.

Telescope: We utilize a version of Telescope for multivariate forecasting available online (Chair of Software Engineering, University of Würzburg, 2020). This version can predict the future values of numerical time series and also include so-called *covariates*. Each covariate is another time series Telescope will use for adjusting the forecast. Hence, we can use them to integrate the prepared context. We predict the intensities of each group separately. For this, Telescope takes as input the past intensity values and the past formatted values of the context variables, whereas each variable corresponds to one covariate. Furthermore, we specify the future values of the variables as future covariates. Telescope will then learn the impact of the covariates from the past data and calculate the forecast appropriately.

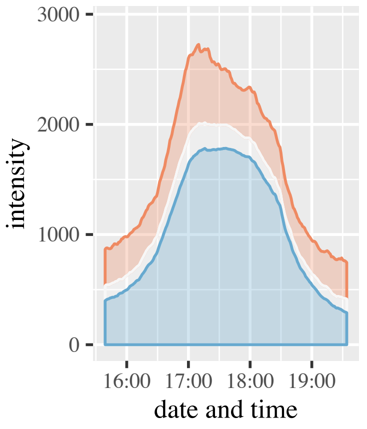
Prophet: Prophet is another time series forecasting tool related to Telescope. Similarly, it provides forecasts for numerical time series that can include so-called *regressors*, which correspond to Telescope's covariates. Thus, we proceed similarly as with Telescope and predict each group's intensities individually. For that, we specify the context variables as regressors and input them in addition to the past intensities. Prophet then outputs the intensity forecast. As the concepts of covariates and regressors correspond, we can sensibly compare Telescope and Prophet in our evaluation.

8.6.3. Workload Aggregation

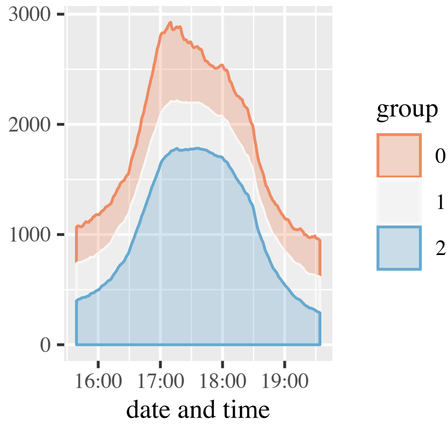
The workload aggregation step corresponds to the `aggregation` section of the LCtL and extracts a workload scenario from the intensity forecast — in case of a future `timeframe` — or the intensities selected from the WMR. Hence, the potentially long period of intensities is reduced to one that a load test can replay. As described in Section 8.5.3, we provide multiple aggregations and also allow for custom ones, which can extract steady-state or varying intensities. In the example, we specified applying the `sharpest-spike` aggregation, which extracts the spike with the highest slope of the total intensity. We illustrate the result of this aggregation in Figure 8.13a, which shows the varying intensities of three groups. As we have defined an outage, the intensities represent the recovery spike caused by the outage.

8.6.4. Workload Adjustments

For integrating qualitative forecasts, we can adjust the workload scenario extracted by the aggregation using the `adjustments` section of the LCtL. Qualitative forecasts need to be done by a user and can, e.g., change the intensity of one group. In Figure 8.13b, we illustrate adding 200 users to group 1, as defined in the exemplary LCtL instance. In combination with the respective behavior models, these intensities constitute the workload model that will be transformed into the load test.



(a) Intensities extracted using the `aggregation` section.



(b) Intensities adjusted using the `adjustments` section.

Figure 8.13.: Illustration of intensities extracted and adjusted using the LCtL instance from Listing 8.7.

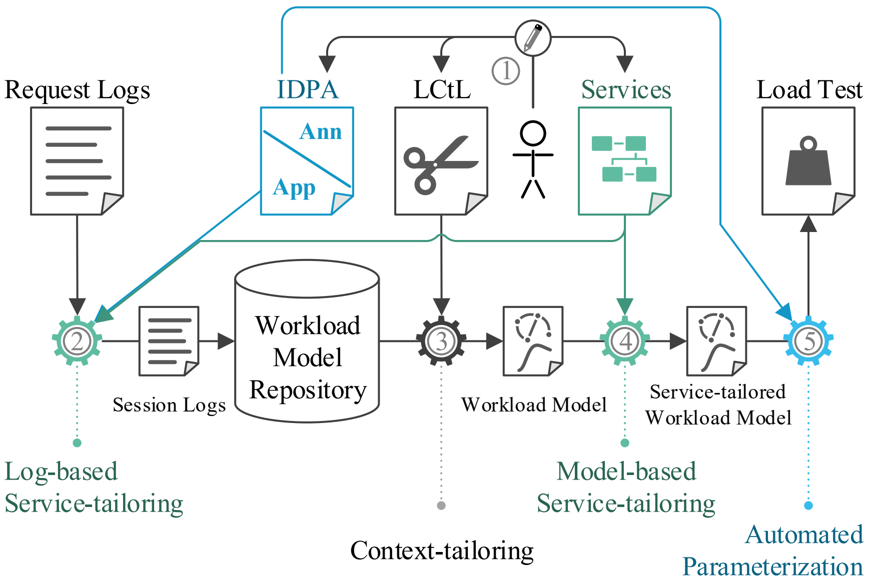
8.7. Integration with Service-tailoring and Automated Parameterization

In this section, we describe the integration of the context-tailoring approach with the approaches introduced in the previous chapters. These approaches are the automated parameterization of the generated load test using an Input Data and Properties Annotation (IDPA) (see Chapter 6) and the two service-tailoring approaches (see Chapter 7). As log-based and model-based service-tailoring addresses artifacts at different stages, we need to integrate them differently.

We provide an overview of all integrations in Figure 8.14. For automatically generating a service- and context-tailored load test, a user needs to specify an IDPA, the list of services to be tested, and an LCtL description ①. Notably, they need to specify them at different points in time. While the list of services and the LCtL description are required at test generation time, the IDPA can be defined in advance and reused for subsequent load test generations.

During the continuous learning of the workload model, we utilize the IDPA for labeling the requests (see Section 6.5.2) ②. At the same time, if log-based service-tailoring (see Section 7.4) should be applied, it is applied here. As a consequence, log-based service-tailoring needs to be configured before learning the workload model. Then, we build separate behavior models and corresponding intensities per list of services.

Based on the LCtL description, we extract a context-tailored workload model from the WMR, as described in this chapter ③. When using model-based service-tailoring (see Section 7.5), we apply it as the next step ④. It modifies the extracted workload model to target the services under test directly. Finally, we transform the workload model into a load test and apply the automated parameterization with the IDPA (see Section 6.5.3) ⑤. Using the same IDPA for labeling the requests and the parameterization ensures that the workload model's states and the IDPA fit. If the API changes during the workload model learning, a user needs to adjust the IDPA, which allows for extracting a load test for the latest API version.



Request Logs
IDPA
Ann
App
LCtL
①
Services
Load Test
②
Session Logs
Workload Model Repository
③
Workload Model
④
Service-tailored Workload Model
⑤
Log-based Service-tailoring
Context-tailoring
Model-based Service-tailoring
Automated Parameterization

Figure 8.14.: Integration of context-tailoring with service-tailoring and automated parameterization.

Because we have to apply log-based and model-based service-tailoring at different stages, the model-based approach has several advantages over the log-based one. First, it does not require specifying the list of services to be tested in advance. Instead, it modifies the context-tailored workload model for the whole application. Second, it underlies fewer fluctuations in the intensities. As the workload arriving at a non-user-faced service depends on the call behavior of other services, which can change frequently and significantly when new service versions are introduced, the intensities of the log-based approach have lower predictive power for the future workloads. In contrast, the model-based approach is based on the non-service-tailored intensities and can always use the most current traces. For these reasons, we recommend using model-based service-tailoring in combination with context-tailoring.

8.8. Summary

In this section, we introduced our approach for tailoring load tests to workload contexts, addressing RQ3: *How can representative load tests automatically be tailored to the contexts of a session-based workload?*

Relying on a continuously learned workload and context knowledge base, a user can generate a context-tailored load test by specifying an appropriate description. For that, we provide the Load Test Context-tailoring Language (LCtL). Then, leveraging the automated load test parameterization, our approach automatically generates a load test that fits the description. The context-tailoring accompanies the service-tailoring for generating resource-efficient and time-efficient load tests, as these approaches allow generating load tests that restrict to the relevant contexts and services.

In the next chapter, we leverage context- and service-tailoring for enabling load testing for non-experts. In Chapter 14, we provide an evaluation of the context-tailoring approach using the request logs from the student information system of Charles University, Prague.

Chapter 9

Enabling Load Testing for Non-experts

Even though load testing is widely considered a crucial quality assurance technique (Jiang and Hassan, 2015), Bezemer et al. (2019) have found that it is often not conducted regularly and adequately. The authors attribute this finding to a barrier of a significant amount of knowledge and expertise required for conducting sound performance engineering such as load testing. This reasoning is especially valid for complex load-testing-based tasks, such as assessing the scalability of different deployment alternatives of a microservice application (Avritzer et al., 2020a). Consequently, Walter et al. (2016) have introduced the notion of *declarative performance engineering*, demanding for decoupling a user's performance concern from the specific solution approach.

In the previous chapters, we have introduced approaches that ease specific parts of the load testing process. Precisely, we allow for generating tailored load tests automatically. Other approaches solve the challenge of automating the test execution lifecycle (Ferme and Pautasso, 2018) or the comprehensible presentation of the load test results (Okanović et al., 2019). Also, as an example of a complex load-testing-based task, we have developed a unified process and measures for assessing the scalability of microservice deploy-

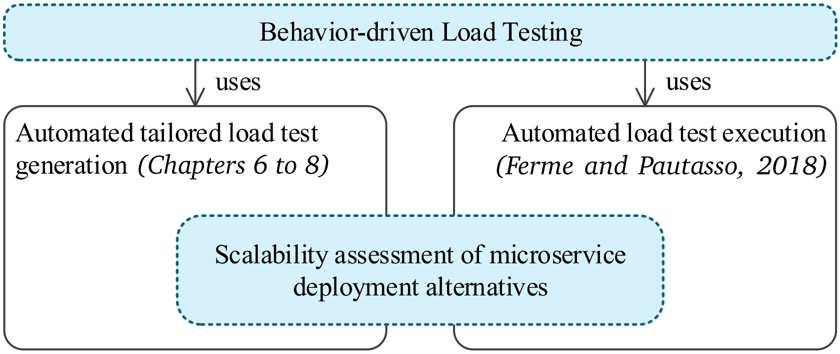

Behavior-driven Load Testing
uses
uses
Automated tailored load test generation (Chapters 6 to 8)
Automated load test execution (Ferme and Pautasso, 2018)
Scalability assessment of microservice deployment alternatives

Figure 9.1.: Overview of the contributions of this chapter.

ment alternatives (Avritzer et al., 2020a), which we realized based on the previously mentioned approaches. However, automated integration of the test generation and the test execution approaches is missing. Furthermore, the approaches require technical specifications, e.g., in the YAML format, that still can form a barrier for sound adoption. In the scalability assessment case, a user also needs to coordinate specific steps of the experiment process.

Therefore, we aim at easing the load testing process for non-experts by coherently integrating automated load test generation and execution. Users shall specify their load test concerns abstracted from technical constructs using template-based natural language. Also, they shall define complex tasks such as the scalability assessment as a single specification. This aim addresses RQ4: *How can we leverage automated tailored load test generation and automated load test execution for enabling load testing for non-experts?*

As a solution, we base upon the Gherkin language (Wynne et al., 2017) used for Behavior-driven Development (BDD) (North, 2006) for providing the Behavior-driven Load Testing (BDLT) language. As illustrated in Figure 9.1, it integrates the features of our approaches for the automated tailored load test generation (Chapters 6 to 8) and the BenchFlow approach by Ferme and Pautasso (2018) for automated load test execution. Furthermore, we integrate the scalability assessment of microservice deployment alternatives

(Avritzer et al., 2020a) as an example of a complex load testing task. For ensuring a high degree of comprehensibility, BDLT specifications are readable in natural language, as illustrated by the following example:

> *Given the next Black Friday, when varying the CPU cores between* 1 *and* 4*, then ensure the maximum CPU utilization is less than* 60 %.

The remainder of this chapter is structured as follows. In Section 9.1, we describe the process of automatically processing a BDLT specification. Section 9.2 introduces the BDLT language. In Section 9.3, we present the scalability assessment of microservice deployment alternatives as a use case of BDLT, followed by a summary in Section 9.3.

This chapter is based on the following joint publications, which appeared in advance:

- H. Schulz, D. Okanović, A. van Hoorn, V. Ferme, and C. Pautasso (2019c). "Behavior-Driven Load Testing Using Contextual Knowledge — Approach and Experiences." In: *Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE 2019)*. ACM, pp. 265–272
- A. Avritzer, V. Ferme, A. Janes, B. Russo, H. Schulz, and A. van Hoorn (2018). "A Quantitative Approach for the Assessment of Microservice Architecture Deployment Alternatives by Automated Performance Testing." In: *Proceedings of the 12th European Conference on Software Architecture (ECSA 2018)*. Vol. 11048. Lecture Notes in Computer Science. Springer, pp. 159–174
- A. Avritzer, V. Ferme, A. Janes, B. Russo, A. van Hoorn, H. Schulz, D. Menasché, and V. Rufino (2020a). "Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-Based Approach Leveraging Operational Profiles and Load Tests." In: *Journal of Systems and Software* 165, p. 110564

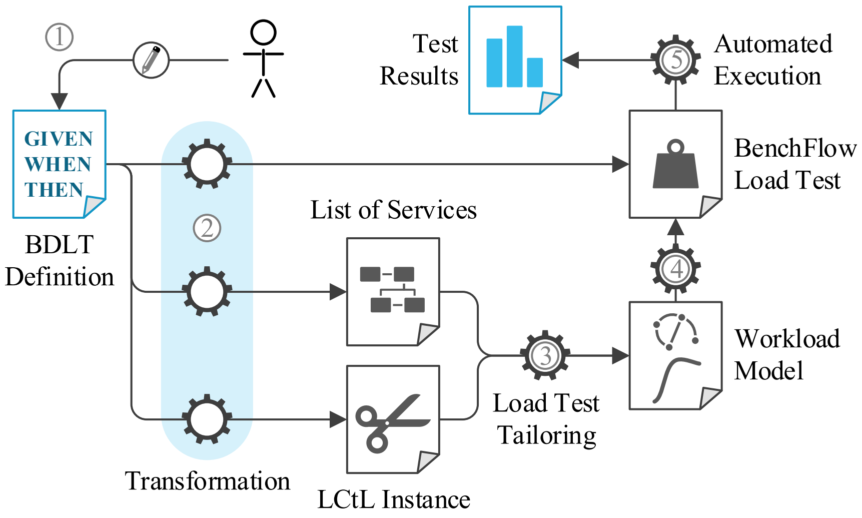


Figure 9.2.: Overview of the Behavior-driven Load Testing process.

9.1. Behavior-driven Load Testing Process Overview

In this section, we provide an overview of the process of generating and executing a load test using the BDLT language. In the next section, we will introduce the language and detail the individual steps of this process. Figure 9.2 illustrates it.

The only points where a user needs to interact are the initial input and final output. As input, a user needs to specify a BDLT definition ①, such as the example in the chapter introduction. With this definition, they describe their load testing concern, consisting of three parts: the initial context and configurations (*given*), changes made to the initial state (*when*), and the expected outcome of and behavior during the test (*then*). The example uses the *Black Friday* context for specifying the workload, states to vary the number of *CPU cores*, and expects the *CPU utilization* to be less than a threshold. Our approach relies on this definition for generating and executing a BenchFlow load test fully automatically, such that we can present the test results to the user.

For that, we transform the BDLT definition into several other models ②. BenchFlow provides a declarative domain-specific language (DSL) that allows for specifying a load test, including the workload model and the configuration of the system under test (SUT) (see Section 4.4.2). Hence, those parts of the BDLT definition describing the load test execution and setup of the SUT are transformed directly into a BenchFlow DSL instance. In the example, such parts are the CPU core variation and the CPU utilization expectation. For also specifying the workload model, we extract the list of services to be tested — if defined — and a Load Test Context-tailoring Language (LCtL) instance. The Black Friday from the example forms a context the LCtL instance will hold. Then, we use our load test tailoring approaches (Chapters 7 and 8) for generating the workload model ③. Using the transformation from the WESSBAS-DSL to the BenchFlow DSL by Palenga (2018) and our automated load test parameterization (Chapter 6), we complete the BenchFlow load test ④.

Finally, we utilize BenchFlow for executing the load test ⑤. In doing so, it automates the whole test lifecycle, including the deployment of the SUT. Besides, it manages the execution of several experiments. For instance, given the example BDLT definition, BenchFlow will execute four experiments — one per number of CPU cores — for evaluating the behavior of the differently configured SUT under the same workload. After the load test execution has finished, we can finally provide the user with the load test results. In this work, we do not include specific preparation of the results for non-experts. For that, we refer to the work by Okanović et al. (2019).

9.2. Behavior-driven Load Testing Language

The BDLT language is the core of our load testing approach for non-experts. It allows users to define load tests and their execution using template-based natural language. For designing the language, we adopt concepts from the Gherkin language (Wynne et al., 2017) used in BDD (North, 2006). In the following, we describe the relation of the BDLT language to Gherkin (Section 9.2.1), introduce the metamodel (Section 9.2.2), and describe the transformation to the LCtL (Section 9.2.3). We omit details on the transformation to the BenchFlow DSL. For that, we refer to our previous publication (H. Schulz et al., 2019c).

9.2.1. Behavior-driven Development as a Basis

BDD (North, 2006) is a paradigm for better integrating functional unit testing into software development. It builds upon test-driven development (TDD) (Beck, 2003), which aims at using tests as a specification for the expected results of the developed software. BDD emphasizes that the tests should define the intended *behavior*. For that, it utilizes simple phrases describing the behavior, which can then be mapped to unit tests. Hence, non-technical stakeholders can be better involved in the testing and development process. With BDLT, we follow a similar approach, as we want to allow users to specify an SUT's intended behavior under a particular configuration and workload. Instead of unit tests, we need to map the phrases to load tests.

For that, we leverage concepts from the Gherkin language (Wynne et al., 2017), which is used in the context of BDD. A Gherkin behavior description is a single natural-language sentence that is structured according to a three-part template. It starts with the keyword *given*, followed by a description of the initial state of the software part, such as a class, to be tested. The second part starts with *when*, followed by changes to be made to the initial state. Finally, the sentence holds postconditions identified by *then*. For testing whether the software fulfills the behavior description, the sentence is mapped to classes and unit tests. The postconditions define whether the test should fail.

For the BDLT language, we reuse the three-part template. However, we need to define specific clauses we can map to load tests rather than unit tests. Specifically, we apply the following structure, which we detail in the next section.

- Starting with *given*, a user can describe the initial workload context, services to be tested, and configuration of the SUT. The example from the introduction defines Black Friday as the workload context.

- The *when* clause holds changes to the initial state, e.g., for testing varying configurations such as in the example or conducting what-if analyses based on irregular workload-influencing contexts.
- In the *then* clause, a user can specify the expected behavior during the test as requirements — such as the example's CPU utilization threshold — and how to react when a requirement is violated.

9.2.2. Metamodel

We introduce the metamodel of the BDLT language in extended Backus–Naur form (EBNF) notation, which we present here using syntax diagrams (Jensen and Wirth, 1975) with the same symbols as in Section 8.5. We provide the pure EBNF in Appendix D. *The BDLT language was first published in our previous work (H. Schulz et al., 2019c).*

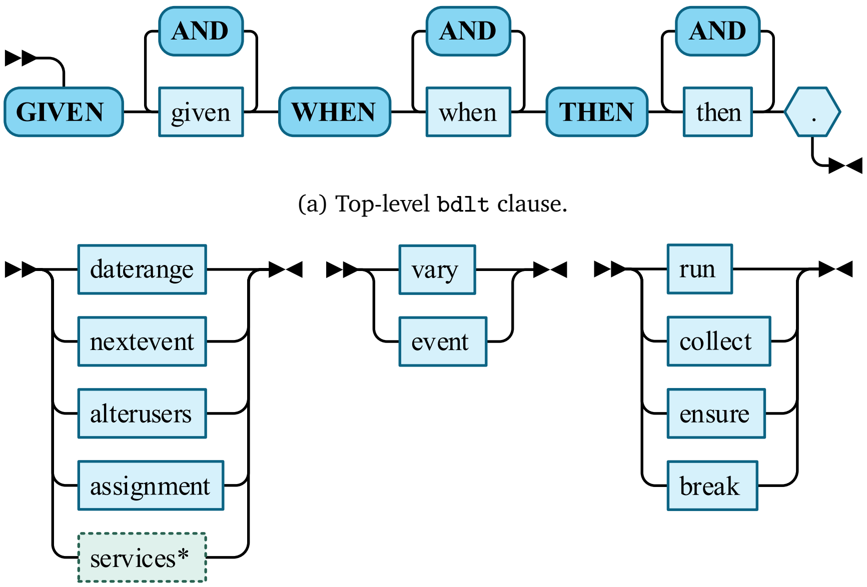
As previously described and illustrated in Figure 9.3a, a BDLT definition consists of three types of clauses starting with *given*, *when*, and *then*, respectively. Multiple clauses of the same type can be concatenated using the *and* keyword. Besides, clause types can be left out. For instance, the example from the introduction without the *when* clause is sufficient for testing whether the CPU utilization is below the threshold under the Black Friday workload:

> *Given the next Black Friday, then ensure the maximum CPU utilization is less than* 60 %.

We provide 11 different types of *given*, *when*, and *then* clauses (see Figures 9.3b to 9.3d), which we explain in the remainder of this section. The language can easily be extended by adding further clauses.

9.2.2.1. Given Clause

Following the *given* keyword, one out of five clauses can be used to describe the initial context and configuration of the load test. `daterange` and `nextevent` describe time frames from which the workload model should be



(a) Top-level `bdlt` clause.

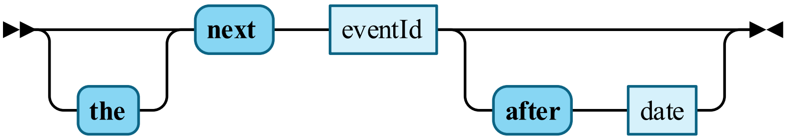
(b) `given` clause. (c) `when` clause. (d) `then` clause.

*not part of the previous publication and the evaluation

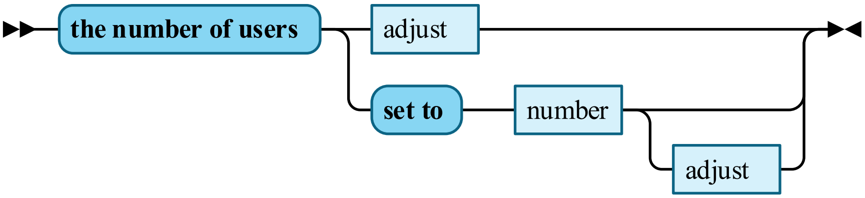
Figure 9.3.: Meta-model of the Behavior-driven Load Testing (BDLT) language (based on H. Schulz et al., 2019c).

extracted. `alterusers` changes the number of users. With `assignment`, a user can describe an initial configuration. `services` should be used to state the services to be tested. Below, we describe the clauses in detail and provide examples.

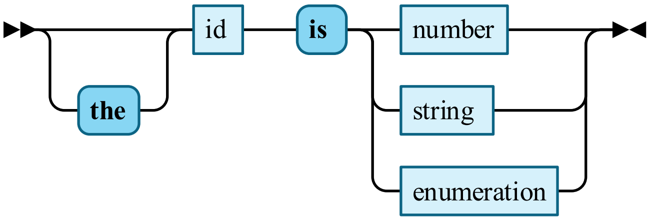
Daterange: The most straightforward description of the time frame is the `daterange` clause (no figure). It explicitly states a time range, e.g., *given 4 days starting from April 1, 2020*. We allow for different date formats, which the BDLT language parser needs to interpret. Based on our context-tailoring approach, we can select the workload that happened during the defined time range — if it was in the past — or predict the future workload.



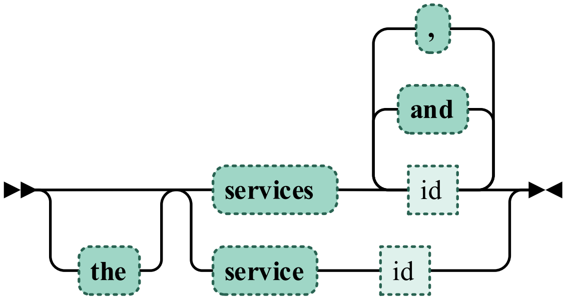
(a) `nextevent` clause.



(b) `alterusers` clause.



(c) `assignment` clause.



(d) `services` clause (not part of the previous publication and the evaluation).

Figure 9.4.: Elements of the `given` clause (based on H. Schulz et al., 2019c).

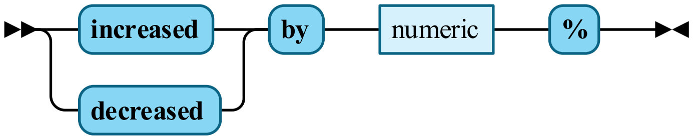
Nextevent (Figure 9.4a): As a more implicit description of the time frame, a `nextevent` clause states to use the workload that is expected to happen during a future event. An example is *given the next Black Friday*. We utilize our context-tailoring approach to predict the workload. For that, a user — e.g., an expert setting up our approach — needs to register *Black Friday* as a context facet in advance (see Section 8.4.2).

Alterusers (Figure 9.4b): With an `alterusers` clause, a user can change the number of users simulated in the load test, which is initially determined based on the `daterange` and `nextevent` clauses. It can either add or subtract a specific percentage of users or set it to a fixed value. In the latter case, a user can also refer to the initial value, such as the following clause: *given the number of users set to the maximum increased by* 20%. The clause refers to the `adjust` and `number` clauses, which we describe in Paragraph *Utility Clauses*.

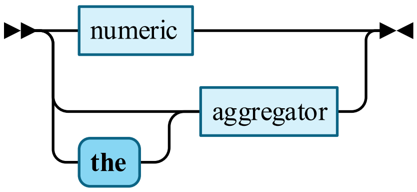
Assignment (Figure 9.4c): The `assignment` clause is for setting the configuration of the SUT. For that, it maps a configuration property such as the amount of memory to a value, e.g., *given the memory is 4 GB*. The value can be a number, string, or enumeration of strings for a complex property. For executing the load test, we transform the property assignments to the `exploration_space` field of the BenchFlow DSL, such that BenchFlow will configure the SUT correspondingly.

Services (Figure 9.4d): We added the `services` clause in addition to the previously published version of the language to allow users to test specific services explicitly. The services are defined as a simple list, such as *given the services carts and payment*. Then, we utilize our service-tailoring approach for generating a workload model that targets the specified services directly.

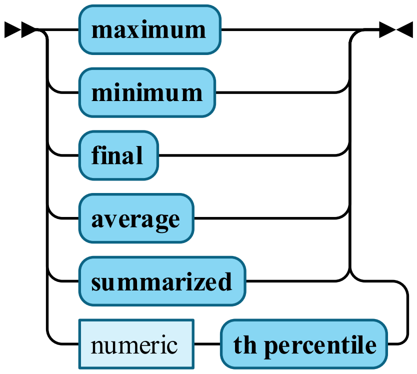
Utility Clauses (Figure 9.5): In the previous paragraphs, we referred to several utility clauses, which we introduce here. The first one is the `adjust` clause, which increases or decreases a value by a percentage. We use it to



(a) `adjust` clause.



(b) `number` clause.



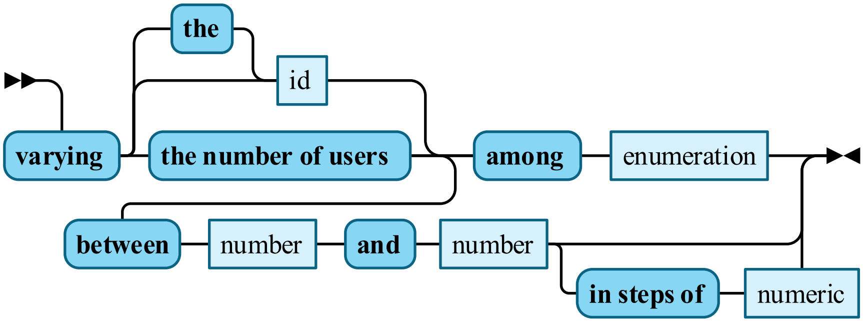
(c) `aggregator` clause.

Figure 9.5.: Utility clauses (based on H. Schulz et al., 2019c).

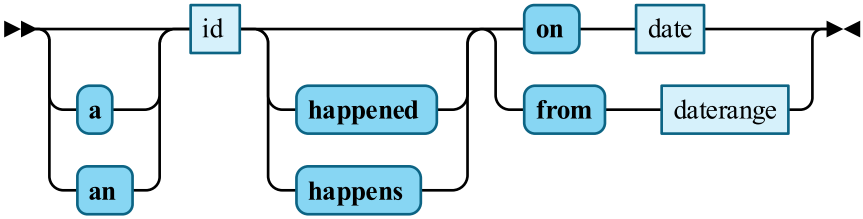
change the number of users determined by our approach. For specifying numbers, we provide the `number` clause. It can either define an exact number or an `aggregator`, which refers to the surrounding clause. As an example, we can set the number of users to the maximum determined value in the `alterusers` clause.

9.2.2.2. When Clause

As we describe in the following paragraphs, the *when* clause can modify the workload and SUT configuration defined in the *given* clause. `vary` can vary the number of users or a configuration property among a set of values. `event` can emulate the occurrence of a workload-influencing event.



(a) `vary` clause.



(b) `event` clause.

Figure 9.6.: Elements of the `when` clause (based on H. Schulz et al., 2019c).

Vary (Figure 9.6a): The `vary` clause has two purposes. First, it allows users to test the SUT under different configurations. For example, the following clause will test three different memory configurations: *when varying the memory among (2 GB, 4 GB, 8 GB)*. With the same mechanism, users can test the SUT under multiple numbers of simulated users, such as *when varying the number of users between the average and the maximum in steps of 100*. Again, users can state exact numbers or refer to the initially determined number of users. For varying configurations or numbers of users in the load test, we utilize the `exploration_space` field of the BenchFlow DSL — similar to the assignment clause. For determining aggregate values based on the determined number of users, we additionally utilize our context-tailoring approach.

Event (Figure 9.6b): To account for workload-influencing events whose occurrence is irregular, we provide the `event` clause. A user can state that a specific event happens, such that the load test constitutes a what-if analysis. An example is when the operators of a webshop want to find an appropriate date for a bargain-sale such that their system will be able to handle it. For checking the effect at a particular date, they can state *when a bargain-sale happens from April 1 to April 4, 2020*. For predicting the expected workload, we use our context-tailoring approach. The *bargain-sale* needs to be registered as a context facet, potentially with custom handling using an extension (see Section 8.4.2).

9.2.2.3. Then Clause

As the final part of a BDLT definition, users can influence the runtime behavior of the load test and define requirements. They can explicitly set the test duration using `run` or define termination criteria using `break`. Also, they can `collect` specific metrics and define pass/fail criteria based on the metrics using `ensure`. We transform all these clauses into properties of the BenchFlow DSL, namely `steady_state`, `termination_criteria`, `observe`, and `quality_gates`. BenchFlow will then take care of controlling the runtime, collecting metrics, and determining whether the load test has failed. The following provides the metamodels of the clauses.

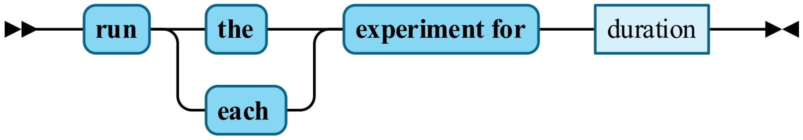
Run (Figure 9.7a): With `run`, a user can explicitly set the test duration, e.g., *then run the experiment for 1 h*. The clause is not mandatory, as for some workloads, e.g., varying ones, the workload scenario to be replayed implicitly defines the duration. However, users can override the duration, e.g., for simulating only the beginning of the scenario, or specify one for steady-state workloads.

Collect (Figure 9.7b): Using the `collect` clause, a user can configure BenchFlow to collect a specific metric. As an example, *then collect the CPU utilization* will cause BenchFlow to measure and collect the specified metric.

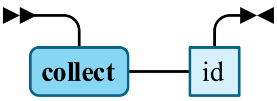
It does not perform any analyses on the metric. For that, one of the following clauses should be used.

Ensure (Figure 9.7c): `ensure` is similar to `collect` but defines a `check` on the collected metric besides. As an example, a user can state to collect the CPU utilization and fail the test if its maximum is above a threshold, e.g., *then ensure the maximum CPU utilization is less than 60 %*. BenchFlow automatically performs the check and determines whether the test failed.

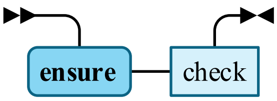
Break (Figure 9.7d): Finally, the `break` clause is similar to `ensure` but additionally controls the test duration. For that, BenchFlow evaluates the check already during the test execution. If the check fails, the test will stop. The clause helps minimize the test duration in case of failures, such that the user receives fast feedback. However, it reduces analysis possibilities, as it shortens the test duration. Therefore, `ensure` might often be a better alternative, as it does not stop the test and defines the same pass/fail criteria.



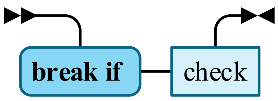
(a) `run` clause.



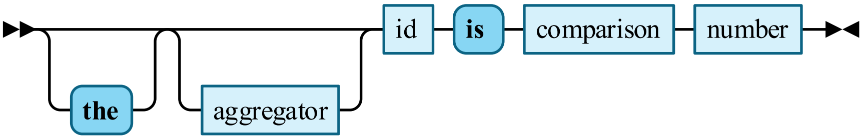
(b) `collect` clause.



(c) `ensure` clause.



(d) `break` clause.



(e) `check` clause.

Figure 9.7.: Elements of the `then` clause (based on H. Schulz et al., 2019c).

9.2.3. Transformation to Load Test Context-tailoring Language

We execute a load test defined using the BDLT language by transforming it into the BenchFlow DSL. While we can transform some clauses directly, others require preprocessing, e.g., for calculating the expected workload during a specified event. For that, we utilize our service-tailoring and context-tailoring approaches. The service-tailoring approach only needs the list of services defined in the `services` clause. The context-tailoring approach is based on the LCtL. Hence, we need to transform the BDLT definition to an LCtL instance. The clauses that are relevant for the transformation are `daterange`, `nextevent`, `alterusers`, `vary`, and `event`.

Before describing the transformation, we introduce two new LCtL aggregations (see Section 8.5.3) using the extension mechanism. The first one is `fixed`, which takes as input a list of *intensities*. Then, it returns multiple per-group intensities, which sum to the passed *intensities*. The second one is `intensity-range`, which takes as input a *lower*, *upper*, and *step* value. It behaves similarly as `fixed`, but determines the intensities based on the provided range and step. As *lower* and *upper* values, it can both process numbers and aggregators. In the latter case, it calculates the range borders based on the intensity values before the aggregation.

In the following, we explain the transformations of the `daterange`, `nextevent`, `alterusers`, `vary`, and `event` clauses to LCtL sections, also using the `fixed` and `intensity-range` aggregations. In the case that the BDLT does not specify an aggregation explicitly, we use the `as-is` aggregation replaying the observed or predicted workload.

9.2.3.1. Transformation of Daterange

The `daterange` clause defines to extract a workload scenario from the range between two specific dates. The analogous concept of the LCtL is the `timerange` clause in the `timeframe` section. Hence, given a `daterange` clause in the BDLT definition, we generate a `timerange` clause as illustrated in Listing 9.1.

```
timeframe:
- !<timerange>
  from: [start of daterange]
  to: [end of daterange]
```

Listing 9.1: timeframe section transformed from a daterange clause.

```
timeframe:
- !<conditional>
  [eventId]:
    is: true
- !<timerange>
  from: [current or specified date]
```

Listing 9.2: timeframe section transformed from a nextevent clause.

```
 # for plain numeric:
aggregation: !<fixed>
  intensities: [numeric]
 # for aggregators except percentile:
aggregation: !<[aggregator]> {}
 # for percentile:
aggregation: !<percentile>
  p: [numeric]

 # only for adjust:
adjustments:
- !<users-multiplied>
  factor: [1+numeric/100]: # for 'increased'
  factor: [1−numeric/100]: # for 'decreased'
```

Listing 9.3: aggregation and adjustments sections transformed from an alterusers clause.

9.2.3.2. Transformation of Nextevent

The LCtL analog to the `nextevent` clause is the `conditional` clause. As Listing 9.2 shows, we generate one such clause that states the context facet with the specified *eventId* needs to be *true*. Hence, the context-tailoring will extract a workload scenario from the time frame where the event occurs. To ensure only future dates or those after the specified date are considered, we add a `timerange` clause.

9.2.3.3. Transformation of Alterusers

Altering the number of users with the LCtL can be done in the `aggregation` and `adjustments` sections. The section we generate depends on the specific instance of the clause. If it contains *set to*, we overwrite the default aggregation as follows and shown in Listing 9.3:

- If the number following *set to* is fixed, we use the `fixed` aggregation with the specified number as single-element *intensities* list.
- If it is a percentile, we use the `percentile` aggregation (see Section 8.6.3).
- If it is an `aggregator` different from percentile, we use the respective aggregation without any additional properties.

Regardless of the existence of *set to,* if the clause contains an `adjust` sub-clause, we add a `multiply-users` adjustment (see Section 8.6.4). We transform the specified percentage to a *factor* to increase or decrease the number of users accordingly.

9.2.3.4. Transformation of Vary

We only consider the `vary` clause in the LCtL if it varies the number of users. In this case, we define an appropriate aggregation — as shown in Listing 9.4 —, which depends on the sub-clause used:

- If the `enumeration` clause is used, we generate a `fixed` aggregation with the *intensities* list holding all specified numbers of users.

- For the sub-clause starting with *between*, we generate an `intensity-range` aggregation with the *lower*, *upper*, and *step* values defined. If the clause does not contain a *step* value, we use 1 as default.

In both cases, the context-tailoring returns multiple intensity values, for which BenchFlow will execute one load test each.

9.2.3.5. Transformation of Event

The final BDLT clause relevant for the LCtL is `event`. It explicitly defines an event as a workload context, which we need to consider when generating the workload model. Therefore, we add a `context` section to the LCtL instance stating the value of the context facet corresponding to the event should be *true*. Furthermore, we restrict the value to the specified *date* or *daterange* by adding a `timerange` clause to the *during* field.

```
 # for enumeration
aggregation: !<fixed>
  intensities: [enumeration]
 # for 'between':
aggregation: !<intensity-range>
  lower: [number]
  upper: [number]
  step: [numeric] # default is 1
```

Listing 9.4: `aggregation` section transformed from a `vary` clause.

```
context:
  [id]:
  - is: true
    during:
    - !<timerange>
      from: [date or start of daterange]
       # only for daterange:
      to: [end of daterange]
```

Listing 9.5: `context` section transformed from an `event` clause.

9.3. Use Case: Scalability Assessment of Microservice Deployment Alternatives

In joint work with Avritzer et al. (2018, 2020a), we have developed an approach for assessing the scalability of different deployment alternatives of a microservice application. Finding an optimal deployment configuration can be challenging, as the application needs to be able to handle various workloads. Also, the configuration space is plentiful, including different virtualization and containerization technologies, such as Docker (Docker Inc., 2020), the number of microservice replicas, and the hardware resources available to each microservice. Therefore, we introduced the *Domain-based metric*, which assesses the ability of a deployment alternative to handle the workload scenarios occurring in production, and a unified experimentation process for determining the Domain-based metric. Comparing the Domain-based metric for different deployment alternatives allows microservice application operators to choose the best alternative easily.

As the determination of the Domain-based metric comprises multiple load tests, we can use the BDLT language for easing the experimentation process. Precisely, we can utilize our context-tailoring approach for extracting the relevant workload scenarios, BenchFlow for executing the load tests for different deployment alternatives, and the BDLT language as a shared interface. Hence, the BDLT language enables a high degree of automation and high understandability for non-experts in addition.

In the following, we introduce the details of the scalability assessment approach with the Domain-based metric and describe how to leverage the BDLT language for these experiments. In the evaluation (Chapter 15), we will furthermore investigate the expressiveness of the BDLT language for the experiments we already have executed in our previous work.

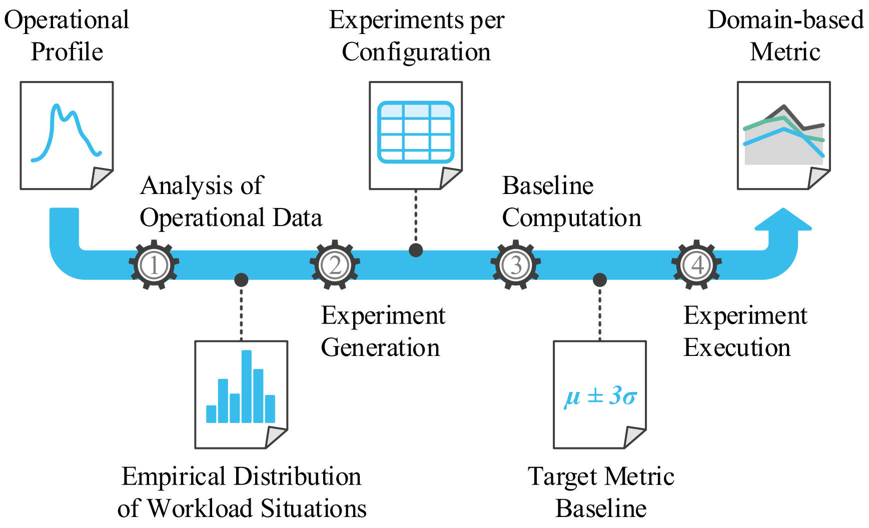


Figure 9.8.: Overview of the Domain-based metric calculation (based on Avritzer et al., 2020a).

9.3.1. Assessing the Scalability with the Domain-based Metric

The goal of the Domain-based metric is to rate the scalability of a microservice deployment alternative. Given a set of deployment alternatives and collected production workload (operational profile), we execute several load tests per alternative for calculating the metric for each of them. Then, we provide a comparison for recommending an optimal deployment configuration. As depicted in Figure 9.8, the Domain-based metric calculation comprises four steps, which we describe in the following.

9.3.1.1. Analysis of Operational Data

In the first step, we analyze the production workload and extract relevant test cases. For that, we treat each point of time of the workload as a *workload situation* characterized by a specific metric, e.g., the number of users. Compared to our previously used terminology, a workload situation equals a steady-state workload scenario. As we cannot execute a load test for all identified workload situations, we aggregate them into *k* bins. As a result, we yield the empirical workload distribution represented by *k* workload situations with a frequency of occurrence in the production workload. The frequencies we will use for weighting the test results.

9.3.1.2. Experiment Generation

The second step is the generation of the experiments to be executed. Each of them consists of a load test replaying a workload situation and an architectural configuration. Hence, we cross-join the workload situations extracted in the first step with the set of deployment alternatives. For instance, given we have obtained ten workload situations and need to analyze three deployment alternatives, we generate 30 experiments.

9.3.1.3. Baseline Computation

The Domain-based metric is composed of the ratio of requests made during the load tests that passed a particular criterion. The criterion is based on a target metric — in our previous work, we have used the response time — and a baseline value per endpoint called by the load tests. For determining the baseline, we execute the first experiment with a low workload, e.g., two users, and assess the mean and standard deviation of the target metric per endpoint. Then, we define the per-endpoint baseline value as the mean plus three times the standard deviation. For the actual experiments, we will compare the mean target metric value with the baseline and mark the endpoint to pass if its mean value is below the baseline or to fail, otherwise. We presume that higher workload will stress the system more, such that the target metric's value increases, which is especially valid for performance metrics like the response time.

9.3.1.4. Experiment Execution

The fourth and final step is the execution of the experiments. We utilize BenchFlow for executing all of them and retrieving the target metric per experiment and endpoint. Then, we evaluate the pass/fail criterion and calculate the Domain-based metric per deployment alternative. For each alternative, we proceed as follows. Let δ_j be the fraction of requests to the j-th endpoint among all requests and c_j be the pass/fail criterion per endpoint and experiment, i.e.,

$$c_j = \begin{cases} 1, & \text{endpoint } j \text{ passes} \\ 0, & \text{endpoint } j \text{ fails} \end{cases}$$

Given the SUT consists of microservices with n endpoints overall, we define the fraction $\hat{s}(\lambda)$ of passed requests for the workload situation λ as below.

$$\hat{s}(\lambda) = \sum_{j=1}^{n} \delta_j c_j$$

Finally, we calculate the Domain-based metric as the sum of fractions $\hat{s}(\lambda)$ for all workload situations $\lambda_1, \ldots, \lambda_z$ weighted with the respective frequency of occurrence $p(\lambda_i)$ (see the first step):

$$D(\alpha, S) = \sum_{i=1}^{z} p(\lambda_i)\hat{s}(\lambda_i) \tag{9.1}$$

Here, α stands for the deployment alternative and S for the test suite consisting of the workload situations.

For visualizing and comparing the Domain-based metric per deployment alternative, we further introduce a specific plot. We define the Domain-based metric per workload situation as below.

$$D(\alpha, S, i) = p(\lambda_i)\hat{s}(\lambda_i)$$

Hence, $D(\alpha, S)$ (Equation (9.1)) is the sum of all $D(\alpha, S, i)$. Then, we generate a plot as the final output of the Domain-metric calculation, as in Figure 9.9. It consists of the maximum metric value, i.e., $p(\lambda_i)$, visualized as an outer polygon. Besides, it holds one polygon per deployment alternative. The

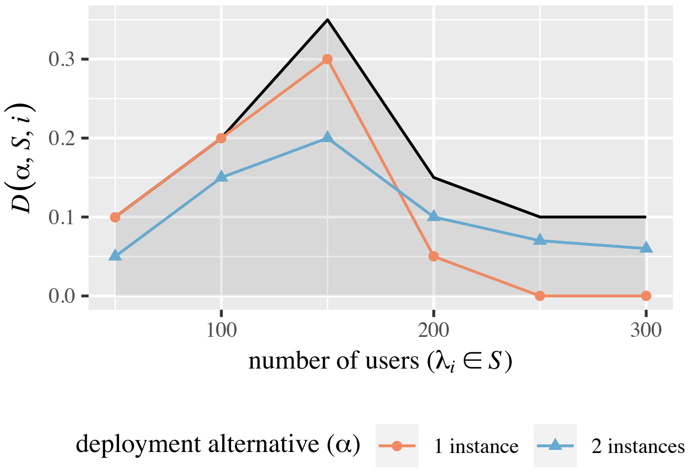


Figure 9.9.: Exemplary Domain-based metric plot (based on Avritzer et al., 2020a). The outer polygon is the maximum metric value per workload situation λ_i.

example figure shows the Domain-based metric for two deployment alternatives with different numbers of instances of a particular microservice. It illustrates that a single instance is better suited for handling lower workloads, while two instances are preferable for high workloads.

9.3.2. Expressing Scalability Experiments with the BDLT Language

Even though the scalability assessment with the Domain-based metric provides a high degree of automation, it still requires expert knowledge. For instance, users have to deal with the production workload, which is an input to the assessment process. Using the BDLT language, we can add another abstraction layer, which provides the users with an easy-to-understand interface. Namely, a user can assess the scalability of a microservice application using the template shown in Listing 9.6.

Given such a BDLT definition, our context-tailoring approach will automatically select the relevant workload — e.g., the last half-year — and extract

```
Given [a workload time range],
when varying the number of users between the minimum and the maximum
    in steps of [step size],
and varying [certain configurations]
then collect the Domain—based metric.
```

Listing 9.6: BDLT template for scalability assessment.

all workload situations with the specified step size. The result is an empirical distribution of workload situations as output by the first step of the Domain-metric calculation process. Then, these workload situations are automatically forwarded to BenchFlow, which executes the corresponding load tests for all specified configurations. These configurations correspond to the deployment alternatives.

The only extension we need to add to the original BDLT approach is a custom data collector that computes the Domain-based metric. For that, we can utilize the extension mechanism of BenchFlow, which allows implementing and adding such custom collectors. By connecting the collector with the *Domain-based metric* keyword, BenchFlow calculates and returns the metric automatically.

9.4. Summary

In this chapter, we introduced the Behavior-driven Load Testing (BDLT) language, which allows non-experts to express load tests. For that, a BDLT definition is readable in natural language, consisting of the *given*, *when*, and *then* clauses known from Behavior-driven Development (North, 2006). For generating and executing a defined load test automatically, we leverage our approaches introduced in Chapters 6 to 8 and the experiment automation framework BenchFlow (Ferme and Pautasso, 2018).

While the BDLT language is less expressive than the LCtL introduced Chapter 8 — which we use as an intermediate artifact in the BDLT process —, it is less technical and easier to understand. Still, it can be used for expressing complex load testing tasks such as the scalability assessment of different deployment alternatives of a microservice application. Besides, it integrates the load test execution, which is out of the scope of the LCtL. Hence, we presume the BDLT language to be more usable for non-experts. For evaluating the usefulness and expressiveness in industrial and laboratory contexts, we present two case studies in Chapter 15.

CHAPTER 10

IMPLEMENTATION

For evaluation purposes, we implemented a prototype of our approach as part of the *ContinuITy* (2020) project. Besides the approach presented in the previous chapters, we also developed services that extend interoperability or support experimentation. The source code is available in different repositories on GitHub, as specified below. Besides, we deliver all services as Docker (Docker Inc., 2020) containers on DockerHub (H. Schulz, 2020d). In the following, we provide an overview of the implementations. Section 10.1 focuses on our core approach, while Section 10.2 describes the additional implementations.

10.1. Approach Implementation

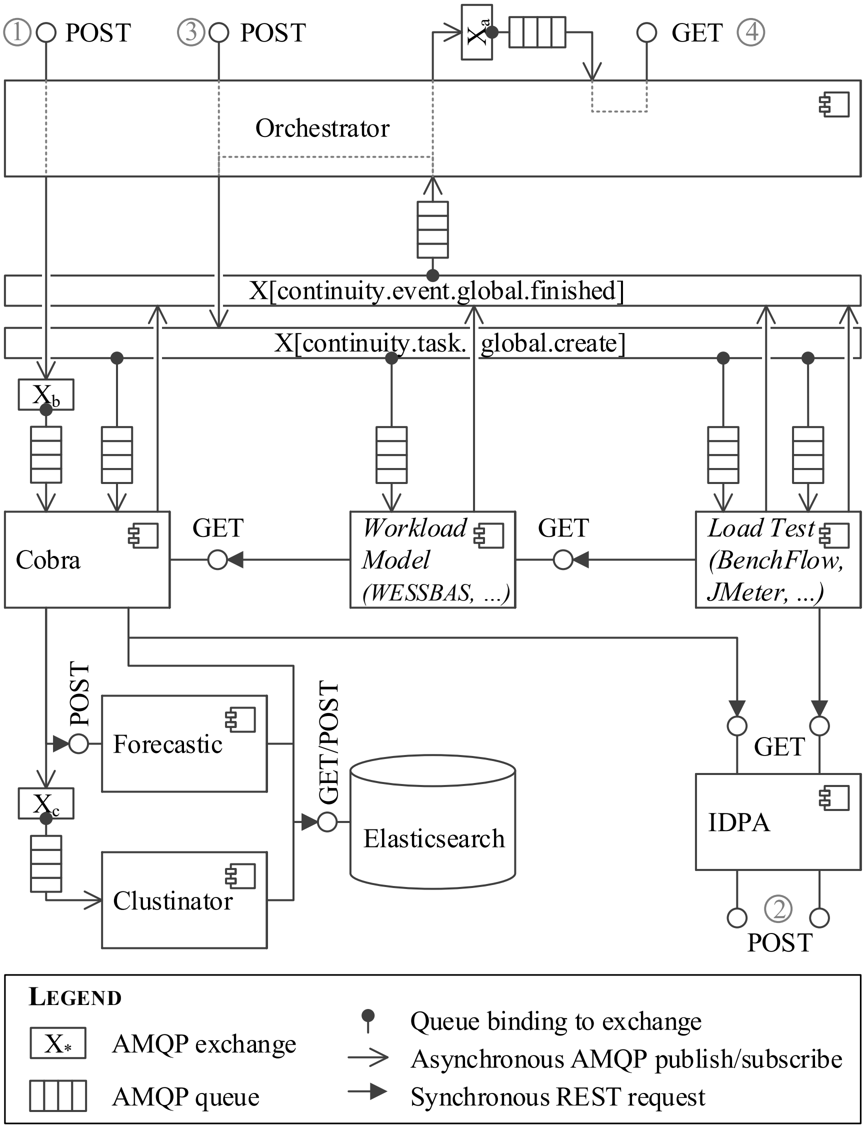
We have implemented our approach — which we introduced in Chapters 6 to 9 — in a framework with an extensible service architecture. It allows for automated generation of tailored load tests, and we have used it in our evaluation. Multiple services provide different features for load test generation, such as parameterization management (see Chapter 6), incremental workload model learning (see Chapter 8), and automated extraction of a load test from this workload model (see Chapters 7 and 8). The services are implemented in Java (Arnold et al., 2005), Python 2 (Van Rossum and Drake Jr, 1995), and R (R Core Team, 2019); for technology-independent communication, we base upon the Advanced Message Queuing Protocol (AMQP) and Representational State Transfer (REST) over the Hypertext Transfer Protocol (HTTP). We provide the source code of the services online (H. Schulz, 2020a; H. Schulz et al., 2020a; H. Schulz and Dang, 2020).

In the remainder of this section, we detail the service architecture, provide an overview of the available services, introduce a command-line interface (CLI) for interacting with our framework, and describe its essential processes.

10.1.1. Extensible Service Architecture

Our implementation consists of multiple services, each providing a specific functionality required for generating tailored load tests. Figure 10.1 illustrates their architecture. The *Orchestrator* constitutes the API gateway for the users. They can interact with the service via REST, and the Orchestrator will then orchestrate further services if required. *Cobra* is responsible for incremental workload model learning and preparation of a context-tailored workload model — as described in Chapter 8. For that, it utilizes the services *Forecastic* and *Clustinator*. The *IDPA* service manages Input Data and Properties Annotation (IDPA) instances. Finally, multiple services are responsible for transforming workload models into load tests — such as WESSBAS — or parameterizing and executing load tests — such as BenchFlow and JMeter.

The services communicate via REST and RabbitMQ (Pivotal Software, Inc., 2020[a]), which is an implementation of AMQP. For abstracting from IP addresses or hostnames and for dynamic service discovery, all services need to register at a central Eureka (Netflix, Inc., 2012) service. Besides, there are global message queues, at which the services can register using their name and type of artifact they generate as routing key. Hence, users can state which services to use, and the Orchestrator can consider the respective services for, e.g., generating a load test. As a result, several services are interchangeable, allowing for extending our framework efficiently. For instance, for using a specific load testing tool, such as Gatling (Gatling Corp, 2020), a service



① POST
③ POST
X_a
GET ④
Orchestrator
X[continuity.event.global.finished]
X[continuity.task. global.create]
X_b
Cobra
GET
Workload Model (WESSBAS, ...)
GET
Load Test (BenchFlow, JMeter, ...)
POST
Forecastic
GET/POST
X_c
Clustinator
Elasticsearch
GET
IDPA
②
POST
LEGEND
X_* AMQP exchange
AMQP queue
Queue binding to exchange
Asynchronous AMQP publish/subscribe
Synchronous REST request

Figure 10.1.: Overview of the service architecture (H. Schulz et al., 2020a).

that transforms workload models into Gatling load tests can be added by registering it at Eureka and the corresponding message queue.

Some essential services, however, are the Orchestrator, Cobra (including its helper services), and IDPA, because they implement specific parts of our approach. For instance, users can upload monitored requests or sessions for the incremental workload model learning ①, which the Cobra service processes. The IDPA service receives and stores IDPAs a user uploads ②. The Orchestrator receives orders from a user ③, orchestrates the other services for processing the orders, and allows the user to retrieve the orders' results ④. In the following sections, we describe these processes in more detail.

10.1.2. Available Services

The following list provides an overview of all services that are currently available. If not stated otherwise, the services are implemented in Java using the Spring Boot (Pivotal Software, Inc., 2020[b]) framework. All services with a REST API provide an OpenAPI (OpenAPI Initiative, 2020) specification. Not listed explicitly, there are also a Eureka and a RabbitMQ management service. Besides, we provide a CLI, which a user can execute locally and which we describe in the next section.

Orchestrator: As mentioned previously, the Orchestrator constitutes the API gateway and orchestrates the other services. Furthermore, it manages service configurations, which a user can define. Besides Spring Boot, it uses the Zuul (Netflix, Inc., 2013) application gateway.

IDPA: The IDPA service stores IDPAs (see Chapter 6), which users upload, and allows other services to retrieve them.

Cobra: This service is the start of most load test generations, as it prepares the initial data. It implements the context-tailoring (see Chapter 8), including the incremental session clustering. Also, it implements the log-based service-tailoring (see Section 7.4). It employs Forecastic and Clustinator as helper services and an Elasticsearch (Elasticsearch B.V., 2020) instance for storing workload models, intensities, and contexts.

Forecastic: Forecastic is implemented in R and aids Cobra in forecasting workload intensities. We have chosen R, because it provides forecasting tools, such as Telescope (Bauer et al., 2020) and Prophet (Taylor and Letham, 2018), as libraries. The service is triggered by Cobra via REST and accesses the intensities stored in Elasticsearch.

Clustinator: For clustering sessions incrementally, we provide the Clustinator service. It is implemented in Python, as this language provides the powerful scikit-learn (Pedregosa et al., 2011) library. It accesses Elasticsearch for retrieving sessions and storing clustering results, and it is triggered via an asynchronous message queue.

WESSBAS: The WESSBAS service is a workload model service and a wrapper around the WESSBAS approach (Vögele et al., 2018). It can extract WESSBAS-DSL instances from sessions and transform these instances into load tests. Here, we also implemented the model-based service tailoring (see Section 7.5).

RequestRates: This service is an alternative to WESSBAS and extracts and transforms request-based workload models. We use it in Chapter 13 for generating baselines of service-tailored load tests.

BenchFlow: BenchFlow (Ferme and Pautasso, 2018) is a load test automation framework, as described in Section 4.4.2. This service generates parameterized instances of the BenchFlow DSL, which then can be executed by BenchFlow.

JMeter: As an alternative to BenchFlow, we provide the JMeter service. It parameterizes JMeter (Apache Software Foundation, 2020[a]) load tests, which it receives from a workload model service. Also, it can execute the load tests and return the tests' results.

10.1.3. Command-line Interface

For suitable interaction with the framework, we provide a CLI, which we have implemented using Spring Shell (Pivotal Software, Inc., 2020[c]). Users can execute the CLI locally, submit commands to the CLI, which will then

```
continuity:> app-id myapp
continuity (myapp@latest):> version v1

continuity (myapp@v1):> idpa
continuity/idpa (myapp@v1):> open
Opened the IDPA application model with app-id myapp
Opened the IDPA annotation with app-id myapp
continuity/idpa (myapp@v1):> app upload
Successfully uploaded application models for app-ids
    [myapp]: (details omitted)
continuity/idpa (myapp@v1):> ..

continuity (myapp@v1):> order
continuity/order (myapp@v1):> create
Created and opened a new order.
continuity/order (myapp@v1):> submit
There is no jmeter service available to produce a
    load-test!
continuity/order (myapp@v1):> submit
Submitted the order, order ID is myapp-1. For further
     actions: (details omitted)
continuity/order (myapp@v1):> wait
---
order-id: myapp-1
number: 1
max: 1
successful: true
artifacts:
  intensity: cobra/intensitiy/myapp-1
  app-id: myapp
  behavior-model:
    type: markov-chain
    link: cobra/behavior-model/myapp/all/latest?
        before=1536537600000

continuity/order (myapp@v1):> get behavior-model
```

Listing 10.1: Exemplary interaction with the CLI. Statements following the `continuity:>` prompt are user inputs, while all other lines are output by the CLI.

forward requests to the REST API of the Orchestrator. Listing 10.1 illustrates such an interaction between a user and the CLI. The user first defines an *app-id* and *version*, which identify the application they aim to test. Then, they edit the IDPA of this application (lines 4ff), create and submit an order for load test generation (lines 12ff), and retrieve the created artifact (line 32).

The CLI is structured hierarchically, corresponding to the processes that the framework supports, which we detail in the next section. For instance, the `idpa` command switches to the corresponding process (line 4), for viewing, editing, and updating IDPAs. Similarly, users can switch to the `order` process (line 12), where they can trigger the generation of load tests or other artifacts. For switching to the parent process, the `..` command can be used (line 10).

The CLI also provides auto-completion of the commands and sends feedback to the user. For instance, when an error occurs, the user receives an error message (line 16). The `help` command shows a list of all available commands or explains specific commands.

10.1.4. Essential Processes

The framework implements three essential processes, which users can apply for generating tailored load tests. First, they can manage, i.e., store and update, IDPAs for load test parameterization. Second, they can upload monitored requests or traces, which our framework clusters incrementally. Third, users can submit orders for generating tailored load tests or other artifacts. In the following, we describe these processes.

10.1.4.1. Parameterization Management

The first step a user should do when using our framework is the definition of an IDPA for the application they want to test. The framework will use this IDPA for labeling requests and parameterizing the generated load tests. For that, users can utilize the CLI, as illustrated in Listing 10.1, which uploads the IDPA application and annotation models to the respective endpoints of the IDPA service (② in Figure 10.1). Other services can then request these artifacts from the IDPA service.

For easing the creation and updating of IDPAs, we provide means for generating application models from OpenAPI specifications, as described in Section 6.5.1. Also, users can generate empty annotation templates with a single command. We demonstrate these mechanisms in detail online (H. Schulz, 2019).

The IDPA management in a dedicated service is especially valuable for evaluation purposes. For integration with continuous software engineering (CSE), as described in Section 6.6, using a source code repository, such as Git, would be a suitable alternative. However, the IDPA service can also be synchronized with a source code repository via the REST API.

10.1.4.2. Incremental Workload Model Learning

The Cobra service is responsible for the incremental workload model learning, which we describe in Section 8.4. Users can upload requests or traces, from which the framework extracts sessions, via a REST endpoint of the Orchestrator (① in Figure 10.1). Because the processing of the uploaded data can be long-lasting, the Orchestrator forwards the data asynchronously to the Cobra service via a dedicated message queue (X_b in the figure). Cobra then retrieves the required IDPA application model from the IDPA service, extracts and labels the requests (see Section 6.5.2) — applying log-based service-tailoring (see Section 7.4) if configured —, and groups the labeled requests into sessions. It stores the sessions into Elasticsearch and sends an asynchronous message to Clustinator, which clusters the sessions incrementally. Finally, Clustinator reports back to Cobra, which calculates the per-group intensities and stores them into Elasticsearch.

The Cobra service accepts multiple data formats. First, requests can be uploaded in the Apache common log format (Apache Software Foundation, 2020[c]). As many web servers use this format for request logging, the format covers many use cases. For more generic use cases, Cobra can also parse comma-separated values (CSV) files with predefined column names and one request per row. Third, Cobra can process session logs in the format used by the WESSBAS approach (Vögele et al., 2018). Finally, users can upload traces — which are required for service-tailoring — in the OPEN.xtrace (Okanović et al., 2016) format. For integration with standard application performance management (APM) tooling, we also provide a transformation from Zipkin (Zipkin, 2020) traces to OPEN.xtrace (see Section 10.2.1).

For adding contexts to the learned intensities, the Cobra service provides a REST API that accepts context records (see Section 8.4.2). These records are added to the stored intensity values, which allows querying intensities based on context definitions. Uploading contexts is synchronous because it does not require long-lasting processing.

10.1.4.3. Orchestration of Load Test Generation

The most extensive process is the generation of a load test. Instead of a load test, a user can also generate intermediate artifacts, such as a workload model. For that, they need to define an *order* containing the type of artifact to be generated, the tested application's app-id and version, a list of services for the service-tailoring, a Load Test Context-tailoring Language (LCtL) instance, and the services the framework should use for generating the artifact. Optionally, an order can also contain links to artifacts from previous generations, which the framework reuses. The CLI supports the user in the creation and submission of an order, as illustrated in Listing 10.1.

When the Orchestrator receives an order via a POST request (③ in Figure 10.1), it confirms the reception and processes the order asynchronously. The confirmation message contains a link, which the user can request for checking the processing status (④ in the figure). Then, the Orchestrator dynamically determines the services to invoke. Precisely, it applies the following steps.

1. *Creation of a recipe*: Essentially, the generation of a load test is a series of transformations. Each service registers at Eureka with the artifacts it can produce and those it needs as an input. Hence, based on the registered services, the target artifact, and the services specified in the

order, the Orchestrator computes the recipe as a list of tasks. Each task defines the (intermediate) artifact to be created and the service that should perform the transformation. As an example, the default recipe for generating a load test is a transformation of the stored sessions into a *behavior model* by Cobra, followed by a transformation into a *workload model* by WESSBAS, followed by a transformation into a *parameterized load test* by JMeter.

2. *Checking available artifacts*: If the order contains previously generated artifacts, the Orchestrator checks whether they can be reused. For instance, if a WESSBAS workload model is specified, the generation only requires a transformation to a load test, and the Orchestrator skips the other transformations.
3. *Sending tasks to services*: Next, the Orchestrator processes the first task of the recipe by sending a message to the global message queue with name *continuity.taks.global.create*. The routing key is a combination of the invoked service's name and the type of artifact to be created. Also, it sends links to the available artifacts, which the invoked service can retrieve. This prevents unnecessary sending of large artifacts.
4. *Handling finished tasks*: After the invoked service has finished the transformation, it publishes an event to the message queue with name *continuity.event.global.finished*. Hence, the Orchestrator can react and decide about the next steps. Considering the recipe, it decides whether the target artifact has been created — in this case, it sends a report to the user — or whether further tasks need to be sent to services.
5. *Reporting failures*: Individual services may fail to perform their transformation. As we explicitly designed the framework for evaluation purposes, we make failures explicit rather than applying failover mechanisms, such as typically applied in microservice applications (Newman, 2015). On failures, the services send messages to a dedicated queue. The Orchestrator reacts to these messages by stopping the recipe and reporting the failure to the user.

6. *Sending a report to the user*: When the recipe is processed entirely — or aborted due to failure — the Orchestrator sends a report to a message queue (X_a in the figure). The user can then retrieve the report via the link they have received when submitting the order (④ in the figure).

10.2. Additional Implementations

In the following sections, we describe further services we have implemented for experimentation purposes or better integration with common formats. These are a service that converts Zipkin traces into the OPEN.xtrace format and a service that mocks a system under test (SUT).

10.2.1. Zipkin to OPEN.xtrace Converter Service

Zipkin (Zipkin, 2020) is widely used for open-source APM and can, among other data, collect traces. It conforms to *OpenTracing* (2020) and *OpenCensus* (2020), which are open standards for trace collection. Hence, being able to process the traces that Zipkin collects enables the integration with many application setups. Therefore, we have implemented a converter service.

This service integrates with the framework described in the previous section as a proxy. It is implemented in Java using Spring Boot, and its source code is available on GitHub (H. Schulz, 2020c). It needs to be configured with the IP address or hostname of a Zipkin server, from which it can retrieve traces. Furthermore, it provides an endpoint for retrieving traces in the OPEN.xtrace (Okanović et al., 2016) format, which it serves by requesting traces from the Zipkin server and transforming them. Hence, when uploading traces to the framework, users can define a link to the endpoint of the converter service, which will automatically transform the Zipkin traces into a format the framework can process.

10.2.2. System under Test Mock

For experimentation, we have implemented a service that mocks an SUT during the execution of a load test. It is written in Java using Spring Boot, with the source code being available on GitHub (H. Schulz, 2020b). The service responds to all HTTP requests it receives and logs them. It buffers the requests in a queue, and consumer threads write the requests to local files asynchronously. For performance reasons, requests that do not fit into the queue are discarded and responded with a 500 error status code. Furthermore, the service provides endpoints for retrieving all logged requests and restarting the service. Hence, experiment automation scripts can execute multiple load tests in a row, while restarting the SUT mock remotely. The restarts ensure a clean environment for every test execution.

10.3. Summary

This chapter describes the implementation of our approach, which covers the concepts presented in the previous chapters. The implementation has an extensible architecture that allows adding further services with further functionality, e.g., the transformation of workload models to specific load test formats. Also, we have implemented services for transforming traces into a standard format and for mocking SUTs. We have used the implementation in our evaluation, which we provide in Part III.

Part III

Evaluation

Chapter 11

Evaluation Design

We have introduced our approach in the previous part, consisting of automated load test parameterization, service-tailoring, context-tailoring, and integration with BenchFlow (Ferme and Pautasso, 2018) addressing non-experts. In the following, we provide an extensive evaluation of all four parts. It is structured according to the work packages and approach parts. This chapter provides an overview of the studies (Section 11.1) and introduces the evaluation metrics (Section 11.2).

11.1. Overview of Evaluation

Overall, we have conducted twelve studies with ten different subjects. Table 11.1 provides an overview. As shown in the table, we have evaluated each work package separately. In this section, we summarize the evaluation per work package and highlight results. Chapters 12 to 15 provide precise evaluation methods, results, and discussions. For study replication, we provide supplementary material online (see Appendix E).

Table 11.1.: Overview of Studies Conducted

		Method						**Scale**	
Subject	**Section**	Lab experiment	Case study	Expert survey	Estimation	Formal proof	Formal analysis	quantitative	qualitative
WP1 — Load Test Parameterization									
Effort estimation models	12.1				●			●	
Neuxs (Sonatype, Inc., 2013)	12.2	●						●	
Heat Clinic (Broadleaf Commerce, LLC, 2017)	12.3	●						●	
Four industrial projects	12.4		●						●
WP2 — Service-tailoring									
–	13.1					●			
Sock Shop (Weaveworks, Inc., 2020)	13.2	●						●	●
WP3 — Context-tailoring									
SIS of Charles University, Prague (Maňásek and Tůma, 2019)	14.2						●	●	
	14.3		●	●					●
	14.4	●						●	
	14.5	●						●	
WP4 — Load Testing for Non-experts									
Project partner	15.1		●						●
Avritzer et al. (2018, 2020a)	15.2	○	●					○	●

● primary method/scale ○ study basis

11.1.1. WP1 — Load Test Parameterization

The evaluation of WP1 addresses RQ1: *How can load test parameterizations be evolved without manual intervention at test generation or execution time?* We derived effort estimation models, conducted two experimental studies to assess the representativeness of generated load tests, and performed an industrial case study to assess the expressiveness of the Input Data and Properties Annotation (IDPA).

Our results show that our approach is a suitable answer to RQ1. Separating parameterizations from generated workload models or load tests with an IDPA does not require manual intervention at test generation or execution time, because all parameterizations can be defined in advance. Besides, our approach reduces the maintenance effort from a quadratic to a linear function of time. Simultaneously, the representativeness of load tests parameterized with an IDPA is high, especially for session-dominated workloads. Session-dominated means that the order and timing of requests within a session are more relevant to the SUT than the requests' payloads. Also, due to its extensibility, the IDPA can express various parameterizations required by industrial projects.

We present the evaluation in Chapter 12.

11.1.2. WP2 — Service-tailoring

The evaluation of WP2 addresses RQ2: *How can representative load tests be tailored to specific services of a session-based application?* The studies we conducted are a correctness proof of the log-based and model-based service-tailoring algorithms we introduced in Chapter 7 and an experimental study for assessing quantitative and qualitative differences.

We found both algorithms to generate representative service-tailored load tests with differences in several aspects. In a nutshell, the model-based algorithm generates slightly more representative load tests, while the load tests of the log-based algorithm are smaller, i.e., scale better with an increasing number of endpoints. For integration with context-tailoring, model-based service-tailoring is preferable because it can tailor load tests on-demand, while log-based service-tailoring needs to be configured before learning the workload models. However, the model-based algorithm needs to approximate think time distributions, which should be resolved in future work.

Chapter 13 provides the evaluation.

11.1.3. WP3 — Context-tailoring

The evaluation of WP3 addresses RQ3: *How can representative load tests automatically be tailored to the contexts of a session-based workload?* The evaluation bases upon a publicly available dataset of the student information system (SIS) of Charles University, Prague (Maňásek and Tůma, 2019). We analyzed the workload models we incrementally learned using our approach, conducted a case study for expressiveness evaluation, which also included an expert survey, and performed two series of experiments to evaluate the representativeness of workload scenarios predicted with perfect and real forecasting.

Generally, we assess the overall framework of our approach, which comprises incremental workload model learning and on-demand load test generation based on the Load Test Context-tailoring Language (LCtL), to be suitable for context-tailored representative load testing. Notably, the clustering into user groups and the use of contexts significantly improve the representativeness of generated load tests compared to predicting the total number of users without considering contexts. However, we also identified limitations of existing work we have leveraged, especially regarding predictions of sharp spikes. Future work needs to address this issue, e.g., by integrating think times into the session clustering — which we presume to increase the accuracy of the learned workload models during spikes — and by extending forecasting tools for predicting the shapes of spikes instead of only the height.

The evaluation of WP3 is presented in Chapter 14.

11.1.4. WP4 — Load Testing for Non-experts

The evaluation of WP4 addresses RQ4: *How can we leverage automated tailored load test generation and automated load test execution for enabling load testing for non-experts?* In two case studies, we evaluated the suitability of our approach for industrial and laboratory use cases.

The case studies show that our Behavior-driven Load Testing (BDLT) approach, which integrates our load test tailoring with the BenchFlow test execution by Ferme and Pautasso (2018), has multiple use cases in industrial and laboratory contexts. The industrial case study participants rated the easy-to-understand load test descriptions and the high degree of automation positively. We also found that the BDLT language encourages collaboration — e.g., when identifying relevant load test scenarios —, which is beneficial even without automated test generation or execution. We conclude that the BDLT approach allows non-experts, such as product owners, to perform load testing or at least actively participate in it. An important lesson we learned is the requirement for customizable load tests, which the BDLT language provides via extension mechanisms. In the laboratory context, we mainly found the integration of load test generation and execution to be useful, e.g., for microservice scalability assessment.

We present the case studies in Chapter 15.

11.2. Metrics

In our evaluation, we utilize several metrics for assessing different properties of generated load tests. This section summarizes these metrics. It contains a metric that can be used for determining the manual effort required for maintaining load tests (Section 11.2.1), five metrics describing different aspects of the representativeness of a workload based on request and session characteristics (Sections 11.2.2 to 11.2.6), and a metric defining the minimum required test execution duration (Section 11.2.7).

The following sections are revised versions of the below publications as follows:

- *Sections 11.2.1 and 11.2.2*: H. Schulz, A. van Hoorn, and A. Wert (2020c). "Reducing the Maintenance Effort for Parameterization of Representative Load Tests Using Annotations." In: *Journal of Software Testing, Verification and Reliability* 30.1, Section 5.2
- *Sections 11.2.5 and 11.2.7*: H. Schulz, T. Angerstein, D. Okanović, and A. van Hoorn (2019a). "Microservice-tailored Generation of Session-based Workload Models for Representative Load Testing." In: *Proceedings of the 27th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2019)*. IEEE Computer Society, pp. 323–335, Section 5.C
- *Sections 11.2.3, 11.2.4, and 11.2.6*: H. Schulz, D. Okanović, A. van Hoorn, and P. Tůma (2021). "Context-tailored Workload Model Generation for Continuous Representative Load Testing." In: *Proceedings of the 12th ACM/SPEC International Conference on Performance Engineering (ICPE 2021)*. To appear. ACM, Section 5.B

11.2.1. Maintenance Effort for Parameterization

To measure the maintenance effort required for creating and evolving load test parameterizations, we introduce a metric θ. It is based on the change types defined in Section 6.4.2 and summarizes the effort for applying changes to individual elements. Even though the change types refer to elements of an IDPA, we presume we also can apply it to manually parameterized load tests, because the same logical operations are done. This presumption is especially valid as we only use the metric for qualitative comparison.

Each change $c = (t_c, o_c, k_c)$ consists of the changed element's type t_c, the change operation $o_c \in \{\mathrm{A}, \mathrm{C}, \mathrm{R}\}$ denoting add, change, and remove operations, and the number of changes k_c of that type and operation. The types of possibly changed elements are `EndpointAnnotation`, `ParameterAnnotation`, and subtypes of `Input`. We denote the effort introduced by one change as $\gamma(t_c, o_c)$. Hence, $\gamma(t_c, o_c)$ depends on the complexity of applying a particular operation to a particular IDPA element type. Given C holds all changes introduced to an IDPA, we can express the resulting formula for the overall effort metric as follows:

$$\theta(C) = \sum_{c \in C} k_c \cdot \gamma(t_c, o_c) \qquad (11.1)$$

Calculating precise values for θ — e.g., measured in person-hours — requires extensive empirical studies for determining the precise values of $\gamma(t_c, o_c)$. This is out of the scope of this dissertation, and we leave it for future work. Instead, we still can use the metric for qualitative comparisons. In doing so, we consider the $\gamma(t_c, o_c)$ as abstract but constant variables, based on which we derive formulas describing the asymptotic effort. The results of empirical studies can then be used to parameterize the formulas and compute precise values.

11.2.2. Representativeness of Parameterized Load Tests

In our evaluation, we need to assess the representativeness (see Section 3.2.3) of a generated load test compared to a reference workload. In the cases where we want to evaluate our approach to automated parameterization (see Chapter 6), we need to pay special attention to the mix of called endpoints and their responses. For that, we introduce a metric ρ comparing the measurements of the load test execution with a reference measurement. The closer the load test measurement is to the reference measurement, the better is the value of ρ.

We designed ρ to have the following characteristics.

1. ρ has an optimum at 0, i.e., smaller values indicate better representativeness.
2. ρ can be used to compare multiple load test executions with the same reference, each resulting in one scalar value.
3. ρ is based on the request rates per endpoint.
4. Because wrong input data often lead to HTTP error response codes (e.g., 400 or 500), the metric accounts for that.

5. ρ considers the percental deviation of the request rates to all endpoints equally,
6. except for minimal request rates, which have a lower impact on the metric value, because high percental deviations are likely in this case.
7. ρ accounts for workload deviations over time, i.e., while the request rates and response codes might be met well in summary after the whole test, there can be more considerable differences during some periods.

To calculate the metric value, we consider endpoints e_i of the system under test (SUT) and the respective request rates $x_{i,c}$ to endpoint e_i with response code c, e.g., $c \in \{200, 302, 400, 500\}$. For further calculations, we arrange the request rates in matrix $X_{(\cdot)} := (x_{i,c})$. For instance, a part of the matrix resulting from one of our experiments is the following.

$$\begin{array}{c|cccc} & 200 & 302 & 400 & 500 \\ \hline e_{home} & 81.3 & 0 & 0 & 0 \\ e_{add} & 49.4 & 1.5 & 0 & 0 \\ e_{logout} & 0 & 70.2 & 0 & 0 \end{array} \tag{11.2}$$

In the following, let X_{ref} be the reference measurement. Let X_{gen} furthermore be the measurement of a generated load test execution. The Frobenius distance $\left\|X_{\text{ref}} - X_{\text{gen}}\right\|_F$ constitutes a metric conforming characteristic 1 to 4. To meet characteristic 5 as well, we normalize the measurements with a matrix

$$\mathcal{N} := \operatorname{diag}(X_{\text{ref}} \cdot \mathbb{1})^{-1}, \quad \mathbb{1} = (1, \ldots, 1)^T \tag{11.3}$$

To meet characteristic 6 — i.e., reduced impact of small request rates —, we introduce a weight function for the request rates. We want to have low weights on low request rates and high weights on high request rates with an asymptotic value of 1. Therefore, we use the logistic function

$$w(x) = \frac{1}{1 + e^{-kx}\left(\frac{1}{w(0)} - 1\right)} \tag{11.4}$$

To determine the steepness k, we define values of w. We define $w(0) := 0.01$ and, because we only want to have low weights on the lowest request rates, $w(10) := 0.9$. These values result in $k = 0.6792$. For $\boldsymbol{x} = (x_1, \ldots, x_n)^T$, we define $w(\boldsymbol{x}) := (w(x_1), \ldots, w(x_n))^T$ and obtain a weight matrix:

$$\mathcal{W} := \operatorname{diag}(w(X_{\text{ref}} \cdot \mathbb{1})), \quad \mathbb{1} = (1, \ldots, 1)^T \tag{11.5}$$

Then, we define the representativeness metric ρ for the measurement X_{gen} of a generated load test as

$$\rho(X_{\text{gen}}) := \left\| \mathcal{N} \cdot \mathcal{W} \cdot (X_{\text{ref}} - X_{\text{gen}}) \right\|_F \tag{11.6}$$

Finally, for accounting for the factor of time as requested by characteristic 7, we will consider the metric for a small unit of time like one minute and calculate the cumulative sum as an overall measure for the whole test. For distinguishing between them, we use ρ as a symbol for the per-minute representativeness and abbreviate the cumulative sum with $\sum \rho$.

To obtain an SUT-specific metric baseline, we execute one load test p times and retrieve measurements X_j, $j = 1, \ldots, p$. Then, we calculate $\rho(X_j)$, $j \geq 2$ per minute using X_1 as the reference. We calculate the mean μ_ρ and standard deviation σ_ρ from the resulting measures and use $\mu_\rho \pm 3\sigma_\rho$ as a baseline. $t\mu_\rho \pm 3\sqrt{t^2\sigma_\rho^2}$ defines a baseline for $\sum \rho$ after t minutes.

11.2.3. Representativeness of Request Mixes

The ρ metric introduced in the previous section considers both the total request rates and the request mix, i.e., the relative frequencies of the endpoints called. However, there are cases where we are interested in the total request rates and request mix separately. Therefore, we define the $\overline{\rho}$ metric, which has the same characteristics as ρ, except for characteristic 3, which we replace with, *$\overline{\rho}$ is based on the relative frequencies per endpoint called.*

For calculating the metric, we apply a slightly modified version of Equation (11.6). Let $\overline{X}_{\text{gen/ref}} := X_{\text{gen/ref}} / \|X_{\text{gen/ref}}\|_1$ be the request mixes of the test execution and the reference. Let furthermore $\overline{\mathcal{N}} := \operatorname{diag}(\overline{X}_{\text{ref}} \cdot \mathbb{1})^{-1}$ be the normalization matrix. Then, $\overline{\rho}$ is calculated as below. Remarkably, the weight matrix $\mathcal{W}$ is similar to Equation (11.6), i.e., it is calculated based on the absolute request rates.

$$\overline{\rho}(X_{\text{gen}}) := \left\| \overline{\mathcal{N}} \cdot \mathcal{W} \cdot (\overline{X}_{\text{ref}} - \overline{X}_{\text{gen}}) \right\|_F \tag{11.7}$$

11.2.4. Request Gap

For an isolated assessment of the representativeness of the total request rate, we introduce the request gap. Given the number x_{gen} of requests submitted by a load test and the number x_{ref} of requests of a reference workload, we define the absolute request gap as $|x_{\text{gen}} - x_{\text{ref}}|$. For a single test execution, we calculate the cumulative absolute request gap per small time unit, e.g., per minute. For comparing multiple executions with each other, we utilize the relative average request gap per small time unit. Similar to the ρ metric, we can calculate a baseline from multiple executions of the same load test. This baseline describes the normal request gap variation.

11.2.5. Representativeness of Microservice Workloads

Measuring the representativeness of a session-based workload arriving at a microservice requires a specific metric. While we can characterize the user sessions for examining a user-faced service, other services receive requests not explicitly bundled in sessions. Thus, we cannot use session information for a representativeness metric. However, we still want to cover variations in the workload stemming from the overlay of several user sessions. Therefore, we use the arrival rate of requests or its reciprocal, the inter-arrival time. Similar to the ρ metric (see Section 11.2.2), we consider a reference and a generated workload. The metric is a distance between the two workloads.

Given we collected two samples $X_{\text{ref},e}$ and $X_{\text{gen},e}$ of inter-arrival times for an endpoint e for the reference and generated workload, we calculate the Kolomogorov-Smirnov statistic D_e for the significance level α. D_e is a measure for the distance of the inter-arrival time distribution for e. The Kolmogorov-Smirnov test rejects a null hypothesis $H_0 : F_{X_{\text{gen},e}}(x) = F_{X_{\text{ref},e}}(x)$ if the following holds for the critical value $c(\alpha)$ (Massey Jr., 1951):

$$D_e > c(\alpha) \cdot \sqrt{\frac{|X_{\text{ref},e}| + |X_{\text{gen},e}|}{|X_{\text{ref},e}| \cdot |X_{\text{gen},e}|}}$$

Hence, we yield an aggregate measure for all endpoints $\mathcal{E}$ involved in one load test by calculating the weighted average of the D_e:

$$D := \frac{1}{|\mathcal{E}|} \sum_{e \in \mathcal{E}} D_e \cdot \sqrt{\frac{|X_{\text{ref},e}| \cdot |X_{\text{gen},e}|}{|X_{\text{ref},e}| + |X_{\text{gen},e}|}}$$

We name D the *workload distance* and use it as representativeness measure in evaluations with microservice applications.

11.2.6. Session Length and Duration

Assessing the representativeness of session-based workloads should not only consider the requests but also characteristics of the sessions. We reuse the session length and session duration metrics from existing work (Vögele et al., 2018). The session length is the number of requests submitted in one session, while the session duration describes the time elapsed between the first and last request of a session.

For aggregating the individual session lengths or durations to a measure for the overall test execution, we use the Kolmogorov-Smirnov statistic. For inter-test comparison, we normalize it in the same way as in the previous section. Notably, Vögele et al. (2018) have found load tests with Markov-chain-based workload models to generate sessions that tend to be longer than real sessions.

11.2.7. Load Test Duration

One aspect of a load test or performance test, in general, is the time it needs to be executed. Existing works decide whether to stop a test by detecting a stable state of performance measures at runtime (Alghamdi et al., 2016). Here, we follow a similar approach.

Presuming a steady-state load, aggregate measures such as the median response time at test end are of interest for performance evaluation. Therefore, we determine the required test duration by comparing the median response time calculated at a particular timestamp with the final value. Given a sample of median response times $(x_1^{(e)}, \dots, x_n^{(e)})$ per time unit — e.g., one second — for endpoint e, we calculate the distance to the finally determined median response time $x_n^{(e)}$ at index i:

$$\widetilde{x}_i^{(e)} := \frac{|x_i^{(e)} - x_n^{(e)}|}{x_n^{(e)}}$$

Then, we define the required test duration for $\beta = 0.01$ and tested endpoints $\mathcal{E}$ as

$$\max_{e \in \mathcal{E}} \left(\min\{i \mid \forall j \geq i : \widetilde{x}_j^{(e)} \leq \beta\} \right)$$

Chapter 12

Evaluating Automated Load Test Parameterization

In Chapter 6, we describe our approach to the automated parameterization of load tests. We introduced the Input Data and Properties Annotation (IDPA) for decoupling parameterizations from load tests and removing the manual effort from the test generation. In this chapter, we evaluate this approach in four different studies relating to the sub-questions RQ1.3 to 1.6 of RQ1: *How can load test parameterizations be evolved without manual intervention at test generation or execution time?*

The first study analyzes the reduction of the maintenance effort using estimation models. In an experimental study with Nexus (Sonatype, Inc., 2020[a]) — a build artifact management system —, we evaluate the impairment of the representativeness when using an IDPA instead of the original parameterization. A second experimental study evaluates how much the representativeness of a generated load test can be improved by proper parameterization with an IDPA. As system under test (SUT), we utilized the

Heat Clinic (Broadleaf Commerce, LLC, 2017). Finally, we present a case study assessing the expressiveness of the IDPA for the requirements of four industrial projects. Details about all studies are available online as part of the replication package (H. Schulz et al., 2019f).

According to the evaluation, our IDPA approach can be used for parameterizing generated load tests with low manual effort. The effort estimation models show that it significantly reduces the cumulative effort for maintaining load tests. When parameterizing load tests directly without an IDPA, the cumulative effort is a quadratic function of time for a typical mix of API changes. Our approach reduces it to a linear function of time. The two experimental studies with Nexus and the Heat Clinic reveal that the parameterization by an IDPA preserves the representativeness of the load test under particular circumstances. Notably, the workload needs to be dominated by the type and order of requests — i.e., the user sessions — rather than the input data. For Nexus, the IDPA slightly decreased the representativeness compared to the original workload, because the input data partially influenced the SUT's behavior. However, we could not observe such a decrease with the Heat Clinic. In contrast, the IDPA restored the representativeness while an unparameterized generated load test resulted in significantly different behavior of the SUT and, hence, in falsified test results. By making use of the extension mechanisms, we were able to express the parameterizations required by the industrial projects. However, some of them appeared to be cumbersome. Therefore, we added most of the missing concepts and more concise versions of the cumbersome ones to the IDPA after the study (see Section 6.3.2).

The chapter is structured as follows. In Sections 12.1 to 12.4, we present the four studies, including their methods and results. In Section 12.5, we discuss the results with respect to the addressed research questions. Finally, Section 12.6 summarizes potential threats to the validity of our work.

This chapter is a revised version of Section 5 of the below publication, in which we have published the studies and discussions presented in the following in advance:

- H. Schulz, A. van Hoorn, and A. Wert (2020c). "Reducing the Maintenance Effort for Parameterization of Representative Load Tests Using Annotations." In: *Journal of Software Testing, Verification and Reliability* 30.1

12.1. Maintenance Effort Estimation Models

In the following, we analyze the difference of the maintenance effort required for evolving an IDPA compared to not using an IDPA, i.e., repeatedly parameterizing a load test directly. RQ1.3 formulates this: *To which degree can we reduce the maintenance effort for the evolution of load test parameterizations if the API changes?*

As described in Section 11.2.2, determining the precise maintenance effort requires empirical studies related to the work of Benestad et al. (2010), which we leave for future work. Here, we utilize the θ metric introduced in Section 11.2.1 for deriving formulas we can compare asymptotically. We first introduce our method for deriving the formulas and then present the results of the analysis.

12.1.1. Analysis Method

The goal of our analysis is to compare the asymptotic effort required for evolving IDPAs compared to parameterizing a load test repeatedly. In doing so, we assume that the amount and distribution of changes introduced to the target system's API are constant over time. Even in the case that this assumption does not hold, it affects both parameterization approaches equally and thus, allows for a fair comparison. As a second assumption, we consider the effort to change an IDPA element and the respective load test element to be equal. For instance, we assume the effort to map an `Input` to a parameter using a `ParameterAnnotation` is equal to setting the input value of a JMeter request parameter. This assumption is valid, as we are only comparing the asymptotic behavior.

For deriving the formulas, we consider multiple iterations. In each iteration, a new load test is generated and parameterized. Between two iterations, several API changes can be introduced. In a first step, we derive formulas depending on the average amount of introduced changes per iteration. These formulas will be different for IDPA evolution and direct load test parameterization. In a second step, we then determine the amount of different IDPA or load test element changes based on the frequencies presented by S. Wang et al. (2014) and the API composition of Nexus (Sonatype, Inc., 2020[a]), which we used in the experiment presented in the next section. As a result, we obtain more concrete formulas only depending on a few variables, which we can compare.

12.1.2. Results

The following analysis uses the θ effort metric and corresponding notations introduced in Section 11.2.1. For convenience, we introduce several terms. We use *change* as a synonym for a change to an IDPA or a load test parameterization. In contrast to that, an *API change* denotes a change to the API of the SUT. With θ_0, we denote the effort to create an initial load test. This effort will be the same for both parameterization approaches because it requires adding the same parameterizations. With C, we denote a set of changes introduced in one iteration while $\overline{C}$ denotes all possible changes, i.e., all combinations (t_c, o_c) of possible values of the changed element's type t_c and the change operation o_c. We furthermore define $n := \sum_{c \in C} k_c$ as the number of changes per iteration. We define $\alpha(t_c, o_c)$ as the average number of changes per type and operation to be applied because of one API change. In the first step, we consider $\alpha(t_c, o_c)$ as an abstract variable, and we parameterize it in the second step based on the average distribution of API changes.

Using these definitions, we can express the effort required for evolving an IDPA from iteration $i-1$ to i depending on n and $\alpha(t_c, o_c)$. Furthermore, we can abstract from the precise efforts $\gamma(t_c, o_c)$ by defining the maximum value $\Gamma := \max(\gamma(t_c, o_c))$:

$$\begin{aligned}
\theta_{\mathrm{IDPA}}(C,i) &= \sum_{c\in C} k_c \cdot \gamma(t_c,o_c) \\
&\approx n \cdot \left(\sum_{c\in\overline{C}} \alpha(t_c,o_c) \cdot \gamma(t_c,o_c) \right) \\
&\leq n \cdot \Gamma \cdot \sum_{c\in\overline{C}} \alpha(t_c,o_c) \\
&= \text{const.}
\end{aligned}$$

Hence, the upper estimate of θ_{IDPA} is constant and notably does not depend on the iteration i.

The main difference when directly parameterizing a load test is in the set of changes applied in each iteration. In contrast to using an IDPA, all parameterizations have to be redefined from scratch. However, there are no remove or change operations. Instead, the corresponding parameterizations can be left out or defined correctly according to the new version. As a consequence, when reusing the $\gamma(t_c,o_c)$ of the IDPA, the θ formula is different. First, we need to apply all changes from the previous iteration again plus the changes with an add operation. However, we need to subtract all remove operations but with the effort of adding a parameterization of the particular type. Then, we can estimate the formula with the $\alpha(t_c,o_c)$ and the minimum effort $\gamma := \min(\gamma(t_c,o_c))$. In the formula, we use $\overline{T}$ as the set of all possibly changed types t_c. The effort for direct load test parameterization (dltp) in iteration i is then:

$$
\begin{aligned}
\theta_{\text{dltp}}(C,i) &= \theta_{\text{dltp}}(C,i-1) + \sum_{\substack{c\in C\\ o_c=\text{A}}} k_c \cdot \gamma(t_c,\text{A}) - \sum_{\substack{c\in C\\ o_c=\text{R}}} k_c \cdot \gamma(t_c,\text{A}) \\
&\approx \theta_{\text{dltp}}(C,i-1) + n \cdot \sum_{t_c\in T} (\alpha(t_c,\text{A}) - \alpha(t_c,\text{R})) \cdot \gamma(t_c,\text{A}) \\
&\geq \theta_{\text{dltp}}(C,i-1) + n \cdot \gamma \cdot \sum_{t_c\in T} (\alpha(t_c,\text{A}) - \alpha(t_c,\text{R})) \\
&= \theta_0 + i \cdot n \cdot \gamma \cdot \left(\sum_{t_c\in T} \alpha(t_c,\text{A}) - \sum_{t_c\in T} \alpha(t_c,\text{R}) \right)
\end{aligned}
$$

We can see that in contrast to using an IDPA, the effort depends on the iteration i. Furthermore, the relation of the average number of element additions $\alpha(t_c,\text{A})$ and removals $\alpha(t_c,\text{R})$ is the main influencing factor of the overall effort.

For sustainably comparing θ_{IDPA} and θ_{dltp}, we parameterize the formulas with the actual values of $\alpha(t_c,o_c)$. For that, we base on the API change frequencies determined by S. Wang et al. (2014). As illustrated in Figure 12.1, we can translate the API change frequencies to IDPA change frequencies. For that, we first translate them to API change frequencies based on our API change types introduced in Section 6.4.2. Then, we determine the IDPA changes that would be required and translate the frequencies. Hence, each number attached to a change operation and element type denotes the number of respective changes to be applied for one API change. These numbers are equal to $\alpha(t_c,o_c)$. Because additions or removals of `EndpointAnnotations` can also require additions or removals of `ParameterAnnotations` and `Inputs`, we base on the average number of parameters per endpoint based on Nexus. It has 135 endpoints with 149 parameters overall and hence, 1.1037 parameters per endpoint on average.

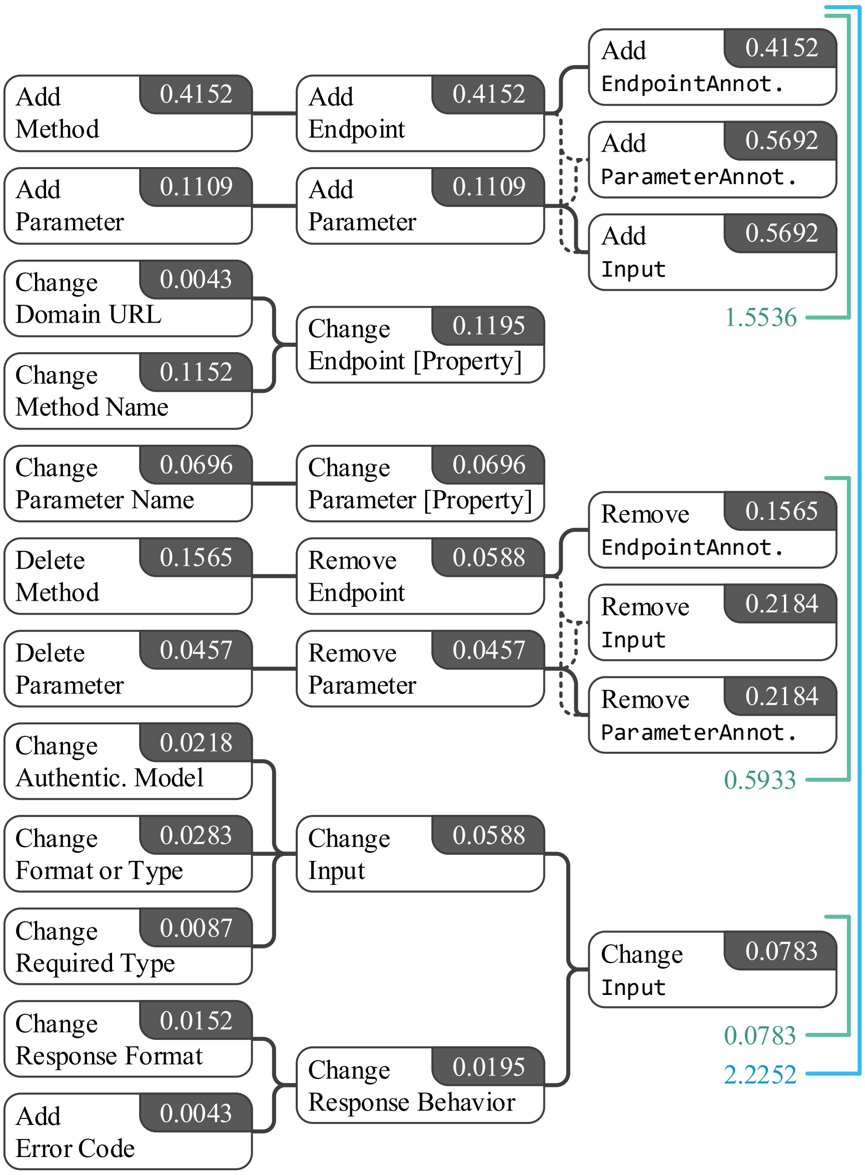
Setting the $\alpha(t_c,o_c)$ to the determined values results in the following formulas of θ_{IDPA}. Also, we calculate the cumulative sum, which represents the overall effort spent on parameterizing load tests until an iteration p. As the average number of changes per API change is 2.2252, it is:

$$
\begin{aligned}
\theta_{\text{IDPA}}(C,i) &\leq n \cdot \Gamma \cdot 2.2252 \\
\sum_{i=0}^{p} \theta_{\text{IDPA}}(C,i) &\leq \theta_0 + p \cdot n \cdot \Gamma \cdot 2.2252
\end{aligned}
$$

We do the same for θ_{dltp}. The average number of additions per API change is 1.5536, while the average number of removals is 0.5933. Hence, the difference is 0.9603, resulting in the following formulas:

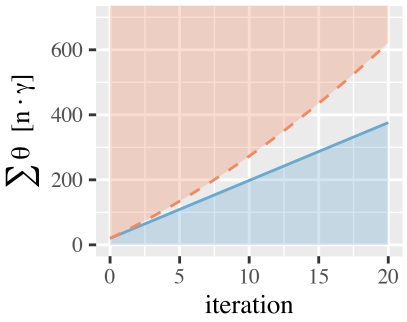
$$
\begin{aligned}
\theta_{\text{dltp}}(C,i) &\geq \theta_0 + i \cdot n \cdot \gamma \cdot 0.9603 \\
\sum_{i=0}^{p} \theta_{\text{dltp}}(C,i) &\geq (p+1)\theta_0 + \frac{1}{2}p(p+1) \cdot n \cdot \gamma \cdot 0.9603
\end{aligned}
$$

We can see that the cumulative effort is linear when using an IDPA while it is quadratic when parameterizing load tests directly. As illustrated in Figure 12.2, the precise relation of the two approaches depends on the initial effort θ_0 and the relation of the effort maximum Γ and minimum γ. The figure shows the upper estimate of the cumulative effort with IDPA and the lower estimate of the cumulative effort with direct parameterization. The values we chose are arbitrary but show the three different scenarios we identified. The first scenario is that in the beginning, there is no significant difference between the two approaches, which, however, increases with more iterations (Figure 12.2a). In the second scenario, the effort for direct parameterization is significantly higher right from the first iteration (Figure 12.2b). Finally, the effort of direct parameterization could be lower than with an IDPA in the beginning but, finally, become larger (Figure 12.2c). Regardless of the scenario, the parameterization using an IDPA will always have less effort in the long term because of the linear asymptotic behavior compared to the quadratic behavior of the direct load test parameterization.

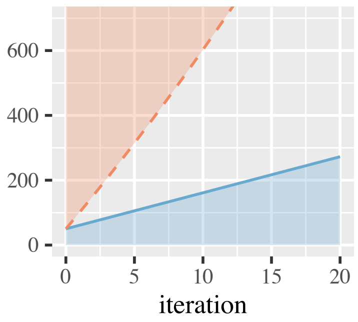


Add Method 0.4152
Add Parameter 0.1109
Change Domain URL 0.0043
Change Method Name 0.1152
Change Parameter Name 0.0696
Delete Method 0.1565
Delete Parameter 0.0457
Change Authentic. Model 0.0218
Change Format or Type 0.0283
Change Required Type 0.0087
Change Response Format 0.0152
Add Error Code 0.0043
Add Endpoint 0.4152
Add Parameter 0.1109
Change Endpoint [Property] 0.1195
Change Parameter [Property] 0.0696
Remove Endpoint 0.0588
Remove Parameter 0.0457
Change Input 0.0588
Change Response Behavior 0.0195
Add EndpointAnnot. 0.4152
Add ParameterAnnot. 0.5692
Add Input 0.5692
1.5536
Remove EndpointAnnot. 0.1565
Remove Input 0.2184
Remove ParameterAnnot. 0.2184
0.5933
Change Input 0.0783
0.0783
2.2252

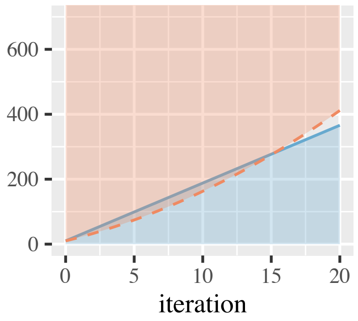
Figure 12.1.: Changes to an IDPA resulting from API changes. The numbers denote the number of changes of the respective type required for one API change according to the distribution determined by S. Wang et al. (2014).



(a) $\theta_0 = 20, \Gamma \geq 8\gamma$



(b) $\theta_0 = 50, \Gamma \geq 5\gamma$



(c) $\theta_0 = 10, \Gamma \geq 8\gamma$

Figure 12.2.: Upper estimate of the cumulative θ_{IDPA} (solid line) and lower estimate of the cumulative θ_{dltp} (dashed line) measured in multiples of $n \cdot \gamma$ over the number of iterations.

12.2. Experimental Study with Sonatype Nexus

In this study, we address RQ1.4: *How much does the parameterization by our approach impair the representativeness of a load test?* For that, we executed an experiment series with Nexus (Sonatype, Inc., 2020[a]), which is a widely used open-source build artifact management system. We generated a representative load test based on real-world access logs and parameterized it using an IDPA. Then, we can compare the results of the original test to the parameterized test results. In the following, we first describe the experimental method and then present the experiment results.

12.2.1. Experimental Method

In the following, we describe the method of this study. We describe the SUT, the prerequisites, and the experiment process and setup.

12.2.1.1. System Under Test

We utilized the Nexus mentioned above as the SUT in our experiments. Its intent is the management of build artifacts such as Maven Project Object Models (POMs) or Java Archives (JARs). Its most famous public instance is Maven Central (Sonatype, Inc., 2020[c]), which world-wide software development projects use. The program code of Nexus is open-source and available on GitHub (Sonatype, Inc., 2015). In our experiments, we utilized Nexus version 2.11, which is available as a Docker image on Docker Hub (Sonatype, Inc., 2020[b]). The Nexus API we based on comprises 135 endpoints for uploading, browsing, searching, and retrieving build artifacts. Besides, there are endpoints for user management.

12.2.1.2. Prerequisites

Before executing the actual experiments, we needed to meet several prerequisites. First, we required a representative load test for Nexus. For that, we recorded the access logs of the publicly available Nexus instance of an IT company (Novatec Consulting GmbH, 2020[b]) within three months. Our goal was to replay the same load that happened on this Nexus instance. For increasing the workload and, thus, increasing the reliability of the experiment results, we proceeded as follows. First, we split the logs into sessions based on the client IP address and interaction pauses of at least 30 minutes. Then, we randomly sampled and concatenated sessions for 100 concurrent threads until each thread lasted at least 30 minutes. When concatenating two sessions, we added a wait time of 5 minutes. A load test can replay

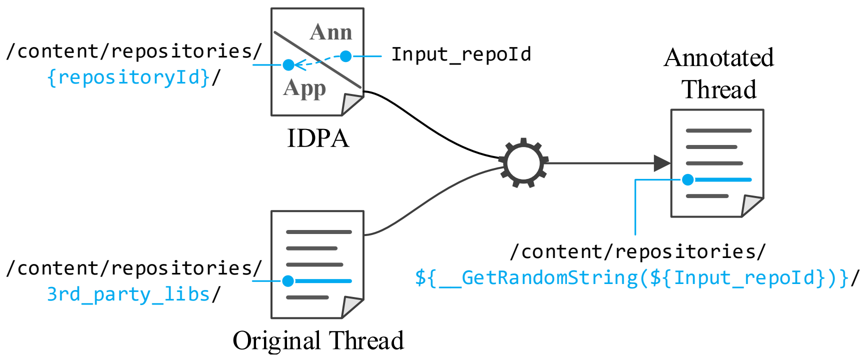


Figure 12.3.: Annotation of a thread using the IDPA.

the resulting threads, representing a varying and representative load that is high enough for reliable comparison. For repeated execution of different workloads, we generated 20 sets of 100 threads each. In the following, we refer to these threads as the *original*.

As a second prerequisite, we derived an IDPA application model from the official Nexus API description (Sonatype, Inc., 2013) and the access logs. Furthermore, we extracted all requested artifacts and other parameter values from the logs and stored it into an IDPA annotation as `DirectListInputs` and `CsvInputs`. In addition, we used a `JsonInput` for the rarely occurring POST requests. Hence, we aimed at parameterizing the load tests similarly as the original requests. For that, we annotated the previously generated threads as illustrated in Figure 12.3. Given a request of an original thread, we mapped it to an IDPA `Endpoint` by comparing the request method and path. Then, we constructed the path to be parsed by JMeter (Apache Software Foundation, 2020[a]) by replacing the parameter placeholders and query string values — e.g., *repositoryId* in the figure — with the value strings of the corresponding `Inputs` (see Section 6.5.3.2) — e.g., *Input_repoId* in the figure. Finally, we used this newly constructed path as a replacement of the original path. We name the resulting threads *annotated*. Furthermore, we stored all artifacts that were successfully requested in the access logs into a

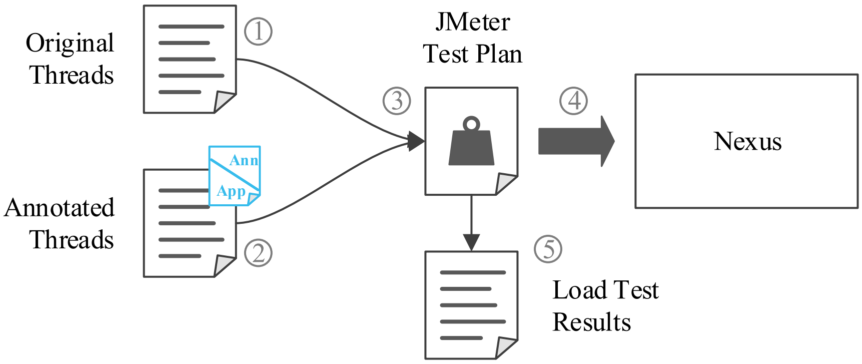


Figure 12.4.: One experiment iteration with Nexus.

separate file for populating the Nexus for the experiments. As a consequence, we could execute the original access logs without modification against our test instance of Nexus and result in the same responses as in the logs.

The last prerequisite was the definition of load tests that replay the generated sessions. For that, we used JMeter. We utilized 100 basic loops, which each have their thread as generated above and send the defined requests with the defined wait times. For the parameterized threads, we transformed the IDPA to JMeter test plan elements so that the parameterized requests of the threads use the specified input data. All threads, JMeter load tests, and the IDPA are part of the replication package (H. Schulz et al., 2019f).

12.2.1.3. Experiment Process

We performed the experiments as shown in Figure 12.4. We executed 20 iterations, which each used one of the 20 sets of original threads ① and the corresponding annotated threads ②. We merged them with the defined JMeter tests ③ and executed the two tests against a dedicated Nexus instance sequentially for 30 minutes each ④. Finally, we collected the JMeter results for comparison ⑤ and continued with the next iteration. Furthermore, for determining a baseline of the ρ metric, we executed the first iteration 20 more times in both the original and annotated variant. Before each load test

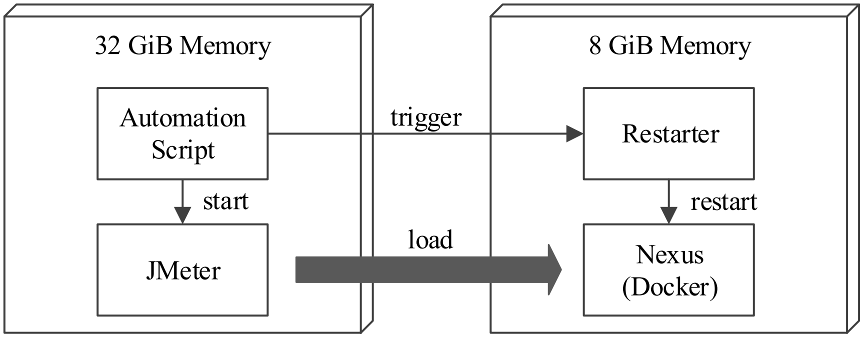


Figure 12.5.: Experiment setup with Nexus.

execution, we restarted and redeployed Nexus and populated it using the artifacts extracted from the access logs.

12.2.1.4. Experiment Setup

For executing the experiments, we utilized two machines, as illustrated in Figure 12.5. The two machines were connected with a 1 Gbit switch. Both had an Intel® Xeon® CPU E5620 with 2.40 GHz clock frequency, 4 cores, and 8 threads. The first machine had 8 GiB memory and hosted Nexus in a Docker container and a lightweight Spring Boot (Pivotal Software, Inc., 2020[b]) service offering a REST API for restarting Nexus. The second machine had 32 GiB memory and hosted the JMeter load driver and a shell script process that ran the experiment series automatically.

12.2.2. Results

In the following, we review the results of the experiment series with Nexus. The raw data and the analysis scripts are available online as part of the replication package (H. Schulz et al., 2019f). First, we consider the first iteration individually. Figure 12.6 shows the requests per minute divided by the endpoint and response code. We can see that the number of requests varies over time. The most frequent response code is 404 (Not Found),

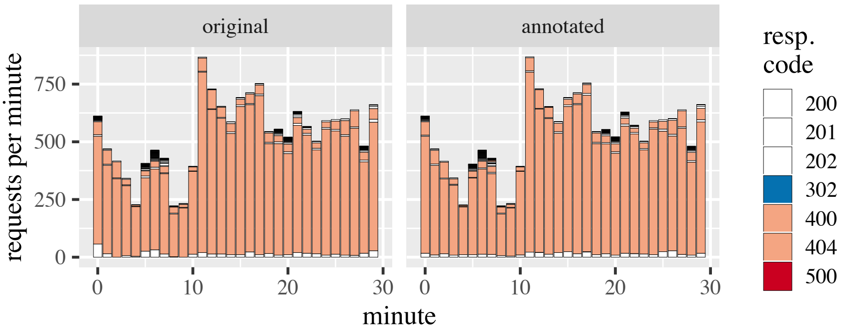
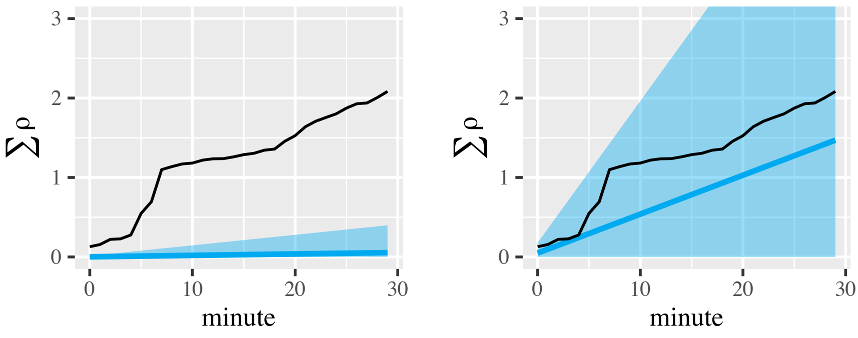


Figure 12.6.: Requests per minute divided by endpoint and response code for the first experiment iteration.

which we attribute to clients that are checking whether a particular artifact is present. On a first sight, there is no visual difference between the request rates of the original and the annotated results.

For analyzing potential differences further, we calculate the ρ metric. Additionally, we calculate a baseline for the ρ metric using the experiment series we executed for this purpose. For both the original and annotated tests, we consider the first iteration as a reference and calculate the ρ metric for each of the remaining 20 executions per minute. For the resulting $20 \cdot 30$ values of ρ, we calculate the mean and the standard deviation, which describe the mean error and variance of the results of the same test. These are $\mu_{\rho,\text{orig}} = 0.0018$ and $\sigma_{\rho,\text{orig}} = 0.0038$ for the original baseline, and $\mu_{\rho,\text{ann}} = 0.0490$ and $\mu_{\rho,\text{orig}} = 0.0431$ for the annotated baseline. We then use $\mu_{\rho,\cdot} \pm 3\sigma_{\rho,\cdot}$ as the baseline.

Figure 12.7 shows the cumulative $\sum \rho$ metric per minute calculated for the annotated results with the original results as reference, compared to the two baselines. We can see that ρ is higher than the original baseline. Hence, the representativeness is impaired, which we cannot explain with the natural variation of the original test results. However, for the annotated baseline, it is only slightly higher than $\mu_{\rho,\text{ann}}$ and lies inside the range of three standard deviations. Hence, we can explain most of the inaccuracy

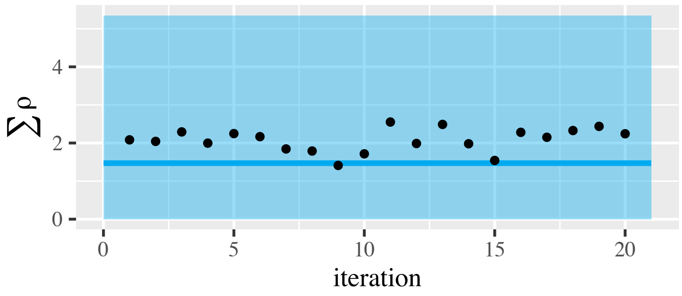


(a) Original baseline. (b) Annotated baseline.

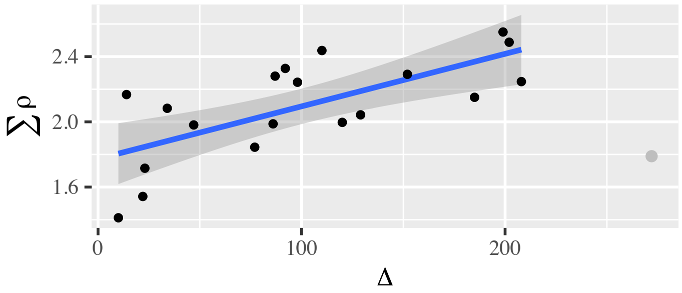
Figure 12.7.: $\sum \rho$ metric for the first iteration vs. $\mu_{\rho,\cdot} \pm 3\sigma_{\rho,\cdot}$ for the original and annotated baseline.

with the natural variations of the annotated test results, which is higher than for the original test. We explain this finding by the difference in the original and annotated JMeter tests. While the original test only replays the recorded requests, the annotated test also loads CSV files, which serve as feeders for the parameterized requests. Before each request, it loads a new line of each CSV file. Hence, it introduces small delays, which can slightly change the inter-request timings. This effect especially impairs the ρ metric when there are requests of many different endpoints, such as between minutes 5 and 7 in the first iteration, because some requests can be counted in a different minute. ρ is relatively high during this time range, as we can see in the sharp increase of the cumulative plot. The fact that most of the highest values of our baseline calculation were at minute 7 supports this hypothesis.

The remaining iterations 2 to 20 produce similar results compared to the first iteration, as illustrated in Figure 12.8a. We show the cumulative $\sum \rho$ metric at the end of the test per iteration compared to the annotated baseline. Except for iteration 9, ρ is slightly higher than $\mu_{\rho,\text{ann}}$ but still inside the baseline. Iteration 9 is even less than the baseline. We can conclude that most of the inaccuracy of the annotated test results can be explained with the natural variation. However, because of the high amount



(a) Per iteration vs. $\mu_{\rho,ann} \pm 3\sigma_{\rho,ann}$.



(b) Correlated with the difference of 404 response codes between the original and annotated tests (Δ).

Figure 12.8.: $\sum\rho$ at test end.

of iterations with ρ values higher than the baseline mean, we also conclude that the representativeness is slightly but systematically impaired. As already mentioned before, the most frequent response code is 404, which Nexus returns if a requested artifact is not present. Hence, if the annotated tests request a different amount of artifacts that are not present, the number of 404 responses will differ from the original tests, and ρ will increase. Therefore, we correlate the difference in the number of 404 responses between the original and the annotated test — Δ in the following — with $\sum\rho$, as shown in Figure 12.8b. Except for one outlier — which is iteration 8 —, a clear trend can be seen. The trend is supported by a fitted linear model $\sum\rho = \beta_1 + \beta_2\Delta$, which results in $\beta_1 = 1.7730$, $\beta_2 = 0.0032$, and deviance of 0.8594. We explain this finding by an imperfect IDPA annotation, which parameterized the requests slightly different than it was originally. We used the overall distribution of requested artifacts in the IDPA annotation, but the original test of each iteration only uses a subset of all artifacts.

To conclude, the representativeness of the annotated tests compared to the original tests — measured by ρ — is slightly impaired. We attribute this to a higher variance of the results of the annotated tests, which is due to the CSV files that need to be loaded. Furthermore, a smaller influencing factor is slightly different input data. However, the ρ values of all iterations are clearly within a baseline of $\mu_{\rho,\text{ann}} \pm 3\sigma_{\rho,\text{ann}}$.

12.3. Experimental Study with Broadleaf Heat Clinic

In this experimental study, we address RQ1.5: *To which degree do evolved parameterizations improve the representativeness of a generated load test?* We utilize the WESSBAS approach (Vögele et al., 2018) to generate load tests based on recorded requests and parameterize the generated tests using an IDPA. The study is composed of two experiment series. The first series only considers changes in the workload to investigate whether an IDPA can be used to preserve the representativeness of a generated load test repeatedly. The second series additionally includes API changes of the SUT. In the following, we describe the method we applied and present the results.

12.3.1. Experimental Method

This section outlines the method of this study. It comprises the SUT, prerequisites to be met before executing the experiments, the experiment process, and the setup.

12.3.1.1. System Under Test

As SUT, we used the Heat Clinic (Broadleaf Commerce, LLC, 2017), which is a showcase implementation of the Broadleaf Commerce Community Edi-

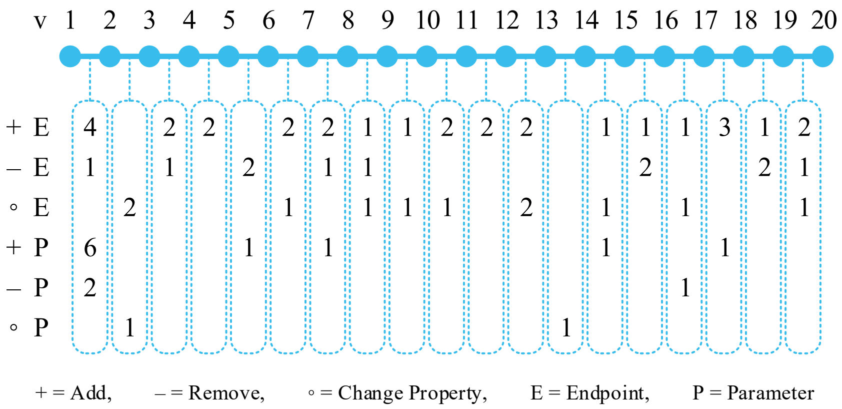


Figure 12.9.: Number of changes per operation and element between versions v1 to v20 of the Heat Clinic.

tion (Broadleaf Commerce, LLC, 2020). Broadleaf Commerce is an enterprise e-commerce platform built on current open-source technologies. The Heat Clinic is a webshop for hot sauces, providing standard webshop functionalities like browsing products, collecting them in a cart, and purchasing, as well as managing accounts, including addresses, payments, and wishlists. Overall, it provides an API with 184 endpoints as per April 16, 2018. Even though the Heat Clinic is a sample application, we expect it to be elaborate and representative for real webshop applications, because it intends to show the capabilities of the Broadleaf Commerce.

For our evaluation, we needed to have different versions with different APIs. In the commit history of the Git repository (Broadleaf Commerce, LLC, 2017), we identified one commit (010f8a2 on August 3, 2017) that introduced API changes. In the following, we refer to the version before this commit as v1. As the second version v2, we denote the current version as per April 16, 2018. For further versions v3 to v20, we adapted version v2 by adding a randomly chosen number between 1 and 5 of the most common change types identified in Section 6.4.2. We applied the changes in random order with the frequencies of occurrence provided by S. Wang et al. (2014)

(see Section 6.4.2). Figure 12.9 illustrates the result. We focused on the change types that can be (semi-) automatically evolved by our approach and have a frequency of at least 1 %. These change types are *Add/Remove Endpoint*, *Change Endpoint [Path]*, *Add/Remove Parameter*, and *Change Parameter [Name]*. Furthermore, we presume that the change operations can be generalized to all kinds of *Change Endpoint [Property]* and *Change Parameter [Property]*, as they have in common that only the application model has to be adjusted while the annotation remains unchanged. In order not to break functionality, we duplicated randomly chosen elements as additions and removed only duplicated elements. The list of applied changes is part of the replication package (H. Schulz et al., 2019f).

12.3.1.2. Prerequisites

To run the experiments, we needed two main prerequisites. First, we had to define an IDPA for parameterizing load tests. Second, we required a reference load test that mimics the production workload. To ensure that the input data specifications of the IDPA and the reference load test are equal, we used the JMeter load testing tool (Apache Software Foundation, 2020[a]) for both the reference load test and the generated representative load tests. Furthermore, we defined the IDPA first and specified the inputs in JMeter similarly. For adapting the IDPA to the duplicated endpoints and parameters of the Heat Clinic versions, we duplicated the respective IDPA elements as well. In order to vary the simulated production workload, we designed the reference load test to hold a Markov chain as the workload model. Each state of the Markov chain holds the transition probability from one endpoint of the Heat Clinic to another. In order to gain different user behavior, we varied the transition probabilities. To make sure there are only state transitions that are possible by using the user interface of the Heat Clinic, we defined the allowed transitions in a template, which we used as the basis for changing the Markov chain.

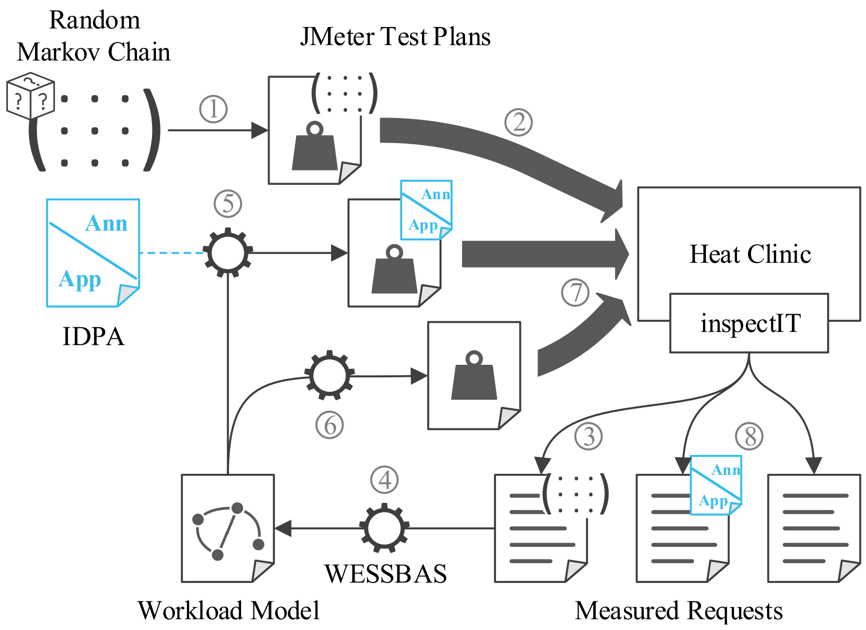


Figure 12.10.: One experiment iteration.

12.3.1.3. Experiment Process

We executed two experiment series with 20 iterations each. In the following, we describe a single iteration of each series.

With the first experiment series, we evaluated whether the IDPA can be automatically applied to a generated load test for preserving the representativeness of the test. Therefore, we used version v2 for all iterations and varied the simulated production workload. Figure 12.10 illustrates one iteration of this series. First, we generated a random Markov chain representing the production workload and replaced the original Markov chain of the reference load test ①. In this way, we had a different simulated production workload in each iteration. In the next step, we executed the reference load test against the Heat Clinic ② and collected measurement data using the open-source APM tool inspectIT (inspectIT, 2020) ③. Then, we used the WESSBAS approach (Vögele et al., 2018) to extract a representative workload model from the measured request logs ④. Next, we transformed the workload model into a JMeter load test once considering the IDPA ⑤ and once without any additional modifications ⑥. Finally, we executed both generated load tests subsequently ⑦ and collected measurement data ⑧. For evaluation, we compared the results of the three executed load tests. In order to have a clean environment, we restarted the Heat Clinic before each load test execution and populated with 200 user accounts, which were then used by the load tests.

With the second experiment series, we evaluated the effect of API changes. Therefore, we executed the same experiment series as before, but increased the version of the Heat Clinic at the beginning of each iteration, starting at v1. Furthermore, we used the IDPA evolution mechanisms to adapt it to the APIs of the respective versions and adjusted the reference load test. All IDPAs and reference tests can be found in the replication package (H. Schulz et al., 2019f).

12.3.1.4. Experiment Setup

For executing the experiments, we utilized the same machines as for the Nexus experiments (see Section 12.2), as illustrated in Figure 12.11. The first machine hosted the Heat Clinic and a lightweight Spring Boot service offering a REST API for restarting the Heat Clinic. The second machine hosted the following services:

- our IDPA evolution approach,
- JMeter for executing load tests,
- the WESSBAS approach as workload model extractor,
- the inspectIT server for collecting measurement data,
- an InfluxDB (InfluxData, 2020) time-series database for storing the measurement data,
- a Java process that ran the experiment series automatically.

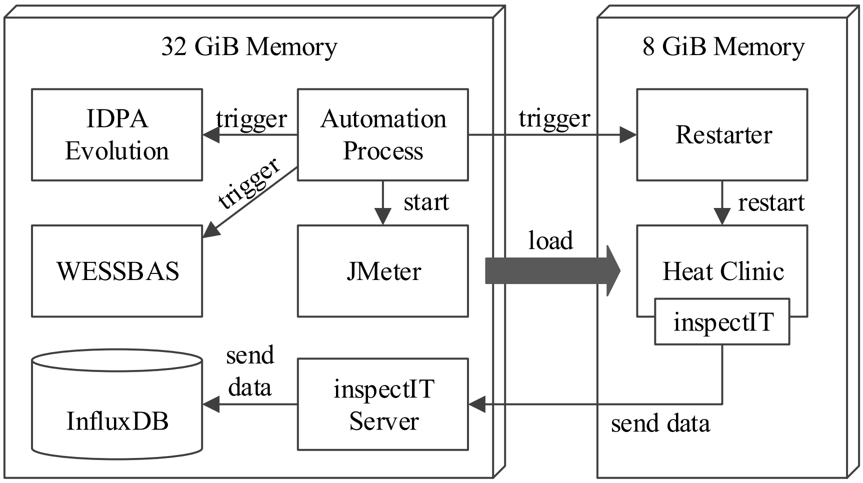


Figure 12.11.: Experiment setup with the Heat Clinic.

12.3.2. Results

In this section, we provide the results of the experimental study with the Heat Clinic. The raw data and the analysis results of all iterations are available online as part of the replication package (H. Schulz et al., 2019f).

12.3.2.1. Varying the Workload Only

We start the analysis of the results with the request rates. Figure 12.12 shows the request rates per endpoint, HTTP response code, and minute for the first iteration. We can see that, except for small variations, the load tests generated with an IDPA have similarly looking bars as the reference load tests. In contrast, the load tests without additional adjustments have a significant number of erroneous requests in addition. These requests target the error page of the Heat Clinic with path */error*. Furthermore, it turns out that the overall request rates of the generated tests are slightly smaller than the ones of the reference tests when ignoring for the additional error requests.

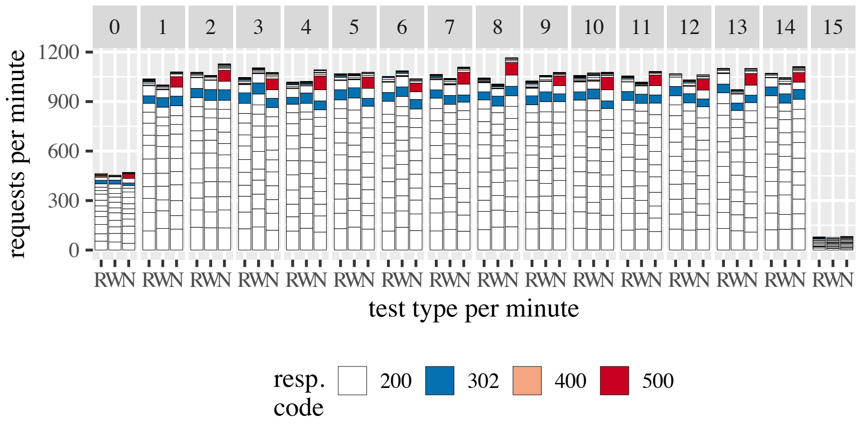
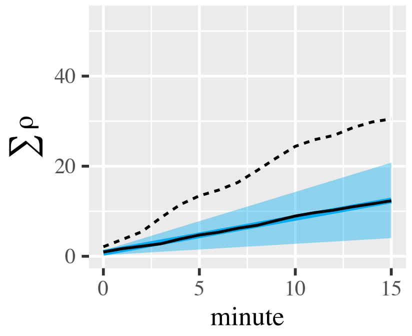
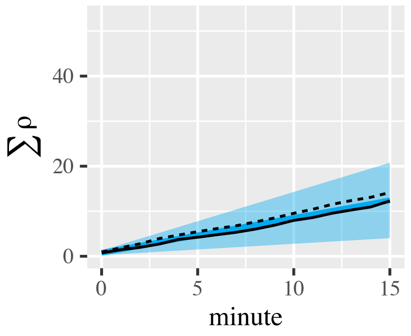


Figure 12.12.: Requests per minute divided by endpoint and response code for the first experiment iteration and the reference test (R) and generated tests with (W) and with no (N) IDPA.



(a) Iteration 1.



(b) Iteration 16.

Figure 12.13.: $\sum\rho$ metric for different iterations vs. $\mu_\rho \pm 3\sigma_\rho$ and the generated tests with (solid line) and without (dashed line) IDPA.

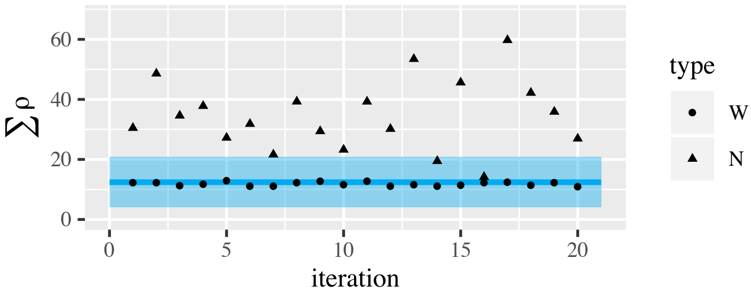


Figure 12.14.: $\sum \rho$ at test end per iteration vs. $\mu_\rho \pm 3\sigma_\rho$.

Investigating the differences in the load test executions in more detail, we calculate the cumulative $\sum \rho$ metric and a baseline for $\sum \rho$ shown in Figure 12.13 for the selected iterations 1 and 16. For the baseline, we use the results of the reference test, which we executed 20 times for this purpose with the first iteration as the reference test and the remaining 19 iterations as generated tests. That results in $\mu_\rho = 0.7753$ and $\sigma_\rho = 0.1744$ per minute. We can see that the metric is higher when not parameterizing the generated load test — ρ_N in the following — compared to using the IDPA — ρ_W in the following. While ρ_W is very close to μ_ρ in iteration 1, ρ_N is significantly larger than the baseline area. However, in iteration 16, ρ_N is close to μ_ρ, too, even though it is slightly larger, while ρ_W is continuously less than μ_ρ.

Figure 12.14 shows the cumulative ρ metric at the test end for all iterations. We can see that iteration 16 is an outlier, while in all other iterations, ρ_N is significantly larger than ρ_W. Especially, ρ_N is greater than the baseline $\mu_\rho + 3\sigma_\rho$ in all iterations except for 14 and 16. In contrast, ρ_W is within the baseline area in all iterations and even less than μ_ρ in 17 iterations. The average cumulative values at the test end are 11.8374 for ρ_W and 34.5529 for ρ_N. Furthermore, there is no significant trend when using the IDPA. Let ν be the iteration number. Fitting a linear model $\sum \rho_W = \beta_1 + \beta_2 \nu$ results in $\beta_1 = 12.0722$, $\beta_2 = -0.0224$, and deviance of 7.8675, not indicating any significant upward trend. We conclude that the generated tests without parameterization significantly impair the representativeness. In contrast to

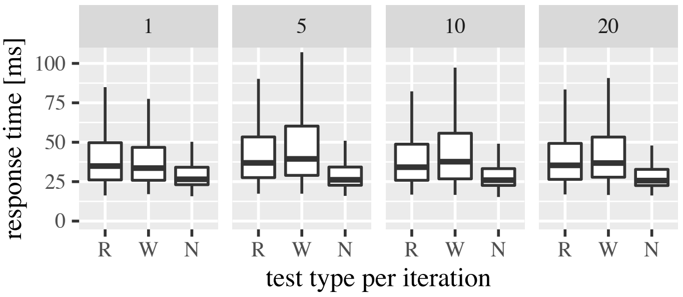
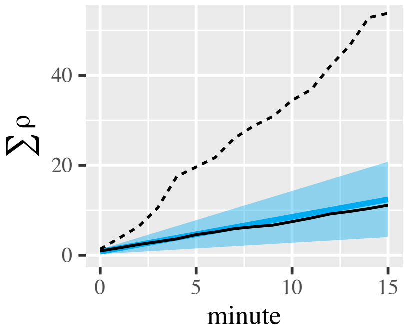


Figure 12.15.: Response time of the home endpoint (without outliers).

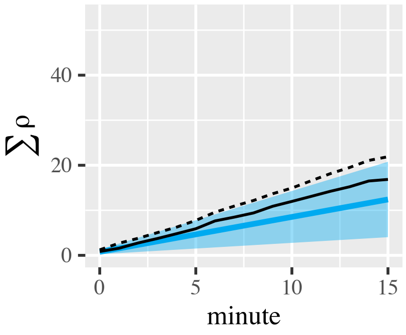
that, the representativeness of the tests generated with IDPA is clearly within normal variations of the reference test results.

Finally, we analyze the consequences of less representative load tests. For this purpose, we investigate the response times of the home endpoint of the Heat Clinic during each load test, shown in Figure 12.15. This endpoint has no parameters and always returned the response code 200. The box plots of the reference test and the generated test with IDPA look relatively similar, while the response times of the generated tests without adjustments appear to be smaller. To verify that the difference is significant and to measure the effect size, we apply a two-sided t-test and Cohen's d. Even though our samples do not follow a normal distribution, they have large sample sizes between 670 and 1536. Thus, we can apply the t-test due to the central limit theorem. As the significance level, we use 0.05. Our null hypothesis for iteration j is $H_{j,0} : \mu_{\text{ResponseTime}_{\text{ref},j}} = \mu_{\text{ResponseTime}_{\text{gen},j}}$.

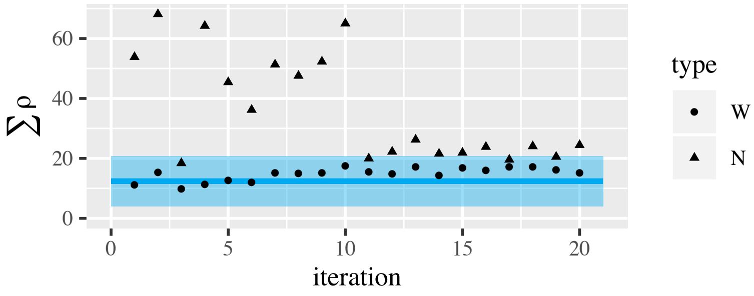
For the IDPA tests, the t-test rejects $H_{j,0}$ and accepts $H_{j,A}$ for nine iterations while it cannot reject $H_{j,0}$ for the remaining 11 iterations. Cohen's d is smaller than 0.2 and, thus, *negligible* for all 20 iterations with a maximum absolute value of 0.1997. For the tests without adjustments, $H_{j,0}$ is rejected for all iterations. Cohen's d is *small* for eight iterations with values between 0.3183 and 0.4888 and *medium* for 12 iterations with values between 0.5346 and 0.7377.



(a) Iteration 1.



(b) Iteration 15.



(c) At test end per iteration.

Figure 12.16.: $\sum \rho$ metric of the experiments with varied API vs. $\mu_\rho \pm 3\sigma_\rho$ for the generated tests with (W/solid line) and with no (N/ dashed line) IDPA.

12.3.2.2. Varying the Workload and the API

The request rates of the second experiment series look very similar to the ones shown in Figure 12.12. The ρ metric appears similar to the one before as well, as shown in Figure 12.16. As before, ρ_W is close to μ_ρ in all iterations and clearly within $\mu_\rho \pm 3\sigma_\rho$. Also, there are iterations where the difference between ρ_W and ρ_N is vast — such as iteration 1 (Figure 12.16a) — and others, where it is less — such as iteration 15 (Figure 12.16b). A visible difference in Figure 12.16c is that ρ_N decreases and stays small after iteration 10. We explain this finding by a change in the API. A potential candidate is the duplicate of the logout endpoint, which was introduced in version v11. Furthermore, the difference between ρ_W and μ_ρ appears to be slightly larger than in the first experiment series. However, ρ_W is still smaller than ρ_N in all iterations. Again, we fit a linear model $\sum \rho_W = \beta_1 + \beta_2 v$ into the measured cumulative values at test end. It results in $\beta_1 = 11.9536$, $\beta_2 = 0.2699$, and deviance of 47.0205. Hence, the fitted model indicates a small upward trend, but with a high fitting error compared to the first experiment series.

12.4. Industrial Case Study

In this industrial case study, we address RQ1.6: *How expressive is our approach compared to parameterizations of load tests used in industrial projects?* For this purpose, we analyzed the load tests used in four different industrial software development projects. In the following, we provide the method, present the results, and provide the lessons we learned while conducting the case study.

12.4.1. Case Study Method

In this section, we describe how we approached the case study in terms of case study design and planning, data collection, and data analysis.

12.4.1.1. Design and Planning

We selected the case study subjects from our existing contacts. The subjects were four different industrial projects from the construction and automotive sector. In each of the projects, a web-based software application was developed, which comprised both user-faced and backend applications. For the sake of confidentiality, we refer to the projects as A, B, C, and D in the following.

Each of the projects created and executed load tests for the developed application, including IDPA-related parameterization concepts, which constituted our source of information. We considered them to reach a high diversity of parameterization concepts. We were given full access to the load tests for analysis purposes.

At the time this study was executed, not all IDPA `Inputs` introduced in Section 6.3 were existing. Essentially, the results of the study motivated adding new `Inputs`. The list of `Inputs` that already existed is:

- `ExtractedInput` with both `RegExExtraction` and `JsonPathExtraction`
- `DirectListInput`
- `CsvInput`
- `CounterInput`
- `JsonInput`

12.4.1.2. Data Collection

The precise artifacts we were given access to were all load tests, including accompanying artifacts that were needed to execute the tests. Table 12.1 provides a summary of the number of targeted endpoints, the number of load tests per project, and the evaluation results. All load tests were implemented in Scala using the Gatling tool (Gatling Corp, 2020). Furthermore, the load tests of each project utilized a common codebase, e.g., with implementations of requests to specific endpoints of the application and configuration files such as CSV or JSON files specifying input data. All load tests have been implemented manually by the respective development team, based on the known or intended usage of the application. Hence, they were not representative as presumed in this work. However, the set of parameterizations used in a load test is independent of the fact whether the test is generated or manually implemented. For conducting the case study, we had access to the load tests, including the common codebases and all configuration files of each project.

Table 12.1.: Overview of the Case Study Results

	Project A	**Project B**	**Project C**	**Project D**
#Endpoints	7	17	1	34
#Load Tests	2	1	1	$3 + 12$
		Overrides		
domain	✓	✓	✓	✓
base-path	✓			
header				✓
		Core Inputs		
Direct	3	3	174	9
Csv	3		11	6
Extracted (Json)	2	10		
Extracted (RegEx)		4		44
Json		17	470	19
		Custom Inputs		
RandomString*		1		1
RandomNumber*		1		3
Auth		1		
Filtered		1		
Datetime*		3	1	1
Environment*			1	1
Combined*				2
		Summary		
% Custom Inputs	0.00 %	17.07 %	0.30 %	9.30 %

*Added to the IDPA after the study (see Section 6.3)

12.4.1.3. Data Analysis

For each of the projects, we analyzed the provided load tests. We identified all specifications that were used to parameterize the tests. These were all specifications defining input data for parameters or modifying the execution of a request from its default. For instance, if a specific header was explicitly defined for a request, we considered it being such a specification. Also, we identified all specifications that would be necessary to transform a request from the production system to the test system. That is, assuming a request was extracted from production request logs, we determined the changes that would need to be made to bring the request into the actual form. Having all parameterizations identified, we tried to express them using the IDPA. In case this was not possible, we introduced new concepts using the provided extension points.

We performed the steps described above for each project individually and sequentially, starting from project A and ending with project D. In each step, we introduced extensions of the IDPA if necessary and reused them in the next project if possible. In this way, we could determine whether the introduced extensions can be generally used and, thus, could be added to the core IDPA, or whether each project requires individual parameterization concepts.

12.4.2. Results

Table 12.1 provides an overview of the results of our industrial case study. We show the number of load tests, utilized `Overrides`, utilized `Inputs` contained in the IDPA metamodel at the time of the study, and utilized custom `Inputs`, which we introduced using the extension points of the IDPA. Also, we provide the percentage of custom `Inputs` across all utilized `Inputs`.

12.4.2.1. Project A

The first project we investigated had two different load tests that targeted seven endpoints and were implemented using the same code base. We identified two overrides that would need to be used for transforming production requests into requests to the test system. First, the domain name needed to be changed with the *HttpEndpoint.domain* override. Second, each stage in this project had a specific base path, such as */stage/test* or */stage/dev*[1]. Hence, we needed to override this base path using the *HttpEndpoint.base-path* override.

The input data concepts used in the load tests correspond to the IDPA entities `DirectInput`, `CsvInput`, and `ExtractedInput`. For the `ExtractedInputs`, we used `JsonPathExtractions` to extract the input data from returned JSON bodies. We were able to represent all parameterizations solely using the provided IDPA concepts. Hence, the percentage of custom inputs is 0 %.

12.4.2.2. Project B

Project B had one load test, which targeted 17 endpoints and consisted of several Scala code files. As `Overrides`, we only identified *HttpEndpoint.domain* for adjusting the domain name of the requests. The utilized core inputs were `DirectInput`, `ExtractedInput`, and `JsonInput`. The `ExtractedInputs` were both used with `RegExExtractions` and `JsonPathExtractions`. The `JsonInputs` were used to define JSON values, which consisted of both static values and dynamically retrieved values, e.g., from `ExtractedInputs`.

However, the load test of this project utilizes several parameterization concepts we could not represent using the core inputs. Therefore, we introduced custom inputs using the `Input` extension point. First, the load test uses randomly generated universally unique identifiers (UUIDs) as an input. We introduced the `RandomStringInput`, which generates a random string

[1]These base paths do not reflect the actual paths and are only for illustration purposes.

based on a template (see Section 6.3.2.2). For the UUIDs, we used a template to generate random strings in the appropriate format. The second extension we introduced is the `RandomNumberInput`, which randomly selects a number between a lower and an upper limit (see Section 6.3.2.2). In this case, the lower limit was set to 0, while the upper limit was extracted from a JSON response. Next, the load test utilizes an application-specific authentication mechanism. This mechanism generates a specific header to be used in all requests. For that, we introduced the `AuthInput`, which generates these headers. We used a `FilteredInput` for selecting a certain percentage of multiple values from another `Input`, in this case, an `ExtractedInput` utilizing a `JsonPathExtraction`. Finally, for generating dates, which are used as input to some parameters, we introduced the `DatetimeInput` (see Section 6.3.2.2). This `Input` produces the current date and time in a defined pattern (e.g., *yyyy-MM-dd*) and with an optionally defined offset. Overall, seven of the 41 utilized `Inputs` were of the newly introduced custom types, which is 17.07%.

12.4.2.3. Project C

Project C had one load test defined in a single source file, accompanied by configuration files. It targets a single endpoint. As before, the *HttpEndpoint.domain* override would need to be used for transferring production requests to the test system. The used core inputs are `DirectInput`, `CsvInput`, and `JsonInput`.

For this project, we also had to make use of custom extensions. First, we could reuse the `DatetimeInput` introduced for Project B. This time, the pattern used was the time stamp in seconds. Furthermore, we had to add an ability to use input data defined as environment variables. For that, we introduced the `EnvironmentInput`, which reads the value of an environment variable and also allows us to define a particular template into which the value should be inserted (see Section 6.3.2.2). Overall, the percentage of custom inputs is 0.3 %.

While we were able to represent all parameterizations as an IDPA, we encountered limitations of the `JsonInput` and the `CsvInput`. The JSON body of one request of the load test is defined by a JSON file with 1250 lines. We represented this JSON by a `JsonInput` but resulted in a much longer description with 3094 lines. Furthermore, due to the recursive structure of the `JsonInput`, we had to define 470 nested `JsonInputs`. Because of this and because the `JsonInput` does not precisely show the JSON structure, readability is degraded. This finding motivates implementing a new input, which allows specifying long JSON strings more concisely. The limitation of the `CsvInput` is because this load test uses a CSV file with multiple columns. However, the `CsvInput` only allows using one column. Hence, we needed to add one `CsvInput` per column and associate the inputs, which resulted in a less concise definition and redundant information, such as the CSV file's name.

12.4.2.4. Project D

The last project we investigated contained three different load tests. Additionally, there were twelve small tests, which were used for synthetic monitoring, i.e., for executing them against the production system regularly to check whether the system behaves as intended. Because the technology of these tests was similar to the load tests, we considered both test types for our evaluation. Overall, the tests targeted 34 different endpoints.

The overrides we utilized are *HttpEndpoint.domain* and *HttpEndpoint.header* for adding a set of defined headers to the requests. The used core inputs are `DirectInput`, `CsvInput`, `ExtractedInput` with a `RegExExtraction`, and `JsonInput`. From the already introduced custom inputs, we made use of the `RandomStringInput`, `RandomNumberInput`, `DatetimeInput`, and `EnvironmentInput`. However, we also had to introduce another custom input for combining the values of several other inputs. The `CombinedInput` consists of a list of `Inputs` and a template that defines how to combine the `Inputs` (see Section 6.3.2.2). In this case, we utilized `CombinedInputs` for defining a combination of randomly chosen numbers, which we generated using the `RandomNumberInput`, to obtain a random but valid date string. Overall, the percentage of custom inputs is 9.3 %

Another kind of parameterization we identified in this project is explicitly defined HTTP cookies. Before sending several requests, the load tests dynamically set a cookie. With the existing concepts of the IDPA, we were not able to represent cookies, because they can neither be mapped to `Overrides` nor `Inputs`. `Overrides` are not suitable because they cannot deal with dynamically defined values. `Inputs` are not appropriate, because they can only define the full value of a parameter — in this case, the cookie header — and cannot add a value to the existing list of cookies. Therefore, we did not take the cookies into account in the IDPA.

12.4.3. Lessons Learned

During the execution of the case study, we learned several lessons, which we summarize in the following.

12.4.3.1. Most Input Data Specifications Are Common

As is can be seen in Table 12.1, the clear majority of input data specifications used in the projects are list-based, extracted, and JSON input data. Precisely, 97.85 % of all input data specifications fall into these types. This fact indicates that the data flow of most applications can be described with these means. As a consequence, load testing tools or input data models, such as the IDPA, should at least implement these kinds of input data specifications.

12.4.3.2. Individual Input Data Specifications Are Required

Even though the input data specifications that do not fall into the aforementioned common categories are only 2.15 %, there still exist cases in which they are required to parameterize a load test properly. Mainly, there is a need for adding application-specific input data specifications, such as the `AuthInput` in Project B. Without this `Input`, the authentication header required in each request could not be generated. Hence, load testing or input data specification approaches should consider custom input data specifications. The original load test implemented in Gatling used the comprehensive capabilities of the underlying Scala language. Other load testing tools such as JMeter allow adding plugins implementing custom functionality. In our IDPA, we addressed this challenge by extending the `Input`, to implement custom inputs such as the `AuthInput`.

12.4.3.3. Input Data Can Be Large

A challenge we faced when defining the IDPA was the size of some input data. An example is the large `JsonInput` based on a JSON file with 1250 lines in Project C. Hence, the solution used to define this JSON value needs to be scalable. Other examples of extensive input data are CSV files, such as one in Project C with more than 25000 lines. Furthermore, CSV files can have multiple columns that each define a set of input data. Therefore, it is crucial to provide input specifications such as the `CsvInput`, which allows referring to external files instead of inlining its content. In our approach, we encountered limited scalability of the `JsonInput` and `CsvInput`, which we addressed after the study.

12.4.3.4. Common Input Data Specifications Are Often Sufficient

Another finding is that people tend to use comprehensive specification mechanisms if they are available, even if more standard mechanisms would be sufficient. For instance, Project D utilized the `CombinedInput` and several `RandomStringInputs` for generating a random date. The same result could be achieved either by using a single `RandomStringInput` with an appropriate template, or even by using a `CsvInput` with a CSV file prefilled with random dates. The same applies to the `DatetimeInput` used in this project, which was utilized for generating a random but unique number. However, the specifications used by the project are most convenient, because neither complex templates have to be defined nor long lists of input values have to be specified. Furthermore, there are also cases where the `DatetimeInput`, `RandomNumberInput`, and `RandomStringInput` are required, because either the current date has to be defined, or a random value based on extracted values has to be generated. Both are the case in Project B.

Concluding, most input data can be defined with the common types of specification, but with potentially less convenience. At the same time, there can be specifications that cannot be mapped to the standard types. As a consequence, input data models such as the IDPA do not need to implement all variants of imaginable input data specifications, but also need to provide means for several non-standard specifications for being usable in industry.

12.5. Discussion of Research Questions

In this section, we discuss the research questions defined in Section 5.1 based on our approach to automated load test parameterization (Chapter 6) and the evaluation presented in the previous sections.

12.5.1. RQ1.1 — Automated Parameterization Evolution

> *How can load test parameterizations be automatically evolved if the workload changes?*

For evolving load test parameterizations over changing workloads, we introduced the Input Data and Properties Annotation (IDPA). It externalizes all parameterizations of a generated load test and can be applied to the load test automatically. We provide parameterizing JMeter (Apache Software Foundation, 2020[a]) and BenchFlow (Ferme and Pautasso, 2018) load tests, and our approach can be extended to any load testing tool. Hence, users of our approach can specify tool-independent parameterizations without knowledge about the workload in advance. This includes re-applying the parameterizations to a load test generated from a changed workload model.

12.5.2. RQ1.2 — Relevant API Changes

> *Which API change types exist that affect load test parameterizations?*

As the API of the SUT can change as well, we investigated API change types collected in the literature and analyzed their impact on an IDPA. We extracted and summarized the relevant change types in a new classification, which we provide in Section 6.4.2. The relevant change types comprise endpoint and parameter additions, removals, and property changes, and changes in the input and response behavior. We used these types for developing strategies to handle API changes.

12.5.3. RQ1.3 — Reducing the Maintenance Effort

To which degree can we reduce the maintenance effort for the evolution of load test parameterizations if the API changes?

For supporting the evolution of IDPAs if the API of the SUT changes, we developed evolution mechanisms for each of the identified relevant change types. Hence, an IDPA can be semi-automatically adapted based on an expert's feedback. The IDPA can be adjusted directly when the API changes and before a new load test is to be generated. Hence, new load tests can be generated and executed fully automatically without manual effort at test generation or execution time.

For the further evaluation of the reduction of the total maintenance effort, we derived estimation models and compared them asymptotically. Even though we could not determine the precise effort in units such as person-days, our models show that using an IDPA reduces a quadratic cumulative effort over time to a linear one. Irrespective of the precise values of the model's parameters, this will lead to a significantly reduced effort when using an IDPA in the long term. For future work, we suggest parameterizing our derived models with actual values, which could be determined by empirical studies.

12.5.4. RQ1.4 — Impairing the Representativeness

How much does the parameterization by our approach impair the representativeness of a load test?

In our first experimental study with Nexus, we were able to parameterize representative load tests, which replayed recorded requests from a productive Nexus instance, while mostly preserving the representativeness. The representativeness metric ρ we calculated was within a baseline representing the natural variations of such parameterized load tests. However, we also encountered two factors that slightly impaired representativeness. First, the variation of the parameterized tests was higher than the original tests, which is because of CSV files the load test loaded before each request. Second and more important, there was a small but systematic variation from the baseline mean, because we were not able to achieve the same ratio of artifacts successfully requested to those that could not be found (response code 404). We attribute this to the fact that the parameter values of the requests influence whether an artifact will be found and, thus, how the internal program flow behaves. Opposed to Nexus, in our second experimental study with the Heat Clinic, we could not observe such effects, as the input data do not significantly influence the program flow, as long as they are defined correctly.

We conclude that for applications whose workload and internal behavior are dominated by the order and rate of submitted requests, the IDPA can be used for reliably parameterizing a load test without impairing the representativeness. In contrast to that, load tests for applications with significantly different behavior for different parameter values can only be as representative as the input data defined in the IDPA. Hence, the benefit added by a parameterization through an IDPA is less because the dominant part of the workload model has to be defined manually. The load tests for Nexus slightly suffered from this effect, even though we still were able to define input data resulting in representativeness within the baseline range.

12.5.5. RQ1.5 — Improving the Representativeness

To which degree do evolved parameterizations improve the representativeness of a generated load test?

In the second experimental study, we used the IDPA for its intended purpose, namely the parameterization of generated load tests. Compared to unparameterized load tests, the IDPA could significantly improve the representativeness. This difference was reflected in the ρ metric. While the metric values for the parameterized tests can be well explained with the natural variations of the reference tests, the values for the unparameterized tests were much higher than the baseline in general. Also, we could not detect any trend, at least when only the workload was changing. With a changing API, there were more variations, and we could also detect a small trend of decreasing representativeness with the IDPA, even though the linear correlation was not statistically significant. In the end, all ρ values were clearly within the baseline for the parameterized tests in all iterations, while they were outside the baseline for the unparameterized tests in most iterations.

An important finding is that reduced representativeness can also impair the behavior of the tested application, such as the response times. In our experiments, the response times of the unparameterized tests were significantly different from the ones of the reference tests with small to medium effect sizes. In contrast to this, the tests parameterized with the IDPA only had significantly different response times in less than half of the iterations with a negligible effect size. We can conclude that improper or missing parameterization of generated load tests can lead to different behavior of the tested application and, thus, to corrupt load test results. The IDPA turned out to be one possibility to add the required parameterizations to the generated tests repeatedly.

12.5.6. RQ1.6 — Expressing Industrial Parameterizations

How expressive is our approach compared to parameterizations of load tests used in industrial projects?

In our case study with four different industrial projects, we were able to represent the adjustments of static properties and input data of all investigated load tests in IDPAs. However, we had to use the extension points of the IDPA to introduce new `Inputs`. Most of the newly introduced `Inputs` could be shared between the investigated projects. In particular, while we had to add five new Inputs for Project B, we only had to add one new Input for Project C and D, respectively. Hence, we presume we already covered a majority of all relevant input types. In summary, our core input types covered more than 97.85 % of all inputs of all investigated projects.

The newly introduced input types motivate adding them to the core IDPA metamodel. However, this is not suitable for all of them. In particular, the AuthInput is specific to Project B. It should not be added to the metamodel, as other projects cannot reuse it. In contrast, the AuthInput is a good example where extension points are required. Furthermore, the Filtered-Input was used only once in Project B as a specific operation and might not be implementable in all load testing tools. The remaining input types are independent of the project or tested application. Therefore, we integrated them into the IDPA metamodel for reuse, as described in Section 6.3.2.

The case study also revealed three limitations of the core IDPA. First, we encountered that large JSON files cannot be defined concisely using the existing Inputs. The JsonInput enables such definitions, but is cumbersome and decreases readability. Therefore, we added the ConciseJsonInput for better use with long JSONs (see Section 6.3.2.2). As it merely holds the JSON object tree formatted in YAML, it is less flexible than the current implementation but scales better with large JSON values. Second, CsvInputs turned out to define redundant information in the case of multiple rows in the CSV file. The CsvInput is designed in this way because the Parameter-Annotations need to link to one individual column directly. As a solution, we introduced the CsvInputGroup, which holds multiple CsvInputs to which the ParameterAnnotations can refer. In doing so, we can define all the information about the CSV file once in the CsvInputGroup. Last, Project D required setting a defined HTTP cookie. We were not able to map this to an IDPA concept. However, we presume that explicitly setting cookies is not required for representative load testing, which extracts the request model from recorded requests and represents the behavior of real users. A real user would interact with the system without setting cookies explicitly. In contrast to that, the load tests of Project C were manually defined and did not precisely represent a real user's behavior.

Overall, we conclude that the IDPA is suitably expressive for defining parameterizations of extracted representative load tests of industrial applications. The version used for the study lacked in several `Input` types that were required for multiple applications, which we added after the study using the `Input` extension point. Furthermore, we encountered the usefulness of the extension mechanism, because Project B required an input type that is specific to the project and should not be added to the core IDPA, namely the `AuthInput`.

12.6. Threats to Validity

We see the following threats to the validity of our approach to the automated parameterization of load tests and its evaluation. We group the threats by the conclusion, internal, construct, and external validity.

12.6.1. Conclusion Validity

In our evaluation, we derived several conclusions that can bear threats to the validity of our study. In the experimental study with the Heat Clinic, we compared response time means using a two-sided t-test. However, the boxplots of the response times appeared asymmetric and, thus, indicated a distribution other than a normal distribution. However, as all samples had a size of at least 670, we could apply the t-test according to the central limit theorem.

In the study with Nexus, we based our analysis on the results of the JMeter load tests rather than server-side monitoring tools. Hence, there was latency between the first machine hosting the load test and the second machine hosting Nexus, which could potentially have impaired the results. However, we presume this latency was relatively small, as the machines were connected with a 1 Gbit switch. Furthermore, as the low original baseline indicates, there was low variation in the results except for the CSV file loading. Thus, we conclude that there was no significant impairment.

12.6.2. Internal Validity

We defined a metric ρ that bases on a weight function w. Potentially, this weight function could have slightly impaired our results. However, w only reduces the impact of minimal request rates, because they are likely to deviate strongly in percentage. Thus, w smoothens the metric without changing the value for higher, reliable request rates.

Our second metric θ assumes that all changes with a particular operation to a specific type of element require the same effort. In practice, there will be more variation. However, for an asymptotic comparison, the assumption is sufficient. When parameterizing our derived effort estimation models with measured values, future works might consider the variation of the effort, e.g., as a probability distribution per change operation and type.

Another threat is that we evaluated the IDPA's expressiveness with non-representative load tests in the industrial case study. Hence, the findings might not be relevant for generated representative load tests. However, we presume the parameterization concepts used in load tests are independent of the representativeness, as they mostly depend on the tested application.

12.6.3. Construct Validity

Our approach and experimental studies assume that test data with the same quality and quantity as the production data are available. In fact, in the experimental studies, we used the same database for the reference and generated load tests. In practice, this might not be the case. However, generating and managing representative test data is not our research focus. For this purpose, we refer to existing approaches by Barros et al. (2007) and Farahbod and Dadashi (2017).

Another potential threat is the use of Markov-chain-based workload models. In practice, request sequences are used more often. However, as the IDPA is entirely independent of the workload model, we can assume that there were no side effects due to Markov chains. Furthermore, Markov chains are treated to be suitable for load testing (Z. Li and Tian, 2003), and we use them as a central modeling concept in this dissertation.

Finally, we artificially introduced API changes to the Heat Clinic for assessing the influence of such changes on the representativeness of load tests parameterized by an IDPA. Hence, our findings could originate from changes that are uncommon in practice. Therefore, we based on S. Wang et al. (2014) to implement changes according to a typical distribution. Besides, we considered API changes that were already contained in the commit history between version v1 and v2, which did not result in fundamentally different measurements.

12.6.4. External Validity

We concluded from our industrial case study that the IDPA is suitably expressive for defining load test parameterizations. Also, we presumed we covered most of the required `Input` types with the introduced extensions because most of these types could be shared between the projects. However, this finding might stem from a similarity of the investigated projects. We faced this issue by investigating four different industrial projects. Furthermore, the `Input` types, including the ones identified in the study, appeared to be sufficient for all of our other studies as well. In the end, the provided extension points allow adding new input types, which can — if generally applicable — be integrated into the core metamodel.

Finally, we only investigated web applications having REST APIs. We cannot generalize our results to applications that do not meet this assumption. Even though the IDPA is not limited to a particular type of application, such a generalization is not a goal of this work. For future work, we suggest additional studies complementing ours, especially in other domains and contexts, e.g., other than web-based applications.

12.7. Summary

In this chapter, we presented the evaluation of our approach to the automated parameterization of load tests using the IDPA. It comprised four different studies, including estimation models, experimental studies, and an industrial case study. We can conclude that our IDPA is suitable for parameterizing load tests and, thus, generating load tests automatically.

In the following chapters, we provide evaluations of our further work packages. In these studies, we use the IDPA for parameterization. Due to the automation, this parameterization is not only required for the use in continuous software engineering (CSE) but also eases experimentation.

Chapter 13

Evaluating Service-tailored Load Testing

In this chapter, we evaluate our approach for tailoring load tests to services (see Chapter 7). We introduced two algorithms — log-based and model-based tailoring —, which extend the existing extraction process for generating load tests that directly target a particular set of services. Thus, fewer services need to be deployed for the test, and resources can be saved. Our evaluation addresses the sub-questions of RQ2: *How can representative load tests be tailored to specific services of a session-based application?*

The evaluation comprises a formal correctness verification and an experimental study with a representative microservice application. In the verification, we prove that both algorithms fulfill the requirements defined in Sections 7.4.1 and 7.5.1. As previously described, only the think time variation is an approximation, which we cannot prove to be correct. Furthermore, we compare the algorithm's number of states, workload model structure, transition probabilities, and think time variations.

Therefore, we investigated the quality of the tailored load tests further in an experimental study with the Sock Shop (Weaveworks, Inc., 2020), which Aderaldo et al. (2017) assess to be a representative microservice application.

Load tests tailored using our algorithms generate slightly less representative workloads than an untailored load test does but in an acceptable range. In contrast, simple request-based load tests perform significantly worse. Regarding representativeness and qualitative characteristics, the model-based algorithm performs better than the log-based one. However, there are indicators that the latter becomes more relevant for large-scale applications. We provide a replication package online (H. Schulz et al., 2019b).

The chapter consists of five sections. Section 13.1 presents the formal verification of the algorithms' correctness. In Section 13.2, we present the experimental study. In Sections 13.3 and 13.4, we discuss the evaluation results concerning the research questions and the threats to validity. Section 13.5 summarizes the chapter.

This chapter is a revised version of Chapter 5 of our below publication, extended by a correctness proof of the introduced algorithms (Section 13.1).

- H. Schulz, T. Angerstein, D. Okanović, and A. van Hoorn (2019a). "Microservice-tailored Generation of Session-based Workload Models for Representative Load Testing." In: *Proceedings of the 27th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2019)*. IEEE Computer Society, pp. 323–335

13.1. Correctness Verification

In the following, we verify the correctness of the service-tailoring algorithms, related to RQ2.1: *How can we extend the load test extraction process for generating service-tailored load tests?*

In Sections 7.4.1 and 7.5.1, we defined requirements to the log-based and model-based algorithms, formulated in the postconditions $A_{\text{req}}^{(\text{log})}$, $A_{\text{ID}}^{(\text{log})}$, $A_{\text{states}}^{(\text{model})}$, $A_{\text{prob}}^{(\text{model})}$, and $A_{\text{time}}^{(\text{model})}$. Here, we prove that both algorithms fulfill the postconditions, given that the input is appropriate. Furthermore, we compare the algorithms on the basis of the postconditions. In Section 13.2, we will present an experimental study complementing the formal insights.

13.1.1. Log-based Tailoring

In this section, we prove that the log-based tailoring algorithm (Algorithm 7.2) fulfills the postconditions $A^{(\text{log})}_{\text{req}}$ and $A^{(\text{log})}_{\text{ID}}$ we defined in Section 7.4.1. We prove each postcondition individually.

13.1.1.1. Requests — $A^{(\text{log})}_{\text{req}}$

For the first postcondition, we define the following theorem, which we prove in the following.

Theorem 13.1 ($A^{(\text{log})}_{\text{req}}$)
Given a set $\mathcal{T}$ of well-formed non-empty traces, a mapping $\phi : \mathcal{R} \to \mathbb{R}$ of the traces' root requests to session IDs, and a set of endpoints $\mathcal{E}$, TAILORREQUESTLOGS *(Algorithm 7.2) returns request logs $\mathcal{R}'$ to which the following applies:*

$$\forall r \in R : \big(r \in \mathcal{R}' \leftrightarrow \exists \tau \in \mathcal{T} : \varepsilon(r) \in \mathcal{E} \land P_{\mathcal{E}}(\tau, r)\big)$$

We prove the theorem by first proving that certain invariants apply to TAILORREQUEST (Algorithm 7.1), which TAILORREQUESTLOGS uses. We formalize these invariants in the following lemma.

Lemma 13.2
Given a well-formed non-empty trace τ and a set of endpoints $\mathcal{E}$, the following invariants apply to the while*-loop in* TAILORREQUEST *(Algorithm 7.1; lines 3 to 11):*

$$INV^{(log)}_1 \equiv \forall r \in \mathcal{R}' : \varepsilon(r) \in \mathcal{E} \land P_{\mathcal{E}}(\tau, r)$$
$$INV^{(log)}_2 \equiv \forall \tilde{\tau} \in \overline{T} : P_{\mathcal{E}}(\tau, r_{\tilde{\tau}})$$
$$INV^{(log)}_3 \equiv \forall r \in \tau : \big(\varepsilon(r) \in \mathcal{E} \land P_{\mathcal{E}}(\tau, r)\big) \to \big(r \in \mathcal{R}' \lor \exists \tilde{\tau} \in \overline{T} : r \in \tilde{\tau}\ \big)$$

Proof. We prove Lemma 13.2 by induction. We mark the artifacts after the i-th iteration with the superscript (i). Furthermore, we use the notations

from the algorithm, e.g., $\overline{T}^{(i)}, \mathcal{R}^{(i)}$ denote the $\overline{T}, \mathcal{R}'$ of the algorithm, and $\tau, \mathcal{E}$ denote the inputs.
Base Case:

$$\mathcal{R}^{(0)} = \emptyset \Longrightarrow INV_1^{(0)}$$
$$\left(\overline{T}^{(0)} = \{\tau\}\right) \land P_{\mathcal{E}}(\tau, r_\tau) \Longrightarrow INV_2^{(0)}$$
$$\overline{T}^{(0)} = \{\tau\} \Longrightarrow \forall r \in \tau \ \exists \tilde{\tau} \in \overline{T}^{(0)} : r \in \tilde{\tau} \Longrightarrow INV_3^{(0)}$$

Inductive Step ($i \rightsquigarrow i+1$): Given $INV_1^{(i)}$ to $INV_3^{(i)}$ hold for $i \geq 0$, $INV_1^{(i+1)}$ to $INV_3^{(i+1)}$ hold, too. For proving that, we differentiate between two cases, according to the *if* and *else* branches (lines 6 and 8 in Algorithm 7.1). We refer to the trace the algorithm chose from $\overline{T}$ in iteration $i+1$ as $\overline{\tau}$.

Case $\varepsilon(\overline{r}) \in \mathcal{E}$:

$$\begin{aligned}
& INV_1^{(i)} \land \left(\overline{\tau} \in \overline{T}^{(i)} \land \varepsilon(\overline{r}) \in \mathcal{E}\right) \\
\overset{INV_2^{(i)}}{\Longrightarrow} \quad & \left(\forall r \in \mathcal{R}^{(i)} : \varepsilon(r) \in \mathcal{E} \land P_{\mathcal{E}}(\tau, r)\right) \land \left(P_{\mathcal{E}}(\tau, \overline{r}) \land \varepsilon(\overline{r}) \in \mathcal{E}\right) \\
\overset{\mathcal{R}^{(i+1)} = \mathcal{R}^{(i)} \cup \{\overline{r}\}}{\Longrightarrow} \quad & \forall r \in \mathcal{R}^{(i+1)} : \varepsilon(r) \in \mathcal{E} \land P_{\mathcal{E}}(\tau, r) \\
\Longrightarrow \quad & INV_1^{(i+1)}
\end{aligned}$$

$$\begin{aligned}
& INV_2^{(i)} \\
\Longrightarrow \quad & \forall \tilde{\tau} \in \overline{T}^{(i)} \setminus \{\overline{\tau}\} : P_{\mathcal{E}}(\tau, r_{\tilde{\tau}}) \\
\overset{\overline{T}^{(i+1)} = \overline{T}^{(i)} \setminus \{\overline{\tau}\}}{\Longrightarrow} \quad & INV_2^{(i+1)}
\end{aligned}$$

$$\begin{aligned}
& INV_3^{(i)} \land \varepsilon(\overline{r}) \in \mathcal{E} \land P_{\mathcal{E}}(\tau, \overline{r}) \land \overline{r} \in \mathcal{R}^{(i+1)} \\
& \Longrightarrow \forall r \in \tau : \left(\varepsilon(r) \in \mathcal{E} \land P_{\mathcal{E}}(\tau, r)\right) \rightarrow \left(r \in \mathcal{R}^{(i+1)} \lor \exists \overline{\tau} \in \overline{T}^{(i+1)} : r \in \overline{\tau}\ \right) \\
& \Longrightarrow INV_3^{(i+1)}
\end{aligned}$$

Case $\varepsilon(\overline{r}) \notin \mathcal{E}$:

$$INV_1^{(i)} \land \left(\mathcal{R}^{(i+1)} = \mathcal{R}^{(i)}\right) \Longrightarrow INV_1^{(i+1)}$$

$$\begin{aligned}
& INV_2^{(i)} \land \varepsilon(\overline{r}) \notin \mathcal{E} \land \forall \tilde{\tau} \in \overline{T}^{(i+1)} \setminus \overline{T}^{(i)} : (\overline{r}, r_{\tilde{\tau}}) \in \overline{C} \\
& \overset{INV_2^{(i)} \to P_{\mathcal{E}}(\tau,\overline{r})}{\Longrightarrow} \quad INV_2^{(i)} \land \forall \tilde{\tau} \in \overline{T}^{(i+1)} \setminus \overline{T}^{(i)} : P_{\mathcal{E}}(\tau, r_{\tilde{\tau}}) \\
& \Longrightarrow \quad INV_2^{(i+1)}
\end{aligned}$$

$$\begin{aligned}
& INV_3^{(i)} \land \varepsilon(\overline{r}) \notin \mathcal{E} \land \left(\forall r' \in \overline{\tau} : r' \neq \overline{r} \to \exists \tilde{\tau} \in \overline{T}^{(i+1)} : r' \in \tilde{\tau}\right) \\
& \Longrightarrow \quad \forall r \in \tau : \left(\varepsilon(r) \in \mathcal{E} \land P_{\mathcal{E}}(\tau, r)\right) \to \left(r \in \mathcal{R}^{(i)} \lor \exists \tilde{\tau} \in \overline{T}^{(i+1)} : r \in \tilde{\tau}\right) \\
& \overset{\mathcal{R}^{(i+1)} = \mathcal{R}^{(i)}}{\Longrightarrow} \quad INV_3^{(i+1)}
\end{aligned}$$

□

Because TAILORREQUEST processes each request $r \in \tau$ at most once, it will always finish and $\overline{T} = \emptyset$ after the final iteration. Thus, we can conclude the following from $INV_3^{(\log)}$:

$$\forall r \in \tau : \left(\varepsilon(r) \in \mathcal{E} \land P_{\mathcal{E}}(\tau, r)\right) \to r \in \mathcal{R}'$$

We, therefore, can derive the following corollary from $INV_1^{(\log)} \land INV_3^{(\log)}$:

Corollary 13.2.1
Given a well-formed non-empty trace τ and a set of endpoints $\mathcal{E}$, TAILORREQUEST (Algorithm 7.1) returns request logs $\mathcal{R}'$ to which the following applies:

$$\forall r \in \tau : \left(r \in \mathcal{R}' \leftrightarrow \varepsilon(r) \in \mathcal{E} \land P_{\mathcal{E}}(\tau, r)\right)$$

Proof (Theorem 13.1). We can prove Theorem 13.1 using Corollary 13.2.1. Regarding the request logs $\mathcal{R}'$, TAILORREQUESTLOGS (Algorithm 7.2) does nothing else than invoking TAILORREQUEST for every trace $\tau \in \mathcal{T}$ and merging the return values. Let $\mathcal{R}'[\tau]$ be the $\mathcal{R}'$ from the corollary for a fixed τ.

Then, we can follow:

$$\begin{aligned}
&\forall r \in R : \left(r \in \mathcal{R}' \leftrightarrow \exists \tau \in \overline{T} : r \in \mathcal{R}'[\tau]\right) \\
&\overset{\text{Corollary 13.2.1}}{\Longrightarrow} \quad \forall r \in R : (r \in \mathcal{R}' \leftrightarrow \exists \tau \in \mathcal{T} : \varepsilon(r) \in \mathcal{E} \land P_{\mathcal{E}}(\tau, r)) \\
&\Longrightarrow \quad A^{(\text{log})}_{\text{req}}
\end{aligned}$$

□

13.1.1.2. Session IDs — $A^{(\text{log})}_{\text{ID}}$

For the postcondition related to the session IDs, we define the following theorem, which we will prove.

Theorem 13.3 ($A^{(\text{log})}_{\text{ID}}$)
Given a set $\mathcal{T}$ of well-formed non-empty traces, a mapping $\phi : \mathcal{R} \to \mathbb{R}$ of the traces' root requests to session IDs, and a set of endpoints $\mathcal{E}$, TAILORREQUESTLOGS (Algorithm 7.2) returns a session ID mapping $\phi' : \mathcal{R}' \to \mathbb{R}$ to which the following applies:

$$\forall r \in \mathcal{R}' \ \forall \tau = (r_\tau, R_\tau, C_\tau) \in \mathcal{T} : r \in \tau \to \phi'(r) = \phi(r_\tau)$$

Proof. From Corollary 1, it follows that all requests in $\mathcal{R}''$ (line 4 in TAILORREQUESTLOGS) are derived from Trace τ. Hence, as the algorithm explicitly sets the session ID of these traces to $\phi(\tau)$ (line 5), $A^{(\text{log})}_{\text{ID}}$ follows automatically.

□

13.1.2. Model-based Tailoring

For the model-based tailoring algorithm (Algorithm 7.5), we defined three postconditions $A^{(\text{model})}_{\text{states}}$, $A^{(\text{model})}_{\text{prob}}$, and $A^{(\text{model})}_{\text{time}}$ (see Section 7.5.1) concerning the states, probabilities, and timing behavior of the tailored Markov chains. In the following, we show that the algorithm fulfills all three postconditions.

13.1.2.1. States — $A^{(\text{model})}_{\text{states}}$

The first postcondition describes the requirements of the states of the tailored Markov chains. As defined in Section 7.5.1, the following theorem needs to hold.

Theorem 13.4 ($A^{(\text{model})}_{\text{states}}$)
Let $\mathcal{W} = (\{W_1, \dots, W_q\}, f)$ be a workload model with states $\Sigma_j = \mathcal{E}^{(orig)}$ for $1 \le j \le q$, $\mathcal{T}$ a set of traces with root requests to endpoints in $\mathcal{E}^{(orig)}$, and $\mathcal{E}$ a set of endpoints. Let T_j furthermore be the traces corresponding to W_j and s_τ the session of requests in a trace τ targeting $\mathcal{E}$, i.e., for $W_j = (\Sigma_j, \varepsilon_j, p_j, \Delta_j, S_j)$:

$$T_j := \left\{\tau \in \mathcal{T} \mid \exists s \in S_j : r_\tau \in s\right\}$$
$$s_\tau := \left\{r \in \tau \mid \ \varepsilon(r) \in \mathcal{E} \wedge P_{\mathcal{E}}(\tau, r)\right\}$$

Given $\mathcal{W}$, $\mathcal{T}$, and $\mathcal{E}$, TAILORWORKLOADMODEL (Algorithm 7.5) returns a workload model $\mathcal{W}' = (\{W'_1, \dots, W'_q\}, f)$ such that the following applies to each W'_j:

$$\forall \tau \in T_j \ \forall r \in s_\tau \ \exists \tilde{e} \in \Sigma'_j : \varepsilon(r) = \varepsilon'_j(\tilde{e})$$

We prove the theorem by first considering each iteration of the inner loop of TAILORWORKLOADMODEL. For that, we state and prove the following lemma.

Lemma 13.5
Let $\mathcal{W}$, $\mathcal{T}$, $\mathcal{E}$, T_j, and s_τ be as in Theorem 13.4. After each iteration of the inner loop of TAILORWORKLOADMODEL (lines 4 to 15), the following applies to each W'_j, with e being the state processed in the iteration:

$$A_{13.5} \equiv \forall \tau \in T_j : \left(\varepsilon(r_\tau) = \varepsilon_j(e)\right) \to \left(\forall r \in s_\tau \ \exists \tilde{e} \in \Sigma'_j : \varepsilon(r) = \varepsilon'_j(\tilde{e})\right)$$

Proof. As stated in Theorem 13.1, the $\mathcal{R}'$ in line 7 contains exactly those requests from traces in $\mathcal{T}'$ (line 5) that target an endpoint in $\mathcal{E}$:

$$\begin{aligned} A_{13.5.1} &\equiv \forall r \in R : \big(r \in \mathcal{R}' \leftrightarrow \exists \tau \in \mathcal{T}' : \varepsilon(r) \in \mathcal{E} \wedge P_{\mathcal{E}}(\tau, r)\big) \\ &\iff \forall r \in R : \big(r \in \mathcal{R}' \leftrightarrow \exists \tau \in \mathcal{T}' : r \in s_\tau\big) \end{aligned}$$

Based on that, we differentiate between two *if-else* cases (lines 8 and 10). *Case $\mathcal{R}' = \emptyset$:* As it holds $r \notin \mathcal{R}'$ for all r, we can follow from $A_{13.5.1}$:

$$\forall r \in R \; \forall \tau \in \mathcal{T}' : r \notin s_\tau \Longrightarrow \forall \tau \in \mathcal{T}' : s_\tau = \emptyset \Longrightarrow A_{13.5}$$

Case $\mathcal{R}' \neq \emptyset$: First, the following is easy to see:

$$A_{13.5.2} \equiv \tau \in \mathcal{T}' \leftrightarrow \tau \in T_j \wedge \varepsilon(t_\tau) = \varepsilon_j(e)$$

REPLACESTATE adds states with exactly those endpoints to Σ'_j for which a request in $\mathcal{R}'$ exists (line 4 in Algorithm 7.4). In other words, it adds a state for each request:

$$\begin{aligned} &\forall r \in \mathcal{R}' \; \exists \tilde{e} \in \Sigma'_j : \varepsilon(r) = \varepsilon'_j(\tilde{e}) \\ &\overset{A_{13.5.1}}{\Longrightarrow} \forall \tau \in \mathcal{T}' \; \forall r \in s_\tau \; \exists \tilde{e} \in \Sigma'_j : \varepsilon(r) = \varepsilon'_j(\tilde{e}) \\ &\overset{A_{13.5.2}}{\Longrightarrow} \forall \tau \in T_j : \big(\varepsilon(t_\tau) = \varepsilon_j(e)\big) \rightarrow \big(\forall r \in s_\tau \; \exists \tilde{e} \in \Sigma'_j : \varepsilon(r) = \varepsilon'_j(\tilde{e})\big) \\ &\Longrightarrow A_{13.5} \end{aligned}$$

□

Proof (Theorem 13.4). We can prove Theorem 13.4 using Lemma 13.5. Because TAILORWORKLOADMODEL processes each $e \in \Sigma_j$ once, $A_{13.5}$ holds for all of them. Using the precondition of Theorem 13.4 that all traces in $\mathcal{T}$ (and, thus, in T_j) target an endpoint in $\mathcal{E}$, we conclude the following:

$$\begin{aligned}
&\forall e \in \Sigma_j : A_{13.5} \\
&\quad\Longrightarrow \quad \forall \tau \in T_j : \big(\exists e \in \Sigma_j : \varepsilon(t_\tau) = \varepsilon_j(e)\big) \\
&\qquad\qquad\qquad \to \big(\forall r \in s_\tau\ \exists \tilde{e} \in \Sigma'_j : \varepsilon(r) = \varepsilon'_j(\tilde{e})\big) \\
&\overset{\text{precondition}}{\Longrightarrow} \quad A^{(\text{model})}_{\text{states}}
\end{aligned}$$

□

13.1.2.2. Probabilities — $A^{(\text{model})}_{\text{prob}}$

The second postcondition targets the probabilities of reaching one state from another. As stated in Section 7.5.1, the tailoring needs to preserve them.

Theorem 13.6 ($A^{(\text{model})}_{\text{prob}}$)
Let $\mathcal{W}$, $\mathcal{T}$, $\mathcal{E}$, $\mathcal{E}^{(orig)}$, T_j, and s_τ be as in Theorem 13.4. Let $p_j^\infty(e_1, e_2)$ furthermore be the probability to reach state e_2 from e_1 in any number of steps (see Section 7.5.1). Given $\mathcal{W}$, $\mathcal{T}$, and $\mathcal{E}$, TAILORWORKLOADMODEL (Algorithm 7.5) returns a workload model $\mathcal{W}' = (\{W'_1, \dots, W'_q\}, f)$ such that the following applies to each W'_j:

$$\begin{aligned}
\forall \tilde{e}_1 \in \Sigma'_j\ \forall e_2 \in \mathcal{E}^{(orig)} : \big(\text{sub}_j(e_2) \neq \emptyset \wedge \tilde{e}_1 \notin \text{sub}_j(e_2)\big) \\
\to \Big(p_j'^{\infty}(\tilde{e}_1, \text{sub}_j(e_2)) = p_j^\infty(\text{root}_j(\tilde{e}_1), e_2)\Big)
\end{aligned}$$

For proving the theorem, we introduce several notations and conditions. First, we consider the probability to reach the final state \$ from the any state e in any of the Markov chains occurring in this context. By construction, they are absorbing Markov chains, i.e., each state can reach \$ (Grinstead and Snell, 2012). Thus, we can state the following lemma without proof.

Lemma 13.7
Let $W = (\Sigma, p, \Delta, S)$ be any Markov chain occurring in the context of this chapter with final state $\$ \in \Sigma$. Let $p^\infty(e_1, e_2)$ furthermore be as in Theorem 13.6. Then, the following always applies to W:

$$\forall e \in \Sigma : p^{\infty}(e, \$) = 1$$

Second, we introduce the notion of a path through a Markov chain. Paths will be useful for proving the overall probability of reaching one state from another. Thus, we define the following.

Definition 13.1 (Paths)
Let $W = (\Sigma, p, \Delta, S)$ be a Markov chain. We call a sequence of states $\pi = (e_1, \dots, e_{k+1}) \in \Sigma^*$ a path *through W. k is the length of π, e_1 the start state, and e_{k+1} the end state. Furthermore, the probability of π is the product of the probabilities of all transitions of the path:*

$$p(\pi) = \prod_{i=1}^{k} p(e_i, e_{i+1})$$

$\Pi(e, e')$ denotes the set of all paths from state e to e' of any length. We denote the concatenation of paths $\pi_1 \in \Pi(e, e')$ and $\pi_2 \in \Pi(e', e'')$ by $\pi_1 \circ \pi_2 = (e, \dots, e', \dots, e'') \in \Pi(e, e'')$.

Next, we define and prove postconditions for the helper algorithms. Later, we will use them to prove Theorem 13.6. For REMOVESTATE, we state the following.

Lemma 13.8
Let $W_j = (\Sigma_j, \varepsilon_j, p_j, \Delta_j, S_j)$ be a Markov chain as usual, $e \in \Sigma_j$ a state, and $\delta \in \mathbb{R}$. Let furthermore be $\Sigma_j^{(b)}$ and $\Sigma_j^{(a)}$ the Σ_j before and after the execution of REMOVESTATE (Algorithm 7.3) and π a path in $\Sigma_j^{(a)}$. Finally, let $\pi^{(i)} = (e_1, \dots, e_i, e, e_{i+1}, \dots e_{k+1})$ be the path resulting from π by adding e after e_i and, correspondingly, $\pi^{(i_1, \dots, i_q)}$ the path resulting from adding e after e_{i_1} to e_{i_q} each.

Then, the following applies to π for all $\sigma = (1, \dots, 1, 2, \dots, 2, \dots, k, \dots, k) \in \{1, \dots, k\}^{l_1 + \dots + l_k}$, $l_i \geq 0$:

$$A_{13.8} \equiv p_j^{(a)}(\pi) = \sum_{\sigma} p_j^{(b)}(\pi^{(\sigma)})$$

Proof. We prove the lemma by induction over the length k of path π. However, we first extract the transition probability $p_j^{(\mathrm{a})}(e', e'')$ for all pairs e', e'' of endpoints in $\Sigma_j^{(\mathrm{a})}$. According to lines 4, 8, and 10 of REMOVESTATE, it is:

$$p_j^{(\mathrm{a})}(e', e'') = p_j^{(\mathrm{b})}(e', e'') + \frac{p_j^{(\mathrm{b})}(e', e) \cdot p_j^{(\mathrm{b})}(e, e'')}{1 - p_j^{(\mathrm{b})}(e, e)}$$

Base Case ($k = 1$): In this case, $\pi = (e_1, e_2)$ and (using geometric series, Hildebrandt, 2006):

$$\begin{aligned}
\sum_{\sigma} p_j^{(\mathrm{b})}(\pi^{(\sigma)}) &= \sum_{i=0}^{\infty} p_j^{(\mathrm{b})}(\pi^{(\overbrace{1, \ldots, 1}^{i \text{ times}})}) \\
&= p_j^{(\mathrm{b})}(e_1, e_2) + \sum_{i=0}^{\infty} p_j^{(\mathrm{b})}(e_1, e) \cdot p_j^{(\mathrm{b})}(e, e)^i \cdot p_j^{(\mathrm{b})}(e, e_2) \\
&= p_j^{(\mathrm{b})}(e_1, e_2) + \frac{p_j^{(\mathrm{b})}(e_1, e) \cdot p_j^{(\mathrm{b})}(e, e_2)}{1 - p_j^{(\mathrm{b})}(e, e)} \\
&= p_j^{(\mathrm{a})}(e_1, e_2) = p_j^{(\mathrm{a})}(\pi)
\end{aligned}$$

Inductive Step ($k \rightsquigarrow k + 1$): Given a path $\pi_k = (e_1, \ldots, e_{k+1})$ in $\Sigma_j^{(\mathrm{a})}$ of length k to which $A_{13.8}$ applies and $e_{k+2} \in \Sigma_j^{(\mathrm{a})}$, $A_{13.8}$ applies to the path $\pi_{k+1} = \pi_k \circ (e_{k+1}, e_{k+2})$ of length $k + 1$:

$$\begin{aligned}
\sum_{\sigma} p_j^{(\mathrm{b})}(\pi_{k+1}^{(\sigma)}) &= \sum_{\sigma} p_j^{(\mathrm{b})}(\pi_k^{(\sigma)}) \cdot \sum_{\sigma'} p_j^{(\mathrm{b})}((e_{k+1}, e_{k+2})^{(\sigma')}) \\
&= p_j^{(\mathrm{a})}(\pi_k) \cdot \Big(p_j^{(\mathrm{b})}(e_{k+1}, e_{k+2}) \\
&\quad + \sum_{i=0}^{\infty} p_j^{(\mathrm{b})}(e_{k+1}, e) \cdot p_j^{(\mathrm{b})}(e, e)^i \cdot p_j^{(\mathrm{b})}(e, e_{k+2}) \Big) \\
&= p_j^{(\mathrm{a})}(\pi_k) \cdot \left(p_j^{(\mathrm{b})}(e_{k+1}, e_{k+2}) + \frac{p_j^{(\mathrm{b})}(e_{k+1}, e) \cdot p_j^{(\mathrm{b})}(e, e_{k+2})}{1 - p_j^{(\mathrm{b})}(e, e)} \right) \\
&= p_j^{(\mathrm{a})}(\pi_k) \cdot p_j^{(\mathrm{a})}(e_{k+1}, e_{k+2}) = p_j^{(\mathrm{a})}(\pi_{k+1})
\end{aligned}$$

□

We can interpret Lemma 13.8 as follows. Each path π in $\Sigma_j^{(\mathrm{a})}$ subsumes π in $\Sigma_j^{(\mathrm{b})}$ and, additionally, all paths in $\Sigma_j^{(\mathrm{b})}$ that only differ from π in one or several visits of the removed state e. Hence, π in $\Sigma_j^{(\mathrm{a})}$ preserves the probability "removed" by removing e. As this holds for all paths from state e_1 to e_2, we can follow the corollary below.

Corollary 13.8.1
Let $W_j = (\Sigma_j, \varepsilon_j, p_j, \Delta_j, S_j)$ be a Markov chain as usual, $e \in \Sigma_j$ a state, and $\delta \in \mathbb{R}$. Given these inputs, REMOVESTATE *(Algorithm 7.3) changes W_j such that the following applies to the version before and after the execution:*

$$A_{13.8.1} \equiv \forall e_1, e_2 \in \Sigma_j^{(a)} : p_j^{(a)\infty}(e_1, e_2) = p_j^{(b)\infty}(e_1, e_2)$$

Now, we consider REPLACESTATE, for which we state the following.

Lemma 13.9
Let $W_j = (\Sigma_j, \varepsilon_j, p_j, \Delta_j, S_j)$ be a Markov chain as usual, $e \in \Sigma_j$ a state, and $\mathcal{R}'$ request logs. Let W' furthermore be the aggregation of $\mathcal{R}'$, as in REPLACESTATE *(Algorithm 7.4). Given W_j, e, and $\mathcal{R}'$,* REPLACESTATE *changes W_j such that the following statements apply to the version before ($W_j^{(b)}$) and after ($W_j^{(a)}$) the execution:*

$$A_{13.9.1} \equiv \forall e_1 \in \Sigma_j^{(a)} \setminus \Sigma' : p_j^{(a)\infty}(e_1, \mathrm{sub}_j(e)) = p_j^{(b)\infty}(e_1, e)$$
$$A_{13.9.2} \equiv \forall e_1 \in \Sigma' \; \forall e_2 \in \Sigma_j^{(a)} \setminus \Sigma' : p_j^{(a)\infty}(e_1, e_2) = p_j^{(b)\infty}(e, e_2)$$
$$A_{13.9.3} \equiv \forall e_1, e_2 \in \Sigma_j^{(a)} \setminus \Sigma' : p_j^{(a)\infty}(e_1, e_2) = p_j^{(b)\infty}(e_1, e_2)$$

Proof. We prove Lemma 13.9 by considering the intermediate Markov chain that REPLACESTATE constructs after line 13. We denote this Markov chain as $W_j^{(\mathrm{int})}$. The only difference to $W_j^{(\mathrm{a})}$ is that it contains the inner initial and final states I' and $\$'$. Thus, we can use Corollary 13.8.1 for deriving conclusions about $W_j^{(\mathrm{a})}$. We prove each of the statements individually.

$A_{13.9.1}$: In $W_j^{(\mathrm{int})}$, I' is the only state connecting W' with the rest of the Markov chain via incoming transitions. Hence, the probability of reaching a state in Σ' from outside is equal to reaching it via I'. Furthermore, as I' has the incoming transition probabilities from e (line 6), we can conclude:

$$\begin{aligned}
&\forall e_1 \in \Sigma_j^{(\text{int})} \setminus \Sigma' \ \forall e_2 \in \Sigma' : p_j^{(\text{int})\infty}(e_1, e_2) = p_j^{(\text{int})\infty}(e_1, I') \cdot p_j^{(\text{int})\infty}(I', e_2)\\
\Longrightarrow\quad &\forall e_1 \in \Sigma_j^{(\text{int})} \setminus \Sigma' : p_j^{(\text{int})\infty}(e_1, \text{sub}_j(e)) = p_j^{(\text{int})\infty}(e_1, I')\\
\Longrightarrow\quad &\forall e_1 \in \Sigma_j^{(\text{int})} \setminus \Sigma' : p_j^{(\text{int})\infty}(e_1, \text{sub}_j(e)) = p_j^{(\text{b})\infty}(e_1, e)\\
\overset{\text{Corollary 13.8.1}}{\Longrightarrow}\quad &A_{13.9.1}
\end{aligned}$$

$A_{13.9.2}$: In $W_j^{(\text{int})}$, $\$'$ is the only state connecting W' with the rest of the Markov chain via outgoing transitions. Hence, the probability of reaching a state outside Σ' from a state inside is equal to reaching it via $\$'$. Furthermore, as $\$'$ has the outgoing transition probabilities from e (line 10), we can conclude:

$$\begin{aligned}
&\forall e_1 \Sigma' \ \forall e_2 \in \Sigma_j^{(\text{int})} \setminus \Sigma' : p_j^{(\text{int})\infty}(e_1, e_2) = p_j^{(\text{int})\infty}(e_1, \$') \cdot p_j^{(\text{int})\infty}(\$', e_2)\\
\overset{\text{Lemma 13.7}}{\Longrightarrow}\quad &\forall e_1 \Sigma' \ \forall e_2 \in \Sigma_j^{(\text{int})} \setminus \Sigma' : p_j^{(\text{int})\infty}(e_1, e_2) = 1 \cdot p_j^{(\text{int})\infty}(\$', e_2)\\
\Longrightarrow\quad &\forall e_1 \Sigma' \ \forall e_2 \in \Sigma_j^{(\text{int})} \setminus \Sigma' : p_j^{(\text{int})\infty}(e_1, e_2) = p_j^{(\text{b})\infty}(e, e_2)\\
\overset{\text{Corollary 13.8.1}}{\Longrightarrow}\quad &A_{13.9.2}
\end{aligned}$$

$A_{13.9.3}$: In $W_j^{(\text{b})}$, a state $e_2 \neq e$ can be reached from another state $e_1 \neq e$ either by passing e or by not passing e. Similarly, e_2 can be reached from e_1 in $W_j^{(\text{int})}$ either by passing the replacement of e or not. REPLACESTATE does not change any parts of $W_j^{(\text{b})}$ except for e. Thus, the probability of reaching e_2 from e_1 without passing e or its replacement is equal in both Markov chains. Hence, we only need to show the part with e remains unchanged. The probability of reaching e_2 via the replacement is to reach I', then reach $\$'$, and then reach e_2.

Let X be the probability to reach e_2 from e_1 without passing e or its replacement. Then, we can state:

$$\begin{aligned}p_j^{(\mathrm{int})\infty}(e_1,e_2) &= p_j^{(\mathrm{int})\infty}(e_1,I')\cdot p_j^{(\mathrm{int})\infty}(I',\$')\cdot p_j^{(\mathrm{int})\infty}(\$',e_2)+X\\ &= p_j^{(\mathrm{b})\infty}(e_1,e)\cdot 1\cdot p_j^{(\mathrm{b})\infty}(e,e_2)+X\\ &= p_j^{(\mathrm{b})\infty}(e_1,e_2)\end{aligned}$$

Therefore, according to Lemma 13.7, $A_{13.9.3}$ applies to $W_j^{(\mathrm{a})}$. □

Proof (Theorem 13.6). Finally, we can prove Theorem 13.6 using the lemmata and corollaries defined above. We do that by induction over the iteration of the inner *for*-loop (lines 4 to 15) of TAILORWORKLOADMODEL. In each iteration, we prove that $A_{\mathrm{prob}}^{(\mathrm{model})}$ holds under the assumption that $\mathcal{E}$ contains only those endpoints the algorithm already processed; we denote that as $A_{\mathrm{prob}}^{(\mathrm{model},i)}$. Hence, $A_{\mathrm{prob}}^{(\mathrm{model})}$ follows from $A_{\mathrm{prob}}^{(\mathrm{model},i)}$ after the last iteration. We denote the version of W_j' after the i-th iteration as $W_j^{(i)}$.

Base Case: For the base case, i.e., before the execution of the algorithm, it is $\mathrm{root}_j(e) = e$ for each $e \in W_j^{(0)}$. Hence, $A_{\mathrm{prob}}^{(\mathrm{model},0)}$ holds.

Inductive Step ($i \rightsquigarrow i+1$): We assume $A_{\mathrm{prob}}^{(\mathrm{model},i)}$ holds after the i-th iteration and consider state e to be processed in iteration $i+1$. Furthermore, we distinguish between the following two cases.

Case $\mathcal{R}' = \emptyset$: In this case, the algorithm only makes one call to REMOVE-STATE. According to Corollary 13.8.1, this does not change the probabilities of reaching one state from another; thus, $A_{\mathrm{prob}}^{(\mathrm{model},i+1)}$ holds, too.

Case $\mathcal{R}' \neq \emptyset$: In this case, the algorithm replaces the state e with a sub-Markov chain consisting of the states $\mathrm{sub}_j(e)$. According to Lemma 13.9, the statements $A_{13.9.1}$ to $A_{13.9.3}$ hold. As the probabilities to reach one state outside $\mathrm{sub}_j(e)$ from another is unchanged ($A_{13.9.3}$), we only need to consider the combination from one state outside and one inside $\mathrm{sub}_j(e)$.

$$\begin{aligned}
&A_{13.9.1} \\
&\Longrightarrow \forall \tilde{e}_1 \in \Sigma_j^{(i+1)} \setminus \mathrm{sub}_j(e) : p_j^{(i+1)\infty}(\tilde{e}_1, \mathrm{sub}_j(e)) = p_j^{(i)\infty}(\tilde{e}_1, e) \\
&\overset{A_{\mathrm{prob}}^{(\mathrm{model},i)}}{\Longrightarrow} \forall \tilde{e}_1 \in \Sigma_j^{(i+1)} \setminus \mathrm{sub}_j(e) : p_j^{(i+1)\infty}(\tilde{e}_1, \mathrm{sub}_j(e)) = p_j^{\infty}(\mathrm{root}_j(\tilde{e}_1), e)
\end{aligned}$$

$$\begin{aligned}
&A_{13.9.2} \\
&\Longrightarrow \forall \tilde{e}_1 \in \mathrm{sub}_j(e)\ \forall \tilde{e}_2 \in \Sigma_j^{(i+1)} \setminus \mathrm{sub}_j(e) : \\
&\qquad p_j^{(i+1)\infty}(\tilde{e}_1, \tilde{e}_2) = p_j^{(i)\infty}(e, \tilde{e}_2) \\
&\overset{A_{\mathrm{prob}}^{(\mathrm{model},i)}}{\Longrightarrow} \forall \tilde{e}_1 \in \mathrm{sub}_j(e)\ \forall e_2 \in \mathcal{E}^{(\mathrm{orig})} \setminus \{e\} : \\
&\qquad p_j^{(i+1)\infty}(\tilde{e}_1, \mathrm{sub}_j(e_2)) = p_j^{(i)\infty}(e, \mathrm{sub}_j(e_2)) = p_j^{\infty}(e, e_2)
\end{aligned}$$

Hence, we can follow:

$$A_{\mathrm{prob}}^{(\mathrm{model},i)} \land A_{13.9.1} \land A_{13.9.2} \land A_{13.9.3} \Longrightarrow A_{\mathrm{prob}}^{(\mathrm{model},i+1)}$$

□

13.1.2.3. Time — $A_{\mathrm{time}}^{(\mathrm{model})}$

The last postcondition targets the time the Markov chain takes to execute. We decided to use a model with normally distributed think times. In the algorithms, there are operations on the Markov chain that require deviating from the actual think time distribution, for sticking to normal distributions. However, they always preserve the think time mean and model the variation as accurately as possible. Therefore, we validate the postcondition only based on the think time mean. In an experimental evaluation, we will assess the goodness of the tailored Markov chains without this restriction.

First, we define the notion of time in a Markov chain.

Definition 13.2 (Time)
Let $W_j = (\Sigma_j, \varepsilon_j, p_j, \Delta_j, S_j)$ be a Markov chain as usual and $e, e_1, e_2 \in \Sigma_j$ states. Then, $\mu_j(e_1, e_2)$ defines the average think time of a transition (e_1, e_2), i.e., for

a certain σ:

$$\Delta_j(e_1, e_2) \sim \mathcal{N}(\mu_j(e_1, e_2), \sigma)$$

Furthermore, $\mu_j(e)$ defines the average response time of a request to $\varepsilon_j(e)$.

For a path $\pi \in \Pi(e_1, e_2)$ of length k in Σ_j (see Definition 13.1), $\mu_j(\pi)$ describes the time it takes to walk the path:

$$\mu_j(\pi) = \mu_j(e_1, e_2) + \sum_{i=2}^{k} \mu_j(e_i) + \mu_j(e_i, e_{i+1})$$

With that, we can define $\mathrm{time}_j(e_1, e_2)$*, which denotes the average time it takes to come from state e_1 to e_2:*

$$\mathrm{time}_j(e_1, e_2) = \sum_{\pi \in \Pi(e_1, e_2)} p_j(\pi) \cdot \mu_j(\pi)$$

We state the following theorem as a postcondition for TAILORMARKOVCHAIN.

Theorem 13.10 ($A^{\text{(model)}}_{\text{time}}$)
Given $\mathcal{W}$, $\mathcal{T}$, and $\mathcal{E}$ as in Theorem 13.4, TAILORWORKLOADMODEL (Algorithm 7.5) returns a workload model $\mathcal{W}' = (\{W'_1, \ldots, W'_q\}, f)$ such that the following applies to each W'_j:

$$\mathrm{time}'_j(I, \$) = \mathrm{time}_j(I, \$)$$

First, we consider the Markov chains resulting from aggregating session logs. We use existing approaches for that and assume they create representative Markov chains, which also includes the timing. Therefore, we state the following lemma without proof.

Lemma 13.11
Let $\mathcal{S}$ be session logs and $W = (\Sigma, p, \Delta, \mathcal{S})$ the Markov chain resulting from aggregating $\mathcal{S}$. Then, the average time it takes to execute W is equal to the

average duration of each session:

$$\text{time}(W) = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \max_{r \in s}(t(r) + \delta(r)) - \min_{r \in s}(t(r))$$

With that, we can show the two helper algorithms preserve $A_{\text{time}}^{(\text{model})}$. We start with REMOVESTATE.

Lemma 13.12
Let $W_j = (\Sigma_j, \varepsilon_j, p_j, \Delta_j, S_j)$ be a Markov chain as usual, $e \in \Sigma_j$ a state, and $\delta \sim \mathcal{N}(\mu_j(e), \sigma)$. Let furthermore be $\Sigma_j^{(b)}$ and $\Sigma_j^{(a)}$ the Σ_j before and after the execution of REMOVESTATE *(Algorithm 7.3) and π a path in $\Sigma_j^{(a)}$. Finally, let $\pi^{(i_1,\ldots,i_q)}$ be as in Lemma 13.8.*

Then, the following applies to π for all $\sigma = (1,\ldots,1,2,\ldots,2,\ldots,k,\ldots,k) \in \{1,\ldots,k\}^{l_1+\cdots+l_k}$, $l_i \geq 0$:

$$A_{13.12} \equiv p_j^{(a)}(\pi) \cdot \mu_j^{(a)}(\pi) = \sum_{\sigma} p_j^{(b)}(\pi^{(\sigma)}) \cdot \mu_j^{(b)}(\pi^{(\sigma)})$$

Proof. From lines 5 and 9 of REMOVESTATE, we can extract the think time mean of each transition (e_1, e_2) after the execution. Given $\alpha = \frac{1}{1-p_j^{(\mathrm{b})}(e,e)} - 1$, it is:

$$\begin{aligned}\mu_j^{(\mathrm{a})}(e_1,e_2) = {} & \frac{p_j^{(\mathrm{b})}(e_1,e_2)}{p_j^{(\mathrm{a})}(e_1,e_2)} \cdot \mu_j^{(\mathrm{b})}(e_1,e_2) + \frac{p_j^{(\mathrm{b})}(e_1,e) \cdot p_j^{(\mathrm{b})}(e,e_2)}{p_j^{(\mathrm{a})}(e_1,e_2) \cdot (1 - p_j^{(\mathrm{b})}(e,e))} \\ & \cdot \left(\mu_j^{(\mathrm{b})}(e_1,e) + \alpha \cdot \mu_j^{(\mathrm{b})}(e,e) + (\alpha+1) \cdot \mu_j(e) + \mu_j^{(\mathrm{b})}(e,e_2)\right)\end{aligned}$$

Then, we prove the lemma by induction over the length k of the path π.

Base Case ($k = 1$): In this case, $\pi = (e_1, e_2)$ and:

$$
\begin{aligned}
&\sum_{\sigma} p_j^{(\mathrm{b})}(\pi^{(\sigma)}) \cdot \mu_j^{(\mathrm{b})}(\pi^{(\sigma)}) \\
&\quad = \sum_{i=0}^{\infty} p_j^{(\mathrm{b})}(\pi^{(\sigma)}) \cdot \mu_j^{(\mathrm{b})}(\pi^{(\overbrace{1,\ldots,1}^{i\text{ times}})}) \\
&\quad = p_j^{(\mathrm{b})}(e_1, e_2) \cdot \mu_j^{(\mathrm{b})}(e_1, e_2) + \sum_{i=0}^{\infty} p_j^{(\mathrm{b})}(e_1, e) \cdot p_j^{(\mathrm{b})}(e, e)^i \cdot p_j^{(\mathrm{b})}(e, e_2) \\
&\qquad \cdot \left(\mu_j^{(\mathrm{b})}(e_1, e) + i \cdot \mu_j^{(\mathrm{b})}(e, e) + (i+1) \cdot \mu_j(e) + \mu_j^{(\mathrm{b})}(e, e_2)\right) \\
&\quad = p_j^{(\mathrm{b})}(e_1, e_2) \cdot \mu_j^{(\mathrm{b})}(e_1, e_2) + \left(p_j^{(\mathrm{b})}(e_1, e) \cdot p_j^{(\mathrm{b})}(e, e_2)\right) \\
&\qquad \cdot \left(\frac{\mu_j^{(\mathrm{b})}(e_1, e) \cdot \mu_j^{(\mathrm{b})}(e, e_2)}{1 - \mu_j^{(\mathrm{b})}(e, e)} + \frac{p_j^{(\mathrm{b})}(e, e) \cdot \mu_j^{(\mathrm{b})}(e, e)}{\left(1 - \mu_j^{(\mathrm{b})}(e, e)\right)^2} + \frac{\mu_j(e)}{\left(1 - \mu_j^{(\mathrm{b})}(e, e)\right)^2} \right) \\
&\quad = p_j^{(\mathrm{a})}(\pi) \cdot \mu_j^{(\mathrm{a})}(\pi)
\end{aligned}
$$

Inductive Step ($k \rightsquigarrow k + 1$): We reuse the base case and Lemma 13.8. Given a path $\pi_k = (e_1, \ldots, e_{k+1})$ in $\Sigma_j^{(\mathrm{a})}$ of length k to which $A_{13.12}$ applies and $e_{k+2} \in \Sigma_j^{(\mathrm{a})}$, $A_{13.12}$ applies to the path $\pi_{k+1} = \pi_k \circ (e_{k+1}, e_{k+2})$ of length $k + 1$:

$$
\begin{aligned}
&\sum_{\sigma} p_j^{(\mathrm{b})}(\pi_{k+1}^{(\sigma)}) \cdot \mu_j^{(\mathrm{b})}(\pi_{k+1}^{(\sigma)}) \\
&\quad = \sum_{\sigma} \sum_{\sigma'} p_j^{(\mathrm{b})}(\pi_k^{(\sigma)}) \cdot p_j^{(\mathrm{b})}((e_{k+1}, e_{k+2})^{(\sigma')}) \\
&\qquad\qquad \cdot \left(\mu_j^{(\mathrm{b})}(\pi_k^{(\sigma)}) + \mu_j(e_{k+1}) + \mu_j^{(\mathrm{b})}((e_{k+1}, e_{k+2})^{(\sigma')})\right) \\
&\quad = p_j^{(\mathrm{a})}(\pi_k) \cdot p_j^{(\mathrm{a})}(e_{k+1}, e_{k+2}) \cdot \left(\mu_j^{(\mathrm{a})}(\pi_k) + \mu_j(e_{k+1}) + \mu_j^{(\mathrm{a})}(e_{k+1}, e_{k+2})\right) \\
&\quad = p_j^{(\mathrm{a})}(\pi_{k+1}) \cdot \mu_j^{(\mathrm{a})}(\pi_{k+1})
\end{aligned}
$$

□

Lemma 13.12 tells us that the modified probabilities and think times preserve the time REMOVESTATE "removes" from the Markov chain. As a consequence, the time spent in the Markov chain overall remains the same:

$$\begin{aligned}
\mathrm{time}_j^{(\mathrm{a})}(I,\$) &= \sum_{\pi\in\Pi^{(\mathrm{a})}} p_j^{(\mathrm{a})}(\pi)\cdot\mu_j^{(\mathrm{a})}(\pi)\\
&= \sum_{\pi\in\Pi^{(\mathrm{a})}} \sum_{\sigma} p_j^{(\mathrm{b})}(\pi^{(\sigma)})\cdot\mu_j^{(\mathrm{b})}(\pi^{(\sigma)})\\
&= \sum_{\pi\in\Pi^{(\mathrm{b})}} p_j^{(\mathrm{b})}(\pi)\cdot\mu_j^{(\mathrm{b})}(\pi) = \mathrm{time}_j^{(\mathrm{b})}(I,\$)
\end{aligned}$$

We record that in the following corollary.

Corollary 13.12.1
Let $W_j = (\Sigma_j, \varepsilon_j, p_j, \Delta_j, S_j)$ be a Markov chain as usual, $e \in \Sigma_j$ a state, and $\delta \sim \mathcal{N}(\mu_j(e), \sigma)$. Let furthermore be $W_j^{(b)}$ and $W_j^{(a)}$ the W_j before and after the execution of REMOVESTATE *(Algorithm 7.3). Then, the following holds:*

$$\mathrm{time}_j^{(a)}(I,\$) = \mathrm{time}_j^{(b)}(I,\$)$$

Next, we consider REPLACESTATE. Similar to REMOVESTATE, it preserves the time in the Markov chain, given that the input is appropriate.

Lemma 13.13
Let $W_j = (\Sigma_j, \varepsilon_j, p_j, \Delta_j, S_j)$ be a Markov chain as usual, $e \in \Sigma_j$ a state, and $\mathcal{R}'$ request logs. Let W' furthermore be the aggregation of $\mathcal{R}'$, as in REPLACESTATE *(Algorithm 7.4) with $\mathrm{time}(W') = \mu_j(e)$. Given W_j, e, and $\mathcal{R}'$,* REPLACESTATE *changes W_j such that the following statements apply to the version before ($W_j^{(b)}$) and after ($W_j^{(a)}$) the execution:*

$$\mathrm{time}_j^{(a)}(I,\$) = \mathrm{time}_j^{(b)}(I,\$)$$

Proof. The lemma follows directly from the fact that the request logs $\mathcal{R}'$ represent the average duration of e. Similar to Lemma 13.9, we consider the intermediate Markov chain $W_j^{(\mathrm{int})}$ after line 13. From Lemma 13.11, it follows that the time it takes on average to come from $I' \in \Sigma_j^{(\mathrm{int})}$ to $\$' \in \Sigma_j^{(\mathrm{int})}$ is equal to $\mu_j(e)$. Furthermore, the probability and time to come from any

$e' \in \Sigma_j^{(\text{int})} \setminus \Sigma'$ to I' are equal to the ones from e' to e in $W_j^{(\text{b})}$. The same applies to $\$'$ and e'. Hence, $W_j^{(\text{int})}$ preserves the time, as stated by Lemma 13.13. According to Corollary 13.12.1, the two calls to REMOVESTATE preserve the overall timing, too. □

Proof (Theorem 13.10). Finally, we can prove Theorem 13.10. We consider the inner *for*-loop of TAILORWORKLOADMODEL (lines 4 to 15 of Algorithm 7.5). We show that after each iteration, the average time to come from I to $\$$ is the same as before the iteration. First, we want to highlight that $\mathcal{T}'$ (line 5) contains those traces that belong to the Markov chain W_j and have a root request to endpoint e. Then, we distinguish between two cases.

Case $\mathcal{R}' = \emptyset$: In this case, e is removed from W_j. The duration passed to REMOVESTATE follows the distribution of the duration of the traces in $\mathcal{T}'$. Hence, by construction, $\delta(\mathcal{T}') = \mu_j(e)$. According to Corollary 13.12.1, the timing of the Markov chain remains unchanged on average.

Case $\mathcal{R}' \neq \emptyset$: Here, the algorithm extends $\mathcal{R}'$. Each trace τ has its own session ID and two placeholder requests I' and $\$'$, such that:

$$t(\$') - t(I') = t(\tau) + \delta(\tau) - t(\tau) = \delta(\tau)$$

Thus, the Markov chain that REPLACESTATE creates by aggregating $\mathcal{R}'$ will take the time $\mu_j(e)$ to execute on average. Consequently, the preserving of the timing of W_j follows from Lemma 13.13. □

13.1.3. Comparison of the Algorithms

Considering the load test extraction process (see Section 7.3), we can identify several differences between the service-tailoring algorithms. The log-based algorithm produces request logs that perfectly explain the workload. However, the workload clustering uses heuristics for classifying sessions, resulting in potentially imperfect workload models. The model-based tailoring suffers from this, too — it modifies the output from the session clustering — but at least preserves the basic structure of the untailored workload model. However, it approximates the think time variation (see Section 7.5.3).

Table 13.1.: Differences of Log-based and Model-based Service-tailoring

	log-based tailoring	**model-based tailoring**
states	one per endpoint in $\mathcal{E}$	one per endpoint in $\mathcal{E}$ and state in the original Markov chains
workload model structure	depends on the clustering	similar to the untailored Markov chains
transition probabilities	locally correct	locally correct & similar to the untailored workload model
think times	depend on the clustering	approximated variation

Under certain assumptions, the extraction process adjusted with log-based and model-based tailoring will output the same load test. These are the following:

- Each endpoint in $\mathcal{E}$ is only called within the context of one state of the untailored Markov chain.
- The workload clustering is perfect, i.e., it will categorize a tailored session similarly as its untailored correspondent.
- The model-based tailoring calculates the think time variations correctly.

In most of all cases, all of these assumptions are invalid. Therefore, we summarize several differences between the algorithms stemming from violations of the assumptions. We summarize them in Table 13.1. First, the states of the Markov chains can be different. While the model-based algorithm replaces the states of the original chains individually, the log-based algorithm produces one state per endpoint. Hence, the model-tailored Markov chains can have more states than the log-tailored ones. Notably, this is the case when the first of the above-listed assumptions is violated.

Second, the workload clustering is not perfect. That is, it can happen that the tailored session logs are being clustered differently than the untailored ones. This results in a different structure of the workload model — it may have a different number of Markov chains, different transition probabilities, and think times. However, it is formally undecidable whether these differences decrease representativeness. In contrast, the model-based algorithm preserves the structure of the untailored Markov chain.

Even though the workload model structure can differ, the transition probabilities of both tailored Markov chains are locally correct, i.e., the probability of requesting one endpoint after another is the same. This follows from $A_{\text{req}}^{(\text{log})}$ and $A_{\text{prob}}^{(\text{model})}$. However, the sessions generated by the tailored load tests can be different, because they depend on the workload clustering and the states.

Finally, we can identify differences in the think times of each transition. For the log-based algorithm, the workload clustering might introduce an error, especially to the variation. Notably, the type of think time distribution — e.g., normal distribution — might be inappropriate. Again, the model-based algorithm produces Markov chains with think times similar to the original Markov chains, but with an approximated variation.

We can conclude that even though we proved that both tailoring algorithms are correct regarding several requirements, the workload clustering and approximated think time variations introduce an error to the representativeness, which we cannot assess formally. Therefore, we conduct an experimental study for evaluating the approaches without assumptions and quantifying the differences between the algorithms.

13.2. Experimental Study with Sock Shop

In this study, we evaluate the following research questions.

- RQ2.2: *How representative are the workloads generated by the service-tailored load tests compared to an untailored and a request-based test?*
- RQ2.3: *To which degree do the service-tailored load tests impair the performance metrics of the tested services?*

- RQ2.5: *Which qualitative differences of the service-tailored workload models exist?*

We executed a series of experiments with the Sock Shop Microservices Demo (Weaveworks, Inc., 2020). In each experiment, we executed a load test tailored to a specific set of services using a particular tailoring algorithm. Then, we compared the representativeness, impact on performance metrics, and qualitative differences, as asked by the research questions. In the following, we describe the method and results. We will discuss the results concerning the research questions in Section 13.3.

13.2.1. Experimental Method

In the following, we describe the method of this study, including the system under test (SUT), experiment setup, and experiment process.

13.2.1.1. System Under Test

In our experiment series, we used the Sock Shop Microservices Demo mentioned earlier (Section 7.1) as SUT, on which we executed the generated load tests. The Sock Shop serves functionalities such as browsing, shopping cart management, purchasing, and user management via the following microservices (with databases for some): *front-end* (14 endpoints), *catalogue* (4 endpoints), *carts* (4 endpoints), *orders* (2 endpoints), *payment* (1 endpoint), *shipping* (1 endpoint), and *user* (6 endpoints). In addition to the Sock Shop itself, we utilized the open-source monitoring systems Zipkin (Zipkin, 2020) for trace collection, a Java service converting the Zipkin traces into OPEN.xtrace (Okanović et al., 2016) to allow for monitoring tool independence, and Prometheus (Prometheus, 2020) for collecting performance metrics.

13.2.1.2. Experiment Setup

As illustrated in Figure 13.1, we deployed the Sock Shop on a bare-metal machine with 80 cores (2 threads each) at 2300 MHz, 896 GiB RAM, and a

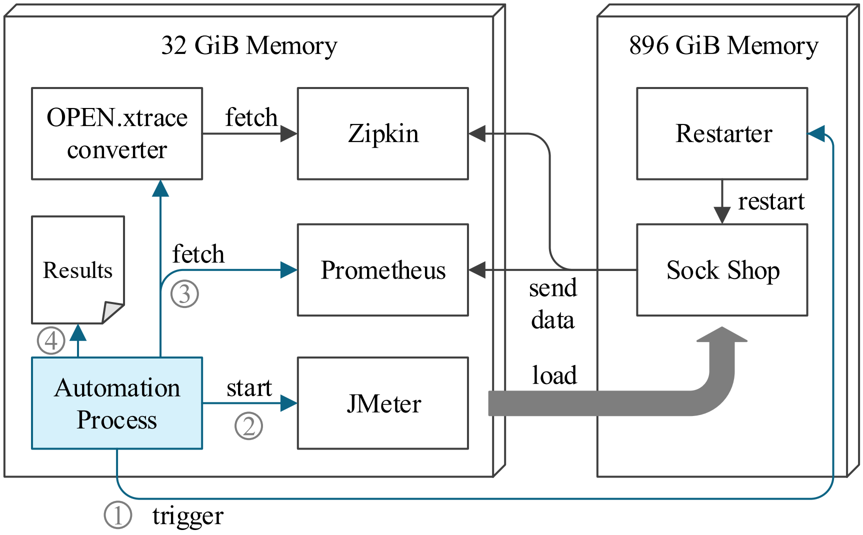


Figure 13.1.: Experiment setup for load test execution.

magnetic disk with 15 000 rpm. We deployed each microservice as a Docker container with two isolated CPU cores and 4 GiB of RAM. Besides, the machine hosted a lightweight Java service for restarting the application remotely. Zipkin, the OPEN.xtrace converter, and Prometheus were deployed on a second machine with 24 cores (2 threads each) at 2300 MHz and 32 GiB RAM, connected via a shared 10 Gbit/s network infrastructure. The second machine also hosted the JMeter (Apache Software Foundation, 2020[a]) load tests we executed and a script automating the experiment (see below). JMeter had a heap size of 512 MiB, except for the untailored test, which needed 2 GiB.

13.2.1.3. Experiment Process

Our evaluation consists of an experiment series in three steps, which we describe in the following.

Simulate Production Workload: As representative load testing uses production monitoring data for generating load tests, we needed to simulate the production workload first. For that, we designed a load test mimicking three different types of users, namely users that visit products (80 users), browse and buy products (60 users), and visit the status of their orders (60 users). An experiment automation script executed this load test, as depicted in Figure 13.1. First, it restarted the Sock Shop to ensure that the load test was executed in a clean and comparable environment ①. Then, it executed the load test ②. During the load test, the Sock Shop sent traces to Zipkin and CPU and memory metrics to Prometheus. After the load test had finished, the metrics and traces — via the OPEN.xtrace converter, which retrieved the Zipkin traces — were collected ③ and stored into a results folder for later analysis ④. In the following, we refer to the workload (model) of this load test as the *reference workload (model)*.

Generate Load Tests: After the execution of the reference workload, we used the collected traces to generate tailored load tests by using the log-based and model-based approaches. As baselines, we generated an untailored (system-level) load test using the plain WESSBAS (Vögele et al., 2018) and request-based load tests, which simply replayed all requests at a rate extracted from the reference workload. We applied the request-, log-, and model-based approaches for generating load tests for the following combinations of microservices, considering the dependencies shown in Section 7.1:

- catalogue — 4 endpoints
- user — 6 endpoints
- carts, orders, payment, shipping, user — 10 endpoints
- catalogue, user — 10 endpoints
- catalogue, carts, orders, payment, shipping, user — 14 endpoints
- catalogue, carts, user — 14 endpoints
- catalogue, carts, payment, shipping, user — 16 endpoints

Please note that we always included dependent microservices, i.e., for the *orders* service, we also included *carts*, *payment*, *shipping*, and *user*. If *orders* was to be tested in isolation, load-test-ready stubs needed to replace the dependent services, which could, however, influence *orders*' performance. Therefore, we used the actual services as "perfect" stubs.

For parameterizing the generated load tests, we utilized our approach from Chapter 6. For each microservice, we defined one Input Data and Properties Annotation (IDPA), and for each load test, we chose the ones corresponding to the tested services to parameterize the test. For good comparability, we used the same parameterization for all tailoring approaches. Besides, this procedure allowed a high degree of automation of the experiments.

Execute Load Tests: The last step was the execution of the generated load tests. Again, we utilized the setup depicted in Figure 13.1. The experiment automation took care of executing all tests for 30 minutes and restarting the Sock Shop before each test to yield comparable results.

13.2.2. Results

In this section, we present the results of the experiment, separated by the research question. For convenience, we denote the load tests transformed from a workload model by the used tailoring approach, e.g., *log-based load test* or *untailored load test*.

13.2.2.1. Representativeness

For assessing the representativeness of the executed load tests, we calculate the distance metric D (see Section 11.2.5) for each of them. Also, we distinguish between the number of endpoints involved in the load test, because we expect it to influence D. Figure 13.2 shows the resulting values including the convex hull per tailoring approach and a line for the critical value $c(\alpha) = 1.36$ (Massey Jr., 1951) for $\alpha = 0.05$. We can see that mostly all values are higher than $c(\alpha)$, indicating a significant difference. However, for our approaches, D is only slightly higher than for the untailored test but

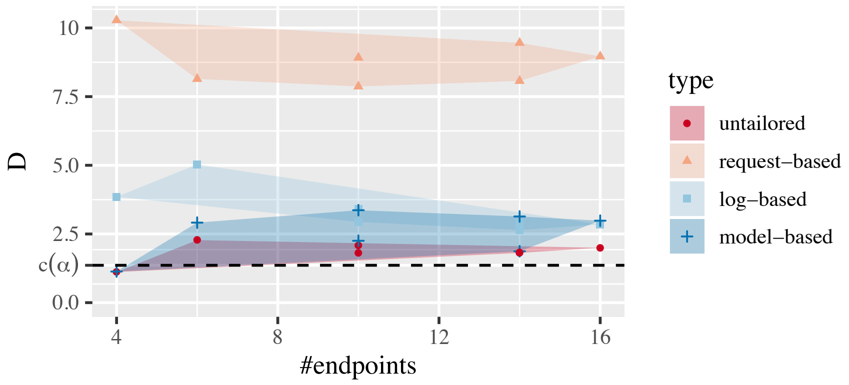
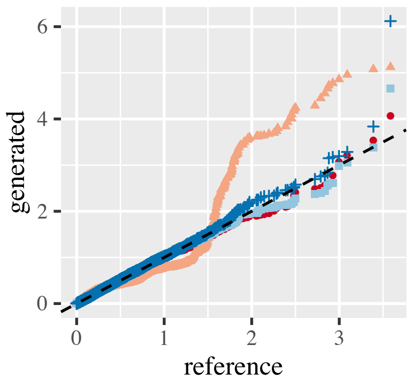


Figure 13.2.: Aggregated Kolmogorov-Smirnov statistic *D*.

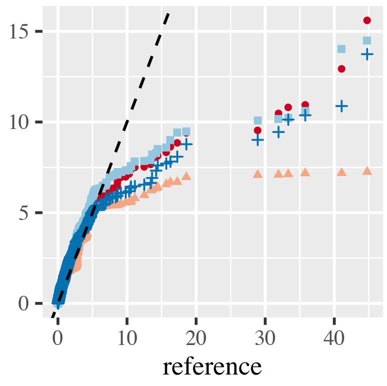
lower than for the request-based tests, which increase *D* by a factor between 3.57 and 9.22 compared to the untailored test.

In general, the model-based load tests generate more representative results — they increase the distance by a factor in the range of 1.02 to 1.86 — than the log-based tests with factors between 1.42 and 3.46. However, the log-based approach becomes better with an increased number of endpoints, while the model-based approach has a slight but ambiguous upward trend. When testing 16 endpoints, the difference between the two approaches is negligible.

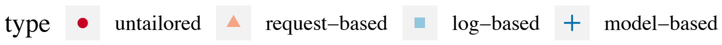
For investigating the difference of the inter-arrival time distributions in more detail, we use quantile-quantile (Q-Q) plots (Figure 13.3). For shorter inter-arrival times, e.g., in case of the *GET /catalogue/size* endpoint when testing the *catalogue* microservice individually (4 endpoints), it can be seen that the request-based test has a different distribution than the reference test. The other generated tests are close to the reference test. For longer inter-arrival times such as the *POST /orders* endpoint measured when testing the *orders* microservice in combination with the dependent ones, the tail of all distributions is different from the one of the reference test. However, the tails of the untailored, log-based, and model-based tests appear similar.



(a) *catalogue GET /catalogue/size*.



(b) *orders POST /orders*.



(c) Legend.

Figure 13.3.: Q-Q plots of the inter-arrival times compared to the reference workload for different endpoints.

13.2.2.2. Performance Metrics

For assessing the influence of potentially less representative load tests, we compare performance metrics. Precisely, we analyze the response times of the requests, the CPU utilization, and the memory consumption during each load test. We extract the response times from the collected traces. Prometheus collected the CPU and memory metrics in 5-second granularity.

Table 13.2 provides a summary of t-tests and Cohen's d applied to the response times, CPU utilization, and the change of used memory per second. Each t-test compares the measurements of a tailored test with the untailored one with $H_0 : F_X(x) = F_0(x)$ and $H_A : F_X(x) \neq F_0(x)$. The numbers count the occurrence of a particular combination of a significant difference, detected by the t-test, and an effect size. For all metrics, there are cases where H_0 is rejected and others in which no significant difference is detected. The most differences — including larger effect sizes — are detected in the CPU

Table 13.2.: Summary of the Statistical Tests

t-test **Cohen's d**	**not sign.** **negligible**	**sign.** **negligible**	**sign.** **small**	**sign.** **medium**	**sign.** **large**
		response time			
request-based	19	31	24		
log-based	28	20	26		
model-based	29	23	22		
		CPU utilization			
request-based	7		7	7	28
log-based	12		6	9	22
model-based	9		8	7	25
		memory change per second			
request-based	38	3	8		
log-based	33	10	6		
model-based	36	8	5		

utilization, which we attribute to the naturally high fluctuation of this metric. For the response times and the memory, the most frequent effect size is negligible. Between the three tailoring approaches, there is no apparent difference, even though for the response times, there are more cases with a significant difference for the request-based test.

13.2.2.3. Required Test Duration

Figure 13.4 shows the test duration required until the median response time reaches its final value except for an error of 1 % (see Section 11.2.7). In relation, it shows the number of tested endpoints. We use the untailored test as a baseline and include the convex hull for illustrating the overall relation. We calculate the duration based on the response times of the tested endpoints, i.e., for the untailored test, we consider the endpoints of the *front-end* service, while we use the endpoints of the respective microservices for the tailored tests. We can see in the figure that the untailored test requires the

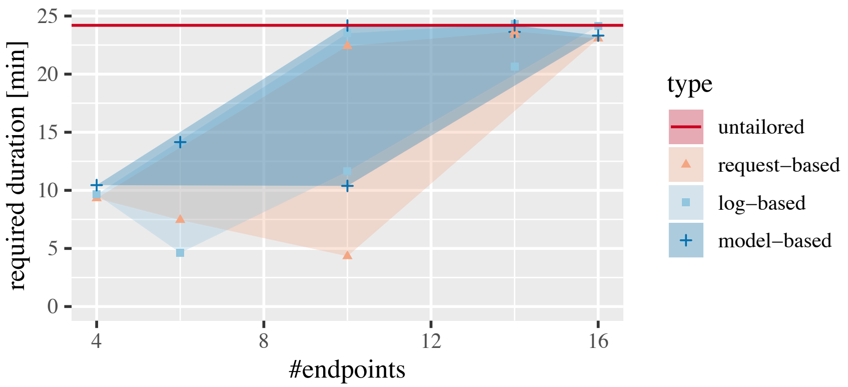


Figure 13.4.: Required test duration per number of tested endpoints and test type.

longest execution time with 24.2 minutes, except for the log-based test for one service combination (*catalogue, carts, orders, payment, shipping, user*), which needs 24.33 minutes. In general, fewer endpoints under test reduce the required test duration. The request-based test could reach the shortest duration of 4.33 minutes for the *catalogue* and *user* service (10 endpoints), followed by the log-based test for the *user* service (6 endpoints) with 4.62 minutes. Furthermore, among the tailoring approaches, the request-based approach tends to require the shortest duration, while the model-based approach requires the longest duration. However, the difference becomes small for an increasing amount of endpoints.

13.2.2.4. Qualitative Differences

Analyzing the behavior models that have been generated during the experiment series, we identify three significant differences between the log-based and the model-based tailoring approaches. These differences support the insights from Section 13.1.3. First, the number of Markov chains (behavior models) is different: while the model-based algorithm reuses the Markov chains generated by the untailored approach — which are five in our case —, the log-based approach generated five chains in most cases, but only three for the *users* microservice tested in isolation. Hence, the clustering of the session logs is different. In general, even though the number of Markov chains is equal for most of the tests, we cannot assume that the clustering is equal.

The second difference is the number of states of each Markov chain. The log-based approach always has as many states as the number of endpoints involved in the test. In contrast to that, the model-based approach can have more endpoints. For instance, the *user GET /customers/id* endpoint is used in the replacements of the *GET /customers/id* and *POST /orders* endpoints of the *front-end* microservice. Hence, it occurs two times in the tailored Markov chain. In our experiments, the chains of the model-based tests have between 25 % and 50 % (34 % on average) more states than the log-based approach.

Finally, the states of the model-based tailored Markov chains correspond to states of the untailored chains. By tracking the origin of each state — e.g., we keep the original state's name in the name of the new state —, this relationship allows for analyzing and changing the Markov chain based on end-user behavior, e.g., removing a particular end-user request due to a changed API. In contrast to that, the session logs tailored by the log-based algorithm are newly created by clustering, resulting in entirely different Markov chains.

13.3. Discussion of Research Questions

In the following, we discuss the research questions defined in Section 5.1. We refer to our service-tailoring approach described in Chapter 7 and the corresponding evaluation presented in this chapter.

13.3.1. RQ2.1 — Extraction Process Extension

How can we extend the load test extraction process for generating service-tailored load tests?

We identified two algorithms that extend the load test extraction process for tailoring the load test to a set of services while preserving the session-based structure. The typical load test extraction process generates several (intermediate) artifacts, which we investigated for tailoring. We identified log-based and model-based tailoring. Both algorithms utilize recorded traces for determining the call relations between multiple services.

Log-based tailoring modifies the monitored request logs, which contain the end-users' requests. For each request in the logs, it identifies the ones the user-faced services (transitively) make to the target services and uses these as a replacement. The remainder of the extraction process remains unchanged, using the modified request logs.

Model-based tailoring uses the original request logs but changes the workload model, which is generated based on the session logs. Here, we presume a Markov-chain-based workload model, as generated by the WESSBAS approach (Vögele et al., 2018). The algorithm replaces each state with a sub-Markov chain representing the aggregated request behavior caused by the end-user request.

We proved that both algorithms are correct according to several requirements, but also differ in the finally generated load tests. Hence, we cannot sustainably conclude their suitability compared to each other. For this reason, we conducted an experimental study, which we use to discuss the research questions below.

13.3.2. RQ2.2 — Representativeness

How representative are the workloads generated by the service-tailored load tests compared to an untailored and a request-based test?

According to the experimental study, our introduced tailoring algorithms slightly reduce the representativeness compared to an untailored load test but significantly increase it compared to a simple request-based load test. The log-based load tests are less representative than the model-based tests but become increasingly representative with more tested endpoints.

We identified the reduction of the representativeness in the aggregated Kolmogorov-Smirnov statistic D, which was, however, only slightly higher than for the untailored test. In contrast to that, the request-based approach was more than two times less representative. For the model-based algorithm, we attribute the reduction to the calculation of the think times. As previously described, when merging state transitions, we need to convolve the think time distributions into another normal distribution (see Section 7.5.3). However, the resulting distribution does not accurately represent the original think time distribution. While the mean remains correct, the variance and distribution function is impaired. Hence, the think time between two requests can be different than in the untailored load test, which influences the inter-arrival times.

In the log-based approach, all applied operations are valid. However, as described in Section 13.1.3, the clustering might classify the tailored request or session logs differently than the original ones, resulting in different Markov chains. The fact that the log-based load test for the *user* microservice has only three chains while the untailored test has five supports this reasoning. For a higher number of tested endpoints, the difference in the Markov chains appears to be less critical.

13.3.3. RQ2.3 — Performance Metrics

To which degree do the service-tailored load tests impair the performance metrics of the tested services?

The performance metrics support the findings of RQ2.2: there are small differences between the tailored load tests and the untailored one. However, these metrics do not reflect the more substantial differences to the request-based tests.

We identified the differences using t-tests and Cohen's d. A possible explanation for the unexpectedly small differences between the request-based load tests and the ones tailored with our algorithms is that the average number of requests per endpoint and second is the same. As a result, the Sock Shop can behave similarly. Regardless of that, a small difference in performance metrics does not implicate a small difference in the representativeness. In contrast, the workload distance discussed in the previous section indicates a higher difference than the performance metrics do.

13.3.4. RQ2.4 — Required Test Duration

How much can service-tailoring reduce the test execution time until measured performance metrics are stable?

A higher number of tested endpoints requires a longer test duration. For a small number of endpoints, the request- and log-based approaches require the shortest durations, but this is at the expense of representativeness, especially for the request-based approach.

We can derive this finding from the calculated test duration metrics, but also need to take into account that there can be individual endpoints that require a longer duration. For instance, the *POST /orders* endpoint of the *orders* service required between 22.42 and 24.33 minutes, while the *GET /customers/* endpoint of the *user* service required only between 8 seconds and 4.97 minutes. This finding indicates that the longer duration is not only due to the sheer number of endpoints but also to the higher likelihood to include an endpoint requiring a longer duration.

Besides, we would like to highlight that the tailoring approaches can only reduce the test duration from the perspective of all tested microservices. If only a particular set of microservices, e.g., *user,* is considered to determine the test duration also in the untailored test, the tailoring approaches cannot reduce it without impairing the representativeness, as it would require the workload arriving at *user* to be changed. In contrast to that, the tailoring approaches can always save resources because only the tested services, including its dependents, need to be deployed. By combining them with stubbing approaches (Baltas and Field, 2012; Becker et al., 2008; Field et al., 2018; Versteeg et al., 2016), the dependent microservices can be removed from the deployment as well.

Concluding, the fewer services are included in a load test, the more resources can be saved by only deploying the tested services and potentially stopping the test after a shorter time. For that, we suggest using existing approaches (Alghamdi et al., 2016).

13.3.5. RQ2.5 — Qualitative Differences

Which qualitative differences of the service-tailored workload models exist?

As we identified in the formal comparison and the experimental study, the model-based tailoring approach generates Markov chains with states that correlate with end-user requests and, thus, can better explain the effect of the end user's behavior. In contrast to that, the log-based approach does not allow for such analyses. As a drawback, the model-based approach generates Markov chains with 36 % more states on average.

Even though the sizes of the Markov chains in our experiment are not critical, it can become memory-critical for large-scale applications. Furthermore, duplicated states hinder manual maintainability. However, as the Markov chains are generated automatically and not meant to be changed manually in the first place, this drawback is less relevant. Therefore, our results indicate that the model-based approach is preferable over the log-based approach regarding qualitative attributes.

13.4. Threats to Validity

We identify the following threats to the validity of our work.

13.4.1. Conclusion Validity

In the analysis of the performance metrics, we applied the t-test for detecting significant differences between the tailored and untailored tests. These metrics are not necessarily normally distributed, which is an assumption of the t-test. However, as the sample sizes are large with at least 305 entries, the t-test can be used according to the central limit theorem.

Furthermore, performance metrics such as CPU utilization can be fluctuating and, thus, unreliable in general. Therefore, the conclusions we have drawn are based on a combination of several performance metrics and also workload metrics.

13.4.2. Internal Validity

After the study, we detected a bug in Algorithm 7.3 (REMOVESTATE). This bug affects the think time of model-tailored Markov chains when states with loop transitions are removed. Precisely, we missed adding the time spent in the state while looping. Hence, the think time can be too short. However, we assessed the model-based tailoring to be more representative than the other tailoring approaches. Therefore, the fixed version could only reinforce our findings. For future work, we strongly recommend using the fixed version of the algorithm presented in this dissertation.

Our evaluation consists of a series of experiments, which we executed automatically in a sequence. For preventing influences of former test executions, we restarted the tested Sock Shop application and the JMeter load driver have before each execution, including a completely new deployment of the Sock Shop. For preventing interactions between microservices running concurrently, we applied CPU pinning and memory reservation.

Another potential threat is that the metrics measured during the reference workload appeared to bear small inconsistencies. This manifested with the steady increase of the response times of the *carts* microservice. During all generated load tests, we could not observe such an effect. Hence, the comparison to the reference test is impaired. However, as the input, i.e., the request logs, for all workload model generation approaches was the same, we presume no side effects regarding the comparability among these tests. Also, we could not detect any trends in the request rates of the reference workload, justifying the steady-state execution of the generated load tests.

13.4.3. Construct Validity

In this paper, we assume normally distributed think times in the Markov chains modeling the workload. In general, think times do not need to be normally distributed, and workload modeling languages such as the

WESSBAS-DSL allow for different distribution functions as well. We chose the normal distribution because it is commonly used in related work. Evaluating other distribution functions, we leave for future work.

13.4.4. External Validity

As the Sock Shop is not an industrial application, it is questionable whether it can represent real-world microservice applications. However, we base on an existing study, which assessed the Sock Shop to be representative (Aderaldo et al., 2017). For future work, we suggest evaluating the tailoring algorithms with another industrial application.

13.5. Summary

In this chapter, we presented the evaluation of our service-tailoring algorithms. We proved that both of them fulfill the postconditions we defined and conducted an experimental study. Our approach allows DevOps teams to tailor load tests to their developed services with little representativeness reduction.

In the next chapter, we evaluate our approach to context-tailored load test generation. In combination, both tailoring approaches allow generating load tests targeting the services of interest and executing the most relevant workload.

Chapter 14

Evaluating Context-tailored Load Testing

This chapter provides the evaluation of our approach to context-tailored load test generation (Chapter 8). Based on a tailoring description specified in the Load Test Context-tailoring Language (LCtL), it extracts a load test from an incrementally learned workload knowledge base. In doing so, we utilize time series forecasting approaches for predicting future workload scenarios. For easing the load test description and improving the forecast, users can define workload contexts. Overall, the evaluation addresses the sub-questions of RQ3: *How can representative load tests automatically be tailored to the contexts of a session-based workload?*

We evaluated our approach using the publicly available requests of the student information system (SIS) of Charles University, Prague of half a year (Maňásek and Tůma, 2019). We learned a workload model from the requests and investigated four different aspects. First, we analyzed the evolution of the workload model when being updated incrementally. Then, we investigated to which degree the LCtL can express relevant workload scenarios we identified. For assessing the general predictive power of the workload model, we generated and executed several load tests using the perfect forecasting approach. Compared to the original requests, the requests these load tests submit indicate how representative the workload model is, without impairment by potentially inaccurate forecasts. Finally, we generated and executed load tests using forecasting tools. A replication package and supplementary material are available online (H. Schulz et al., 2020b).

While there was some fluctuation between the versions, incremental learning did not change the behavior models significantly. However, we observed fluctuating session durations, which motivate the need for incremental learning that better integrates the think times. The LCtL is suitably expressive for the real-world scenarios and contexts of the SIS. Also, our approach is suited for generating representative context-tailored load tests. Predicted workload scenarios are particularly representative when considering the user groups separately and enriching the forecasts with contexts. However, the calculations of these forecasts are also long-lasting. Future work needs to address several limitations of the existing workload modeling and forecasting approaches, which we identified. These include the session duration fluctuations, predictions of sharp spikes, and forecasting durations.

This chapter is structured as follows. In Section 14.1, we introduce the data set this evaluation bases on and describe its preparation. In Section 14.2, we present the analysis of the workload model. Section 14.3 presents the evaluation of the expressiveness of the LCtL. Sections 14.4 and 14.5 comprise the experimental studies. In Section 14.6, we discuss the results of all studies concerning the research questions. We discuss threats to the validity of our work in Section 14.7. Finally, we summarize this evaluation in Section 14.8.

The chapter is an extended version of Section 5 of the manuscript below. We have added the workload model analysis in Section 14.2, the investigation of workload phases as part of the expressiveness evaluation in Section 14.3, and a forecasting duration analysis in Section 14.5.

- H. Schulz, D. Okanović, A. van Hoorn, and P. Tůma (2021). "Context-tailored Workload Model Generation for Continuous Representative Load Testing." In: *Proceedings of the 12th ACM/SPEC International Conference on Performance Engineering (ICPE 2021)*. To appear. ACM

14.1. Preparation

This section describes the preparation we applied for the evaluation. We introduce the dataset we base on in the following, the incremental learning of the workload model, and the enrichment of the workload model with contexts.

14.1.1. SIS Dataset

We utilized a publicly available dataset comprising the Web server request logs of the SIS of Charles University, Prague (Maňásek and Tůma, 2019). Most of the users of the SIS are students and scientific staff, who manage their daily routine with the system. Among other activities, students can apply for courses and exams, check exam grades, and access course material. Scientific staff can schedule courses and exams, communicate with the students, and publish course materials and exam results.

The dataset contains the anonymized HTTP request logs the users of the SIS submitted. Relevant for our research, each request holds the client IP address, a timestamp, the HTTP request method, and the requested path. Hence, we can derive endpoints from the request method and path and identify the user sessions based on the client IP addresses. Overall, the dataset contains about 5.7 million user sessions from May 23 to November 22, 2018. In this time frame, the users submitted about 20.7 million requests to 98 different endpoints — excluding requests to static resources such as images or Cascading Style Sheets (CSS).

14.1.2. Workload Model Learning

For evaluating our context-tailoring approach, we needed to transform the request logs into a Makov-chain-based workload model. For that, we mapped the requests to endpoints by using an Input Data and Properties Annotation (IDPA) (see Section 6.5.2) and grouped them into sessions by using the client IP addresses as session IDs. Furthermore, we split the sessions after 30 minutes of inactivity. Then, we applied our approach

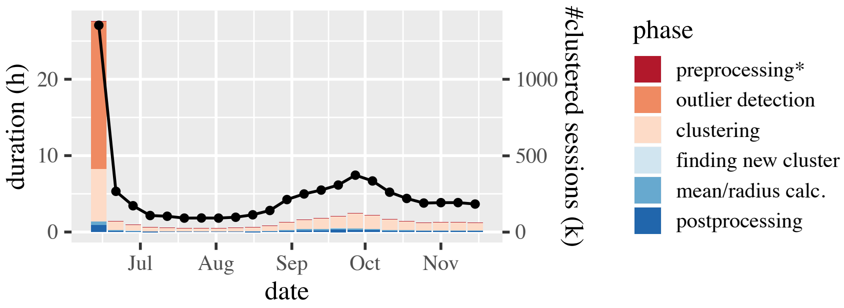
Table 14.1.: Parameters for the Clustering of the SIS Dataset

Parameter	**Value**
initial clusters (k)	20
repetitions (η)	30
quantile range (α)	0.95
radius tolerance factor (β)	1.1
min. sessions per new cluster (m)	500

to the incremental session clustering (see Section 8.4.1). We utilized the session clustering based on k-means with the parameters shown in Table 14.1. Besides, the initial clustering with k-means++ processed the sessions of four weeks, while further iterations processed one week each. We have identified the parameters by starting with an educated guess and trying to improve the resulting clustering. The parameters presented led to the best result regarding separated and tight clusters, low noise, and a reasonable number of new clusters in later clustering iterations. Furthermore, we calculated the intensities per cluster with a granularity of one minute.

We obtained 20+1 initial behavior models — each representing one cluster or user group and one for the noise — with an updated state every week and the varying intensities per group. Over time, five further clusters appeared. In Figure 14.2a, we show the intensities. The figure shows daily and weekly seasonalities and also several phases with a uniform workload. Besides, we can identify intensity spikes.

We executed the clustering on a bare-metal machine with 32 GiB RAM and an Intel® Xeon® CPU E5620 with 2.40 GHz clock frequency, 4 cores, and 8 threads. Figure 14.1 shows the duration and the number of sessions clustered in each iteration. The initial clustering with k-means++ took the longest time (27.6 hours) but also processed the most sessions. It spent the most time on detecting and removing the outliers, followed by the actual clustering. The further iterations, which assigned sessions based on the minimum distance, took between 0.5 and 2.5 hours, whereas they spent most of the time on the actual assignment. Remarkably, all clustering iterations lasted substantially



*We accelerated the preprocessing by using sessions buffered in local files.

Figure 14.1.: Duration of the clustering of the SIS sessions per phase.

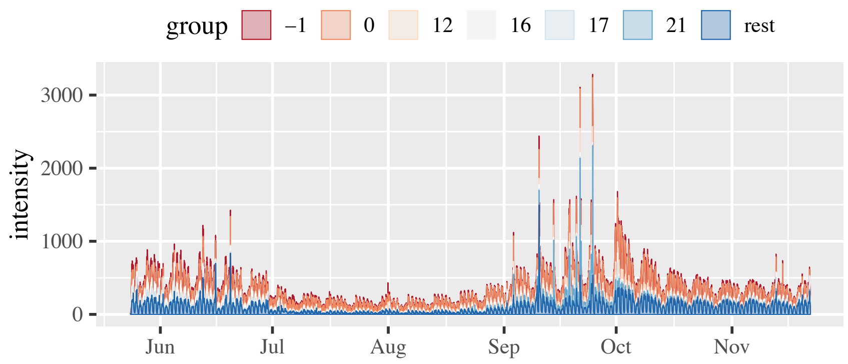
less than the time range of sessions clustered (four weeks and one week for initial and further iterations, respectively).

14.1.3. Context Enrichment

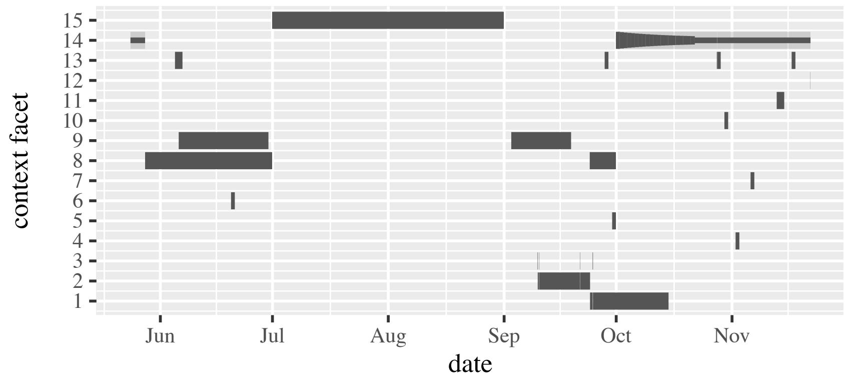
For explaining the different phases visible in Figure 14.2a and for creating a basis for the LCtL, we enriched the intensities with contexts. Defining the relevant context facets, Charles University publishes the semester calendar (Charles University, Faculty of Mathematics and Physics, 2017, 2018). Also, we considered public holidays. Each of the calendar entries we transformed into the state of a context facet, as illustrated in Figure 14.2b. The precise facets and their encoding are as follows.

vacation: We modeled the semester vacation as a boolean facet.

tuition: As it also can be seen in the intensity plot, tuition phases typically have a higher workload in the beginning than later. Accounting for that, we modeled the corresponding facet numerically. The higher the facet's value, the higher the workload. As a base value indicating the presence of tuition, we used 1. At the beginning of each tuition phase, we set the value to 10, decreasing negative exponentially to 1 within three weeks.



(a) Stacked intensities (number of concurrent sessions) per group.



context facets:

1. course_enrolment.open
2. course_enrolment.priority
3. course_enrolment.special
4. deadline.bachelor_topic_decision
5. deadline.higher_year_programmes_registration
6. deadline.thesis_submission
7. deans_sports_day
8. examination
9. final_exam
10. first_year_matriculation
11. graduation_ceremony
12. open_day
13. public_holiday
14. tuition (line width correlates with facet value)
15. vacation

(b) Context.

Figure 14.2.: Illustration of the per-group intensities and context of the SIS dataset.

examination: The examination phase we modeled as a boolean facet. Opposed to the calendar, we did not explicitly separate between Bachelor's and Master's examination phases, as we expect them to influence the workload similarly.

final_exam: We modeled the final exams as another boolean facet.

deadline: Deadlines we modeled as a string facet with the type of deadline as the state. The states comprise the thesis submission, registration for higher year programs, and the decision on the Bachelor's topic.

course_enrolment: We modeled the enrolment of the courses as a string facet. Corresponding to the calendar, we distinguished between the priority and open mode enrolment. Furthermore, we labeled strong spikes, which occurred during the course enrolment, as having the *special* state.

events: University events such as an open day, graduation ceremony, dean's sports day, or the first year matriculation we modeled as separate boolean facets.

public_holiday: Public holidays we modeled as a boolean facet without distinguishing the type of holiday.

Remarkably, the context well explains the different workload phases. During the initial exam phase, there are irregular spikes, which we attribute to differently scheduled exams. During the vacation, the workload is significantly lower but increases again when the next examination phase starts. The course enrolments cause high spikes, as the students try to register for their favorite courses. The tuition phase has a relatively stable workload, which is high in the beginning and normalizes within the first few weeks.

Some of the facets appeared not to influence the workload. Therefore, we only considered the following facets in the evaluation: *tuition*, *examination*, *final_exam*, *vacation*, *course_enrolment*, and *graduation_ceremony*.

14.1.4. Intensity Augmentation

A drawback of the dataset is that it only contains the workload of half a year, i.e., the length of one semester. As the figure shows, the different phases mostly occur only once, representing the half-yearly seasonality. However, forecasting tools require multiple seasons for properly predicting the future. Therefore, for the fourth study, we augmented the per-group intensities by another half year. As precisely documented in the supplementary material (H. Schulz et al., 2020b), we constructed the augmentation as follows:

1. We defined the context of the augmented time range, as described in Section 14.1.3. Then, we re-assembled the per-group intensities according to the context. As an example, we used the first tuition weeks from October 2018 for the first tuition weeks in February 2019.
2. We calculated the resulting total per-minute intensities and the relative per-minute frequencies per group. In the following, we modified the total intensities.
3. We applied locally estimated scatterplot smoothing (LOESS) (W. S. Cleveland et al., 1991) to the intensities.
4. We multiplied the intensities of each day with a factor randomly selected according to the variance of the original days.
5. We jittered the intensities with gaussian randomness based on the difference of each original intensity value to its predecessor. Hence, we obtained intensities with the same variance, but with locally different values.
6. The previous steps reduced the intensity of the highest spikes and lead to a few negative values. Therefore, we re-increased the highest spikes and removed the negative values.
7. We calculated the per-group intensities by multiplying the per-group frequencies with the modified total intensities.

14.2. Analysis of Incrementally Learned Workload Model

This study addresses RQ3.2: *How much does the incremental learning affect the workload models?* We analyzed the incrementally learned and such evolved versions of the workload model. We considered the distances, the expected number of steps and duration of a session emulating a specific behavior model, and the expected request rates. In the following, we describe the analysis method and present the results.

14.2.1. Analysis Method

We analyzed the workload model formally, i.e., by calculation rather than simulation. For that, we utilized the behavior models of the workload model versions, which each have one Markov chain describing the request sequences users of the respective group submit and a think time distribution per Markov transition (see Section 3.2.4.2 for details). We analyzed the distances between the behavior models, expected session length (number of requests), expected session duration (time between first and last request), and expected requests per endpoint and time. Furthermore, for putting these measures into context, we analyzed the number of sessions that have been aggregated into each behavior model. In the following, we describe how we calculated the measures.

14.2.1.1. Distance between Behavior Models

We determined the distances between all versions of all behavior models by principal component analysis (PCA) (Pearson F.R.S, 1901). For that, we used the average number of transitions users have taken per behavior model as principal components. These 5729 components correspond to the session representation used for the clustering and form the basis for the Markov chain representation, whereas each Markov transition corresponds to one (normalized) component. The PCA attempted to reduce the components to two for plotting, preserving the distances between the behavior models.

14.2.1.2. Requests per Endpoint and Session

As the Markov chains are absorbing by construction, we could use existing formulas (Grinstead and Snell, 2012) for calculating the expected number of requests per endpoint and sessions. Given a matrix representation M of a Markov chain and the fact that a load test always starts simulating a Markov chain with the initial state, we calculated the expected number of requests as below. First, we obtained the fundamental matrix N (Grinstead and Snell, 2012) of M. Then, the first row of N described the expected number of requests submitted to the corresponding endpoint:

$$\mathbf{r} = (1, 0, \dots, 0) \cdot N^T$$

14.2.1.3. Session Length

We calculated the expected session length, i.e., the number of requests per session, based on the values calculated in the previous section. Given the vector $\mathbf{r}$ of requests per endpoint and session, the session length is the following.

$$\mathbf{r} \cdot (1, \dots, 1)$$

Furthermore, we calculated the session length variance based on the fundamental matrix N and using the Hadarmard (elementwise) product $\circ$.

$$(2N - \mathrm{diag}(\mathbb{1})) \cdot \mathbf{r} - \mathbf{r} \circ \mathbf{r}, \qquad \mathbb{1} = (1, \dots, 1)^T$$

14.2.1.4. Session Duration

We calculated the session duration by merging the expected number of requests with the think time distributions. Precisely, we calculated the convolution (Montgomery and Runger, 2003) of all think time distributions weighted with the expected number of times the respective transitions will be taken. Hence, we summed the time a session spends waiting. Remarkably, this calculation does not include the time spent waiting for a response of the system under test (SUT). However, in our test executions, the response times only added a negligible delay of about 1 ms on average. Furthermore, the calculation is an approximation disregarding the variance of the number of requests.

Let M be the matrix representation (excluding the row and column of the final state) of a Markov chain, $\mathbf{r}$ the vector of requests per endpoint and session, Δ^{μ} the matrix holding the think time means in a similar row and column order as N, and Δ^{σ^2} the corresponding matrix of think time variances. Then, we calculated the expected session duration as

$$\mathbb{1}^T \cdot ((\operatorname{diag}(\mathbf{r}) \cdot M) \circ \Delta^{\mu}) \cdot \mathbb{1}, \qquad \mathbb{1} = (1, \ldots, 1)^T$$

and the variance as

$$\mathbb{1}^T \cdot \left((\operatorname{diag}(\mathbf{r}) \cdot M) \circ (\operatorname{diag}(\mathbf{r}) \cdot M) \circ \Delta^{\sigma^2}\right) \cdot \mathbb{1}, \qquad \mathbb{1} = (1, \ldots, 1)^T.$$

14.2.1.5. Requests per Endpoint and Time

Finally, we were interested in the number of requests a user emulating one of the behavior models submits per time and endpoint. We calculated the expected number based on the vector $\mathbf{r}$ of requests per endpoint and session and the expected session duration d as $\frac{1}{d} \cdot \mathbf{r}$.

14.2.2. Results

In this section, we provide the results of our analysis. Due to limited space, we only show selected excerpts from the results. The supplementary material (H. Schulz et al., 2020b) also contains the remaining results.

14.2.2.1. Inter-group Comparison

For getting an overview, we compare selected measures of all versions of all behavior models with each other. Hence, we analyze the inter-group differences. We provide the number of sessions aggregated into each behavior model, investigate the distances between the behavior models using PCA, and compare the average session length and duration.

Number of Sessions: The number of sessions that have been aggregated in each version of a behavior model can help to rate specific changes. Figure 14.3 shows this measure per group over time. As each version of a behavior model also includes the sessions of the previous version, the number of sessions is monotonically increasing. The behavior models of different groups appear to have different numbers of sessions; group 0 is the one with the most sessions, followed by 16 and 12. The ratio is relatively stable over time, indicating a stable clustering. The slope of the top-level (stacked) curve is highest during September. This month entails the course enrolment phase (see Figure 14.2), where the highest workload occurred.

Distance between Behavior Models: Figure 14.4 shows the result of the PCA, which we use for visualizing the distances between the behavior models. The shown axes correspond with the principal components of all versions of all behavior models but are rotated and shifted for better visibility. Furthermore, the plot marks the date — i.e., the version — of the behavior models.

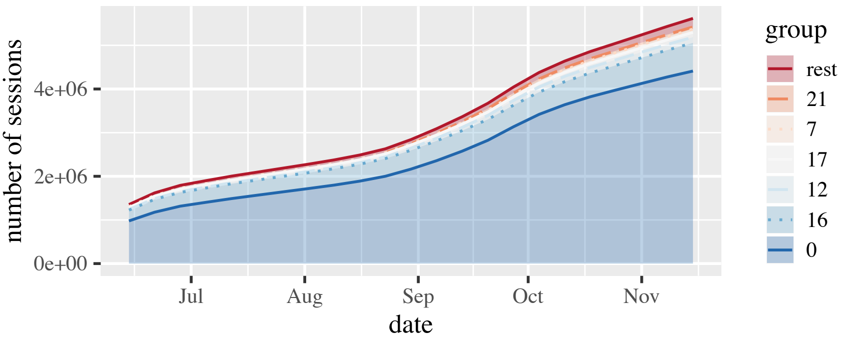


Figure 14.3.: Stacked number of sessions that have been aggregated into each group. The top six groups are shown individually.

Most of the behavior models stay in a small range around the initial version and are separate from the models of different groups. Exceptions to this rule are the noise group (-1), group 9, and some of the groups that appeared after several iterations. For the noise group, fluctuations are natural and acceptable. For the newly discovered groups, the fluctuation might stem from a small initial cluster size, such that newly assigned sessions have a higher impact on the centroid. The same effect can apply to group 9, as it is a relatively small group.

An interesting effect can be observed in groups 1 to 5, 7, 8, 15, and 19, which are arranged on a slightly curved line. Further investigations revealed that these groups have a similar request mix but different session lengths.

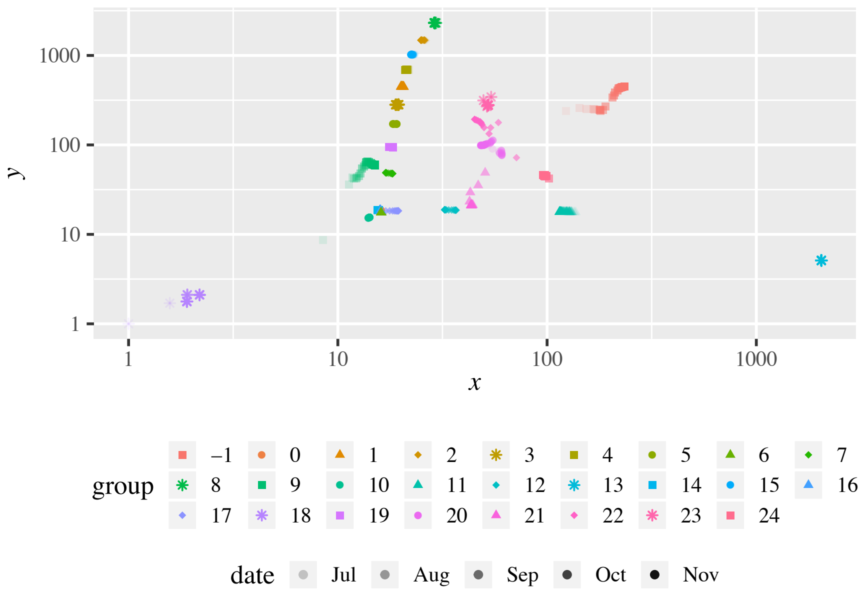


Figure 14.4.: Incrementally updated behavior models per group reduced to two dimensions using PCA. x and y originate from the principal components by rotation by 100.26° and shift to positive values. Please note the logarithmic scale.

Finally, several groups, including 0, 6, 10, 14, 16, and 17, lie in a narrow range and seem to overlap. However, the overlap also can stem from a potential bias of the PCA. Therefore, we investigate further properties of the behavior models. Precisely, we consider the session length and duration.

Session Length and Duration: Figure 14.5 shows the average session length and duration per behavior model version with similar labeling as Figure 14.4. Hence, we can identify the effects of potentially changed behavior models. Here, we can see more fluctuation, but mainly in the dimension of session duration. The session length of most groups remains mostly unchanged. This finding is not surprising, as the incremental clustering assigns sessions by the transition probabilities — which influence the session length — and adjusts the think times — which influence the session duration — afterward.

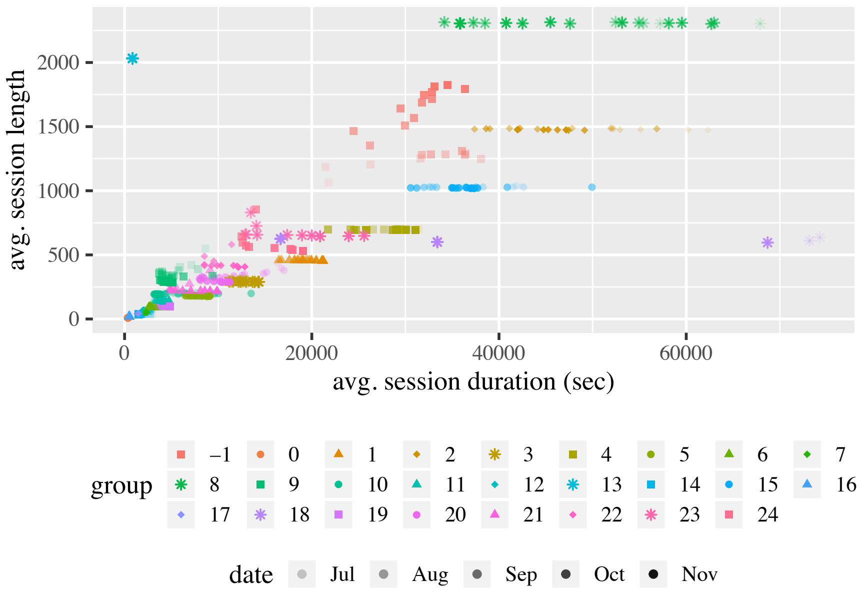


Figure 14.5.: Session length and duration of the incrementally updated behavior models per group.

The groups with more strongly fluctuating session lengths correspond to those with higher variation in the PCA. The groups −1, 9, and 20 to 24 have maximum relative differences of a behavior model version's session length between 22.34 % and 71.43 % compared to the respective initial version. Furthermore, groups 0 and 12 have a maximum difference of 18.55 % and 29.35 %, respectively, but also a relatively small session length. The remaining groups vary by up to 8.68 %. These differences indicate that some models changed over time. However, the impact of the changes remains unclear. Therefore, we provide an analysis of the per-group evolution later.

Finally, Figure 14.5 shows that the session length and duration of the groups having close principal components differ significantly and overlap less, as opposed to Figure 14.4. We conclude that the overlap stems from the PCA rather than an actual overlap of the models. In the next section, we investigate the differences between these models and the discussed fluctuation in more detail.

14.2.2.2. Intra-group Comparison

The inter-group comparison has shown that most behavior models evolved in a narrow range, but some of them also varied. Therefore, we investigate the evolution of the behavior model of each group in more detail. We consider the session length, the session duration, and the number of requests per endpoint. The following per-group analysis focuses on groups 0, 16, 21, and 14, which are the top-, second-, sixth-, and seventh-largest clusters. Furthermore, group 21 is the largest one that has been discovered later. We provide an analysis of the remaining groups in the supplementary material (H. Schulz et al., 2020b).

Session Length: We have analyzed the session length already as part of the inter-group comparison. Here, we analyze it in more detail per group. Figure 14.6 shows the expected average session length plus and minus the standard deviation of the four selected groups. We can see that the session length differs between different groups. Group 0 is the shortest with 8.36

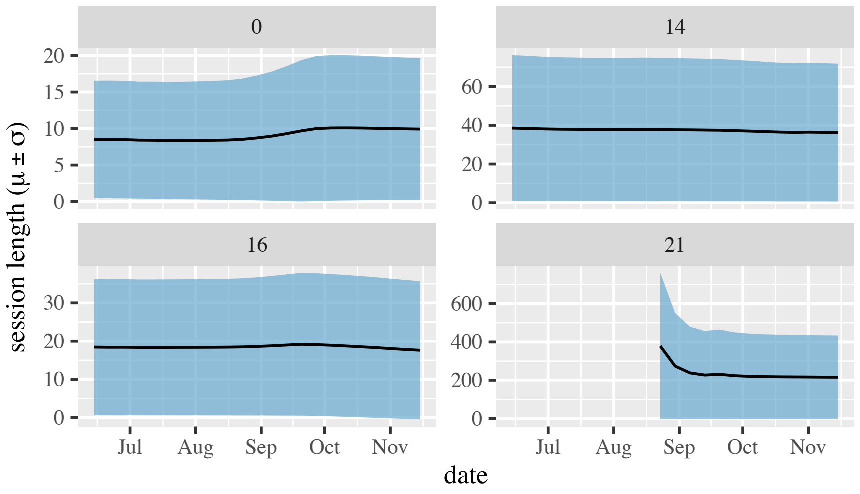


Figure 14.6.: Session length (mean and standard deviation) of selected groups.

requests per session, and group 8 is the longest with 2316.83 requests on average. Furthermore, as previously assessed, the session length of most groups is stable over time.

Group 21 is one example where the session length is less stable. After the discovery of the group at the end of August, the length decreases significantly. A similar effect can be observed for the other groups that occur in a later iteration. Furthermore, the number of requests per session of group 0 and a few others increase during the course enrolment phase. Hence, during this phase, there were sessions different from the ones that were clustered before that were longer — i.e., had more requests — than the previous ones. Before and after that phase, the session length appears to be stable, except for the groups discussed before.

Session Duration: Similar to the session length, we investigate the session duration in more detail. Figure 14.7 shows the session duration of the selected groups in similar plots as Figure 14.6. A first sight reveals that the

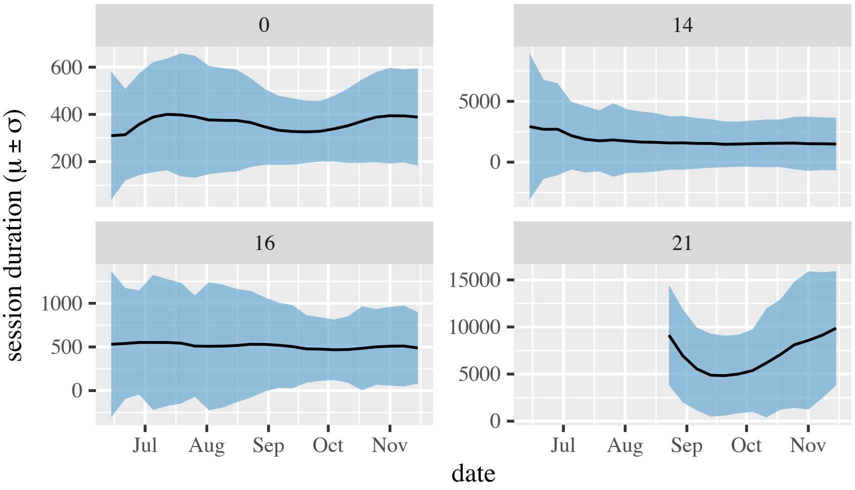


Figure 14.7.: Session duration (mean and standard deviation) of selected groups.

session duration is varying more strongly than the session length. For instance, for group 21, the average duration differs by up to 47.08 % compared to the initial version. For group 10, it varies even by 172.53 %. The standard deviation fluctuates for most of all groups, which we attribute to the incremental clustering, which does not include the think times. Remarkably, for groups 14 and 16 — and some others —, the mean minus the standard deviation is negative. In the next step, we assess the impact of these variations on the number of requests per minute.

Number of Requests per Endpoint: Finally, we investigate the effects of the slight variations of the session length and duration. Both affect the number of requests a single user submits per minute: more requests per session increase the number, while higher durations decrease it. Furthermore, we consider the number of requests per endpoint. Changes to this number indicate a change in the behavior model structure, e.g., that specific paths through the Markov chain become more likely than before.

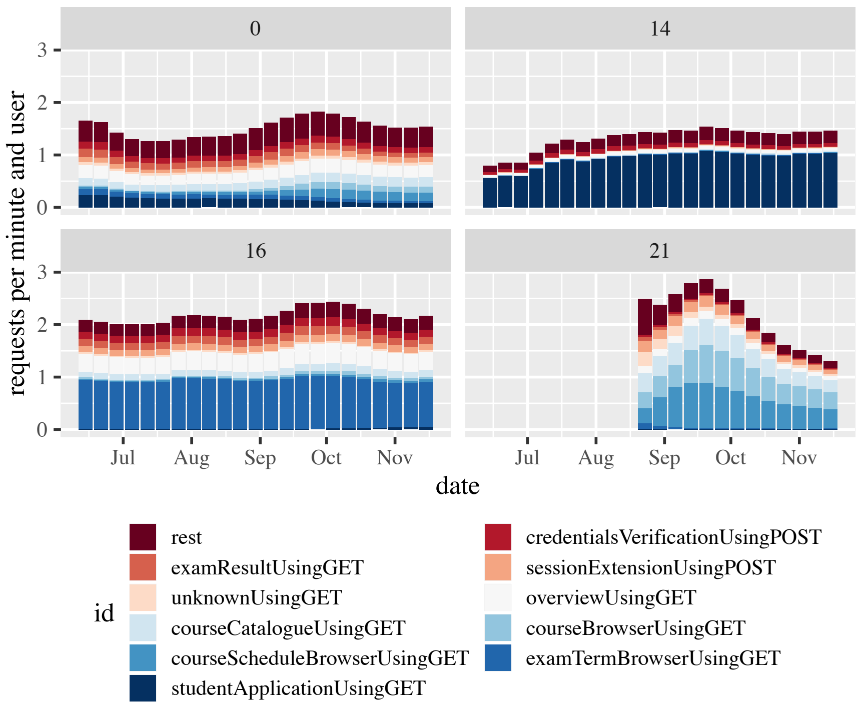


Figure 14.8.: Number of requests per endpoint individual users of selected groups submit per minute. Each bar corresponds to one behavior model version.

Figure 14.8 shows the number of requests per endpoint a single user submits per minute for the same groups as before. A first insight is that most behavior models have visually different request mixes. Furthermore, users belonging to different groups may submit a different number of requests. This finding indicates that the session clustering is valuable and results in significantly different user behavior per group.

As previously discussed, the course enrolment phase also affects the number of requests per endpoint. For several groups, such as 0, 16, and 21, it increases during this phase. Others, such as 14, do not react to the enrolment phase. While for most groups, the ratio of requests per endpoint stays stable, and only the total number of requests changes, the ratio also changes slightly for some groups, such as 21. Concluding, there are sessions during the enrolment phase that differ from the previous ones. Most of them are aggregated in new clusters, such as groups 22 to 24, but some are also assigned to existing clusters. For several groups, they change the session length or duration, while for a few others, they also affect the endpoints each user requests.

14.3. Case Study for Expressiveness Evaluation

This study addresses RQ3.3: *How expressive is the Load Test Context-tailoring Language concerning workload scenarios of a production system?* We analyzed the workload and context of the SIS for identifying use cases for our approach. Then, we aimed at expressing the workload scenarios corresponding to the use cases with the LCtL and assessed the accuracy of the descriptions. The following sections present the details of the case study method and the results.

14.3.1. Case Study Method

We base the evaluation upon the data described in Section 14.1, the publicly available information from Charles University, Prague, and an expert survey. As we aimed to identify relevant use cases and use the LCtL for describing corresponding load tests, we analyzed the learned workload model, workload context, and further potential influences on the workload. We came up with three different questions, which each target different features of the language:

1. Users can utilize the LCtL for describing time frames, from which they want to extract a load test. In doing so, they can rely on collected context facets. We analyzed the facets of the SIS and identified multiple phases, e.g., examination, vacation, and tuition periods, and also combinations of facets. Then, we expressed these phases as an LCtL

`timeframe` section. Hence, we addressed the question, *How precisely and accurately can the LCtL express the workload phases of the SIS?*

2. The second source of information is the workload — i.e., the learned workload model and per-group intensities —, which comprises different workload scenarios, and an expert survey. The survey showed the participants the SIS workload, as presented in Figure 14.2, and asked for relevant load test scenarios. Precisely, we asked about (a) the participant's load testing experience according to the model of skill acquisition by S. E. Dreyfus and H. L. Dreyfus (1980), (b) relevant scenarios without knowing the context, and (c) relevant scenarios when knowing the context. We provide the full survey as part of the supplementary material (H. Schulz et al., 2020b). In order to obtain plausible scenarios, we invited known experts from industry and academia. Then, we defined LCtL instances for all specified scenarios, which a second author of the study (H. Schulz et al., 2021) double-checked. The addressed question is, *How precisely can we describe the relevant scenarios occurring in the SIS workload?*
3. The first two questions mostly target quantitative forecasting. For analyzing further use cases and incorporating qualitative forecasting, we considered the COVID-19 pandemic, which started in spring 2020. Universities, including Charles University, had to interrupt the attendance teaching and switch to (online) remote teaching. These circumstances likely influence the SIS workload. Therefore, we investigated load testing concerns originating from this unusual situation, addressing the question, *How well can the LCtL describe load tests for special circumstances?*

In the next section, we detail the identified use cases for the three questions and present the results of our expressiveness analysis.

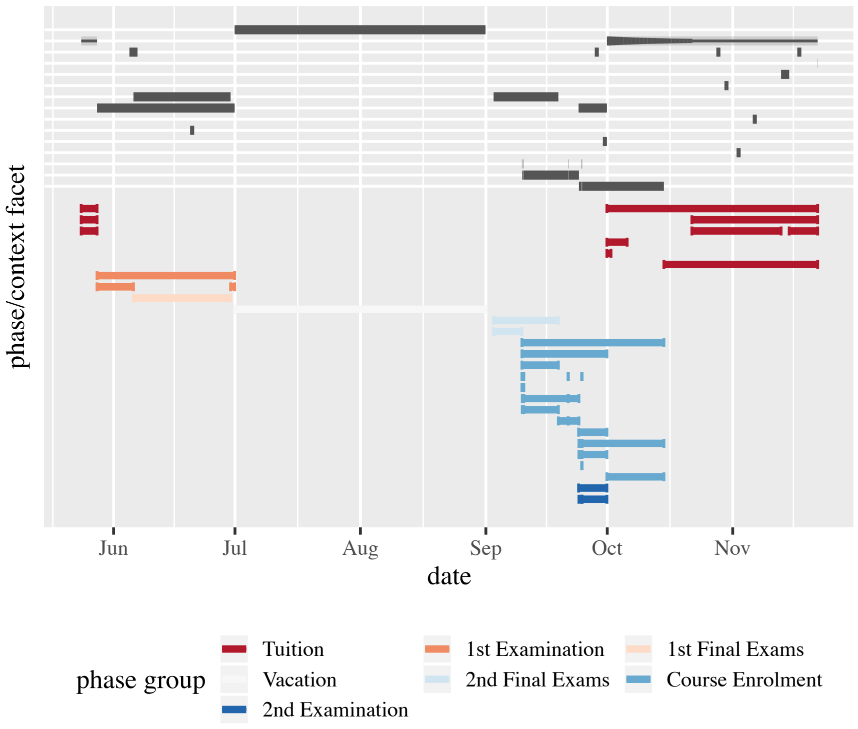


Figure 14.9.: Illustration of the workload phases. The upper part of the figure corresponds to Figure 14.2b.

14.3.2. Results

In the following, we present the results of our expressiveness evaluation, separated by the questions defined above. In Section 14.6, we will discuss the results in order to answer the research questions.

Workload Phases: By analyzing the context facets, we have identified 27 different workload phases. Figure 14.9 provides an overview. We grouped the phases into seven groups, which relate to the particular context facets. For instance, we grouped all phases related to the *tuition* facet into *Tuition*

```
- !<conditional>
  course_enrolment:
    exists: true
```

Listing 14.1: Phase of the *Course Enrolment* group.

```
timeframe:
- !<timerange>
  from: 2018-07-01T00:00:000
  to: 2018-08-15T00:00:000
aggregation: !<percentile>
  p: 95
```

Listing 14.2: LCtL instance describing a specific period.

and the ones related to the *course_enrolment* facet into *Course Enrolment*. For the *examination* and *final_exam* facets, we separated the phases directly after the tuition in May and before the next tuition.

We could express all phases in a `timeframe` section of the LCtL. For the sake of space, we provide the sections only as part of the supplementary material. For all phases, we used a `conditional` clause, whereas we used an additional `timerange` clause in seven cases. The `timerange` clause was required for separating between the first and second *Examination* and *Final Exams* phases. Besides, we identified the need for the *exists* keyword of the `conditional` clause. This was the only way to define a phase that either spans the whole `course_enrolment`—regardless whether it is priority, open, or special—or nothing of it. As an example, Listing 14.1 provides the `timeframe` section of the top-most phase of the *Course Enrolment* group.

Relevant Scenarios: The survey had six participants who self-assessed as competent or expert in load testing. Overall, they specified 36 scenarios. We collect all of them and provide the LCtL instances we derived in the supplementary material (H. Schulz et al., 2020b). In the following, we provide an excerpt from the answers.

```
timeframe:
- !<weekday>
  days: [ monday, tuesday, wednesday, thursday,
      friday ]
aggregation: !<percentile>
  p: 95
```

Listing 14.3: LCtL using the `weekdays` clause.

```
timeframe:
- !<conditional>
  examination:
    is: true
  course_enrolment:
    exists: true
aggregation: !<percentile>
  p: 95
```

Listing 14.4: LCtL instance for overlap of *examination* and *course_enrolment*.

Without knowing the context, most participants specified workload scenarios representing a specific period, such as illustrated in Listing 14.2. Many of these periods correspond to the phases defined by the context. None of the participants explicitly stated to use a varying workload intensity; therefore, we default to the percentile aggregation. One participant also distinguished between weekdays, i.e., workdays and weekends. The LCtL presented in Section 8.5 does not provide a means for expressing this. However, we can easily add a corresponding clause to the `timeframe` section, as illustrated in Listing 14.3. Another participant differentiated between the load of *"peak days," "middle amount,"* and *"base amount,"* which we expressed using the `maximum`, 50th `percentile`, and newly introduced `minimum` aggregations.

When knowing the context, the participants defined similar workload scenarios as before but stated them more precisely. For instance, instead of a date period, they referred to context facets. Several participants stated to test the workload during the overlap of multiple facets, e.g., *examination* and *course_enrolment*, as illustrated in Listing 14.4. Besides, for some scenarios, they specified to test them only under specific conditions, such as the vacation phase only if *"the application can be downscaled"* or to *"only care about graduation ceremony if I knew that complaints existed in that time frame in the past."* Also, it was suggested to prioritize the examination period lower, as *"Students might forgive SIS if the application struggles on these days, but not to [sic] often as they want to know their grades."* In one case, it was unclear whether the statement *"starting with relatively high load 1500 users; ending up with 500 users"* describes a workload phase or a varying workload intensity to be replayed in the load test. In the latter case, we would need to introduce a new aggregation using the provided extension mechanism.

Regarding the answers of one participant, we were not able to derive LCtL instances. They stated to define the relevant scenarios *"based on logical/business steps based on assumed risk"* instead of the recorded workload. We presume they follow a different school of load design than we do, namely fault-inducing rather than representative (Jiang and Hassan, 2015), which our approach does not cover. Still, the LCtL can help *"figure out throughput/ number of users"* of specific groups, which the participant aimed to do.

Special Circumstances: Due to the COVID-19 pandemic, Charles University has taken various measures, which they report online (Charles University, Faculty of Mathematics and Physics, 2020). Among others, the measures we consider to be especially relevant for load testing are the following.

- As of March 11, 2020, students were prohibited from attending class in presence. Teachers were recommended to continue teaching remotely via online tools. The students were required to complete all assignments electronically.
- As of March 16, 2020, all employees had to work from home as far as possible.
- The graduation ceremonies, which should have taken place on April 21, 2020, were canceled. The university planned to have an alternative ceremony at a date to be announced.

```
timeframe:
- !<timeframe>
  from: 2020-03-11T00:00:00
  duration: P4W
aggregation: !<percentile>
  p: 95
adjustments:
- !<users-multiplied>
  factor: 1.2
  group: 0
- !<users-multiplied>
  factor: 1.5
  groups: [ 12, 14, 21, 22, 23, 24 ]
```
Listing 14.5: LCtL instance for testing online tuition.

From these measures, we deduced the following load test scenarios, which cover the special circumstances. For each of them, we provide an LCtL instance in the following.

1. As of March 11, 2020, students and teachers access the SIS more than usual, especially for exchanging course material. Hence, we can test whether the SIS can handle such an increased steady-state workload.
2. In a past occurrence, there was a workload spike during the graduation ceremony phase. If this spike overlaps with other effects, it may stress the system extraordinarily. Once the new date is announced, we can execute a load test with the corresponding spike workload.

Listing 14.5 shows the LCtL instance for the first scenario. We consider a time range of four weeks starting from March 11. As it was unclear how long the online teaching will last when it was announced, we use the four weeks as a first estimate. Also, we expect the workload to decrease rather than increase with ongoing online teaching. Furthermore, we use the steady-state intensity of the 95th percentile. For incorporating the workload increase, we analyzed the groups of the workload model. In the first place, we expect the intensities of the groups related to course material exchange to increase.

```
timeframe:
- !<timeframe>
  from: XT00:00:00
  duration: P1D
context:
  graduation_ceremony:
    is: true
aggregation: !<sharpest-spike> {}
```
Listing 14.6: LCtL instance for testing the delayed graduation ceremony.

The groups we found to be particularly relevant are the following.

- Groups 12 and 21 to 24 frequently (9.3 % to 22.1 %) access the endpoint *courseBrowserUsingGET*, which we correlate with tasks such as accessing course details and material. Furthermore, they call the endpoints *courseCatalogueUsingGET* and *courseScheduleBrowserUsingGET*, which students might use to find the course sites.
- Users of group 14 frequently access the *studentApplicationUsingGET* endpoint (70.4 %), which we identify as being relevant for online learning and study management.
- Group 0 users call many endpoints, including the previously mentioned ones, with a frequency of 4.2 % or more. The most frequent endpoint is *overviewUsingGET*, with 14.8 %. We consider this group to represent normal student activities, which are likely to increase, too, but less than the other groups.

Hence, we can, e.g., increase the intensity of group 0 by 20 % and the intensities of the further listed groups by 50 %. In doing so, we noticed that it is helpful to specify multiple groups per `adjustment`. As a result, the load test reflects the workload predicted by quantitative forecasting, enriched with the manually estimated qualitative forecast of online tuition. Here, we refer to the groups by IDs; labeling them with user-friendly names would increase the comprehensibility of the LCtL instance.

For testing the second scenario, which comprises the spike workload of the rescheduled graduation ceremonies, we can utilize the *graduation_ceremony* context facet. Listing 14.6 illustrates this. As the new date of the graduation ceremony was not set at the time of the study, we use X as a placeholder for the date. X could also be replaced with different values for assessing the effect of different graduation ceremony dates. Then, we define the *graduation_ceremony* to be *true*—such that the forecast will incorporate the effect learned from previous occurrences—and extract the sharpest spike. The difference to the first scenario is that we can rely on quantitative forecasting: the ceremony date is a qualitative information that influences the quantitative forecast.

14.4. Experimental Study with Perfect Forecasting

In this study, we executed an experiment series, addressing RQ3.4: *How well do the continuously learned workload models describe the future workload?* We used our context-tailoring approach with perfect forecasting for generating multiple load tests for the SIS. These tests differ in multiple dimensions, such as the predicted phase, e.g., tuition, the aggregation used, e.g., `sharpest-spike`, and the forecast, e.g., each group individually. We executed the tests against a mock service and compared the results with the recorded SIS workload for evaluating the incrementally learned workload model's ability to predict the recorded workload—i.e., requests and sessions—of the SIS. In this section, we present the experimental method and results. We provide a discussion of RQ3.4 in Section 14.6.

14.4.1. Experimental Method

We generated and executed multiple load tests using our context-tailoring approach with perfect forecasting. In the following, we describe the experiment process and setup we applied.

Table 14.2.: Evaluated Scenarios

phases	aggregations	forecasts	perspectives	total
4	2	2	2	28
start vacation enrolment tuition	percentile sharpest-spike	individual total	start[*] before phase[*]	

[*]For the start phase, the perspectives are equal.

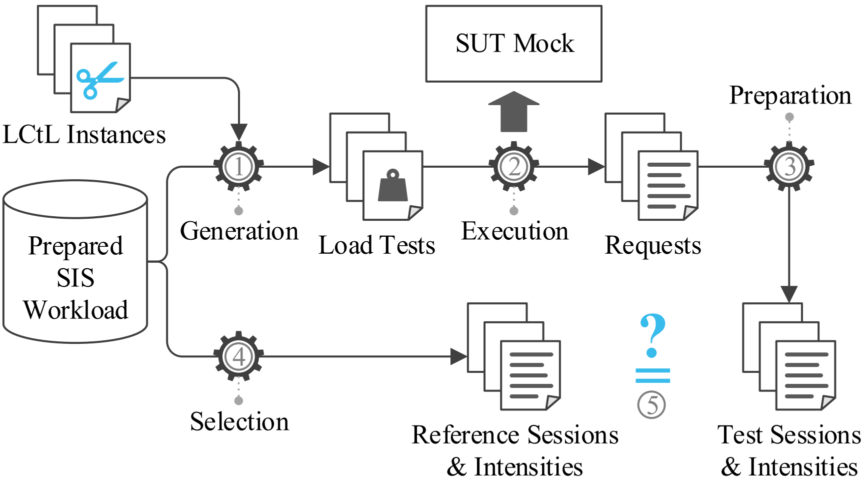


Figure 14.10.: Experiment process.

14.4.1.1. Experiment Process

As illustrated in Figure 14.10, the experiment process comprises the following five steps.

Load Test Generation ①. As summarized in Table 14.2, we generated 28 load tests, which differed in the following dimensions.

1. We predicted different phases using the LCtL `timeframe` section. The first phase is the *start* shortly after the initial clustering period, which we used to assess the ability of the workload model without incremental updates to predict the workload it was built from, which corresponds to existing work (Vögele et al., 2018). Further phases we have based on the phases identified in the previous study. These are *vacation*, *(course) enrolment*, and *tuition* (see Figure 14.9). For preventing overlaps between neighbored phases, we only consider the second half of the vacation phase (starting from August 1), exclude tuition from the enrolment phase, and only use the tuition phase in October.
2. We generated steady-state and varying workloads using the 95th `percentile` and `sharpest-spike` aggregations.
3. We varied the perspective, i.e., the date of the workload model version used, between the *start* and directly *before* each phase. In doing so, we could assess how much incremental updates change the workload model and its ability to predict the reference workload.
4. For evaluating the beneficence of predicting the workload mix in addition to the total intensity, we forecasted the groups' intensities *individually* or all in *total* using the workload mix of the workload model.

Load Test Execution ②. We executed the spike load tests for the duration of the spike and the steady-state tests for three hours (plus ramp-up and cool-down). As a replacement for the SIS, we utilized the SUT mock described in Section 10.2.2. Between two test executions, we restarted the SUT mock using the provided endpoint. Besides, for calculating baselines of the metrics, we executed the load test with *start* phase and perspective, *individual* forecast, and `percentile` aggregation ten further times.

Test Result Preparation ③. For comparison with the reference, we collected the SUT-side requests. We grouped them into sessions using the same IDPA as in the preparation of the SIS workload data (see Section 14.1.2). Hence, we ensured comparability. Furthermore, we calculated the effective

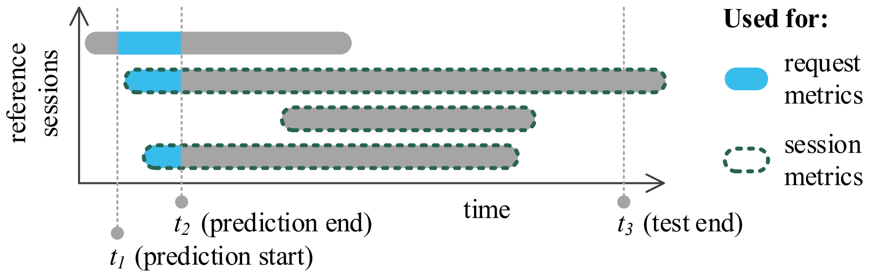


Figure 14.11.: Illustration of the sessions used for calculating request and session metrics.

intensity — i.e., the number of concurrent sessions — generated by the load tests per minute. From the baseline executions, we obtained metric baselines describing the normal variation of a single test execution by using the first execution as the reference (see Section 11.2.2).

Reference Data Selection ④. For assessing the representativeness of the workloads the load tests generated, we selected reference sessions and intensities from the prepared SIS workload. For the spike tests, we selected the sessions and intensities overlapping the time frame of the workload prediction. The percentile tests have a steady-state workload, i.e., the prediction spans only one timestamp. Therefore, we selected the sessions overlapping the timestamp plus the test duration, i.e., three hours, and the intensity at the timestamp.

Result Analysis ⑤. We analyzed the representativeness of the load tests by comparing their results with the respective references we selected. Mainly, we calculated the $\overline{\rho}$ metric, request gap, and session length and duration. Figure 14.11 illustrates the sessions and requests we used. Here, t_1 is the start timestamp of the workload prediction, t_2 the end timestamp, and $t_3 = t_1 +$ test duration. That is, for spike tests, $t_2 = t_3$ and for percentile tests, $t_2 = t_1 + 1$ minute (the intensity granularity) and $t_3 = t_2 + 3$ hours. For the request metrics, we used the reference requests (extracted from the sessions)

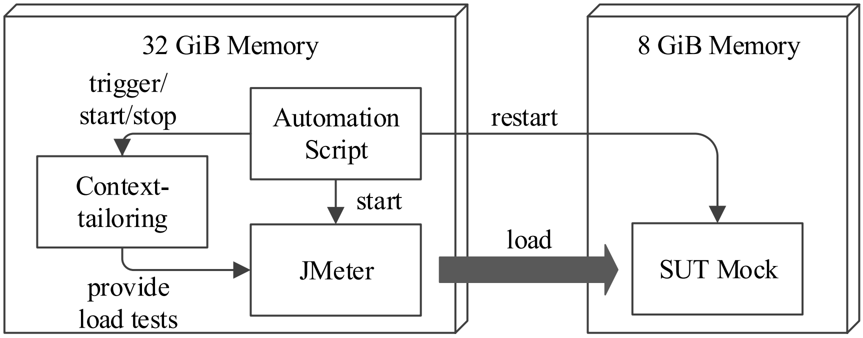


Figure 14.12.: Experiment setup.

between t_1 and t_2, which we extrapolated until t_3 for percentile tests. For preventing overrepresentation of long sessions, we used the sessions starting between t_1 and t_3 for the session metrics.

14.4.1.2. Experiment Setup

Figure 14.12 shows the experiment setup. We used two bare-metal machines, which both had an Intel® Xeon® CPU E5620 with 2.40 GHz clock frequency, 4 cores, and 8 threads, and were connected via a 1 Gbit switch. The first machine had 8 GiB RAM and hosted the SUT mock that collected the requests submitted by the load tests. The second machine had 32 GiB RAM and hosted our context-tailoring approach for generating the load tests, JMeter (Apache Software Foundation, 2020[a]) for executing the load tests, and a script automating the experiment process. It triggered the context-tailoring, started JMeter, and restarted the SUT mock between the test executions. Besides, it stopped the context-tailoring approach to free resources for JMeter when executing the tests. JMeter ran with a heap size of 24 GiB.

14.4.2. Results

In the following, we provide highlights of the results of the experiment series. In the supplementary material (H. Schulz et al., 2020b), we provide all details. We separate the results by the aggregation used, i.e., `percentile` and `sharpest-spike`.

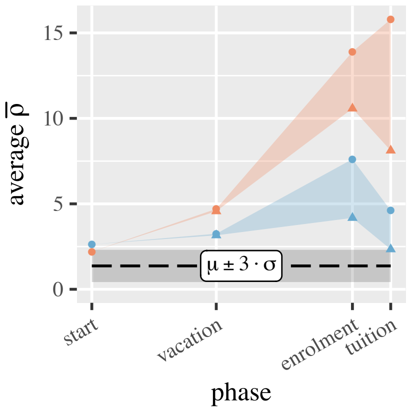
14.4.2.1. Percentile Aggregation

Figure 14.13 provides an overview of the request and session metrics. In the following, we refer to the baseline calculated in step 3 of the experiment process. The average $\overline{\rho}$ exceeds its baseline μ for all tests and, except for the test for the start phase with total forecast, $\mu + 3\sigma$. Hence, the request mix differs (only slightly for some tests) from the reference. For the tests with individually forecasted intensities and perspective before the phase, the difference is the smallest. The phase with the overall largest difference is the course enrolment, which differs sharply from other phases (see Figure 14.2). The request gap is different from its baseline, too.

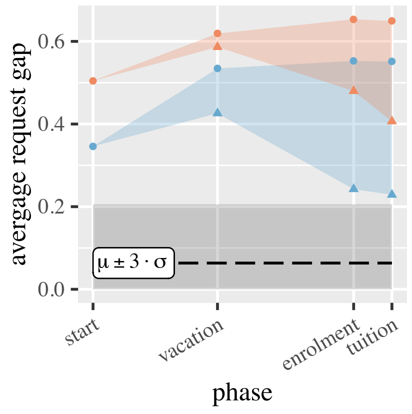
Analyzing the differences further, we identified the gap to the expected numbers of requests, which we obtained as we did in Section 14.2, to be significant as well. As a potential reason, we found that the think time specifications between two endpoints — which are normal distributions, similar to existing work (Vögele et al., 2018) — had a significant negative portion stemming from large variances. These variances are also reflected in the session durations determined in Section 14.2.2. As a load test cannot simulate negative think times, it replaces them with zero, resulting in a biased mean think time. Let $\Delta \sim \mathcal{N}(\mu, \sigma^2)$ be the think time specification and f its density function. Then, the biased mean can be calculated as follows.

$$\mu_{\text{biased}} = \int_{0}^{\infty} x \cdot f(x)\, dx \geq \int_{-\infty}^{\infty} x \cdot f(x)\, dx = \mu$$

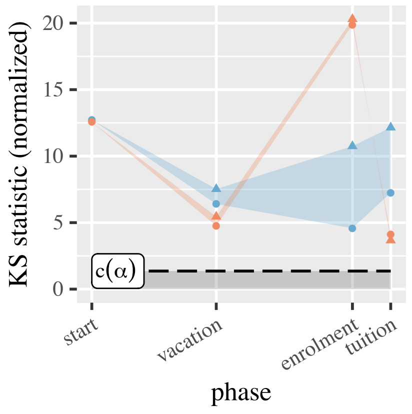
Our explanation is supported by the fact that the lost time correlates with the gap to expected requests with a Pearson correlation coefficient of 0.952, as shown in Figure 14.14. Indicating that it does not stem from bottlenecks in the load driver, the gap does not correlate with the number of concurrent



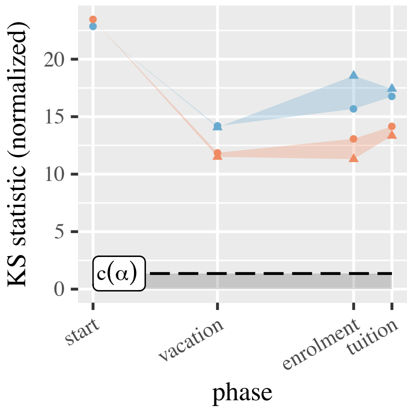
(a) Relative $\overline{\rho}$ metric.



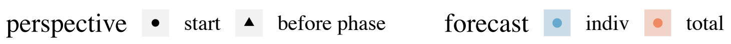
(b) Gap between requests per minute.



(c) Session length (number of requests).

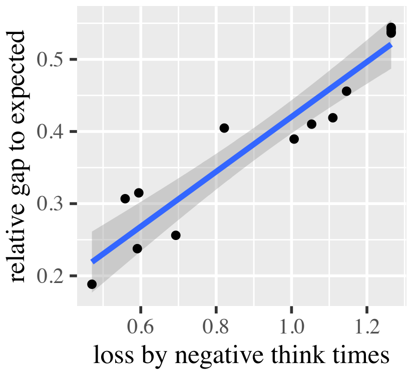


(d) Session duration (seconds).

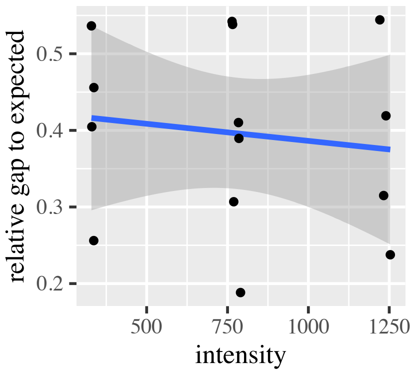


(e) Legend.

Figure 14.13.: Summary metrics of the scenarios with `percentile` aggregation. The dashed horizontal line and gray area illustrate a baseline, which is the critical value $c(\alpha) = 1.358$ for $\alpha = 0.05$ (session length and duration) or based on the mean and standard deviation of the baseline executions ($\overline{\rho}$ metric and request gap). The phases are arranged by date.



(a) Relative time lost due to negative think times.

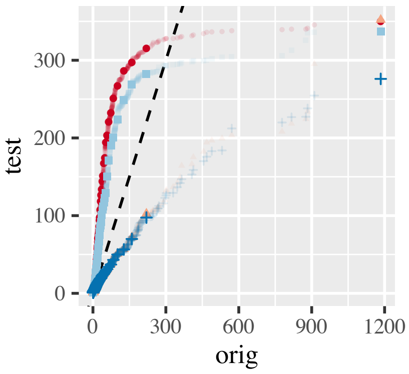


(b) Intensity (number of concurrent sessions).

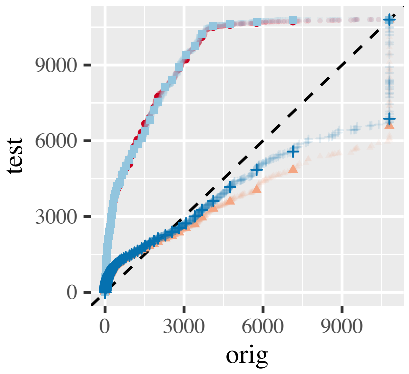
Figure 14.14.: Correlation of the gap to the expected requests of the `percentile` tests with different metrics.

users, where the coefficient is −0.131. Besides, the lost time differs among the groups of the workload model. Hence, it also affects the request mix, which, in turn, affects the $\overline{\rho}$ metric.

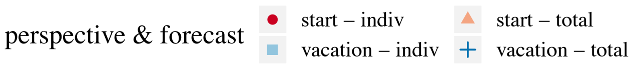
The session metrics (length and duration) give a slightly different picture. As shown in Figure 14.13, the Kolmogorov-Smirnov statistics for the tests with total forecast are below the ones with individual forecast, but still above $c(\alpha)$. An exception is the course enrolment phase, highlighting its peculiarity. Here, the Kolmogorov-Smirnov statistic of the session length with total forecasting is significantly higher than with individual forecasting. For the other phases, quantile-quantile (Q-Q) plots (Figure 14.15) show that the sessions of tests with individual forecasts are longer, except for the tail, while the ones with total forecasts are shorter than the reference. The Q-Q plot for the session duration appears similar, but with the additional effect of time lost due to negative think times, which increases the duration.



(a) Session length (number of requests).



(b) Session duration (seconds).



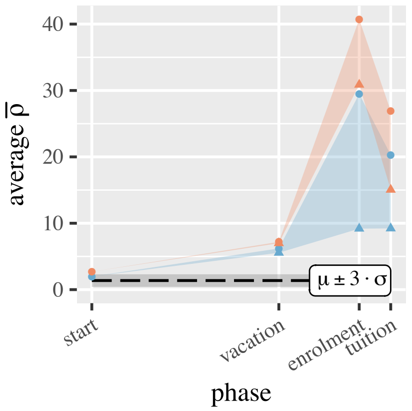
(c) Legend.

Figure 14.15.: Q-Q plots for the *vacation* phase with `percentile` aggregation. The percentiles are highlighted.

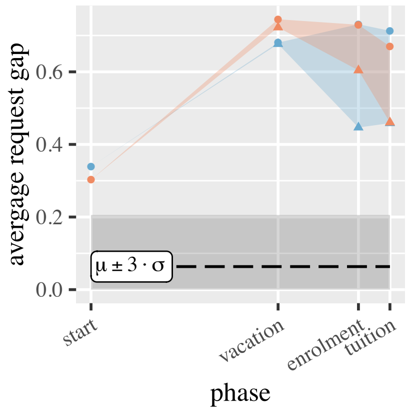
14.4.2.2. Sharpest-spike Aggregation

From an overview perspective, the results of the tests with `sharpest-spike` aggregation have similar characteristics as described before. As Figure 14.16 illustrates, the tests with individual forecast and perspective before the phase generally have the lowest request metrics. Regarding the session metrics, the differences are less pronounced than with the `percentile` aggregation. Another notable difference to the `percentile` aggregation is the $\overline{\rho}$ metric, whose maximum achieved value is significantly higher with the `sharpest-spike` aggregation. The phase with the maximum metric value is the course enrolment.

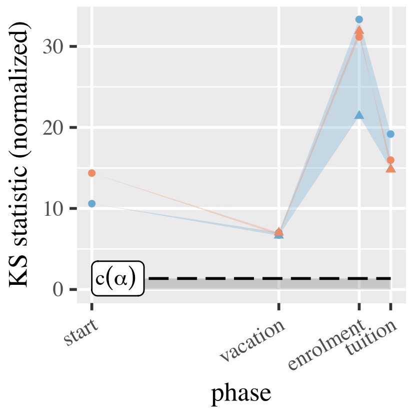
The spike of the corresponding workload is notably sharp, as illustrated in Figure 14.17, at the example of the individual forecast and perspective before the phase. The number of concurrent sessions of the reference workload increases from 479 to 3334 within 100 minutes, and the load tests were able



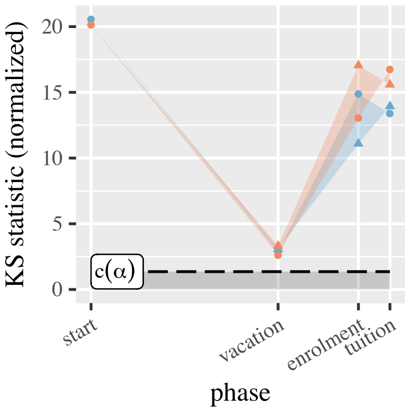
(a) Relative $\overline{\rho}$ metric.



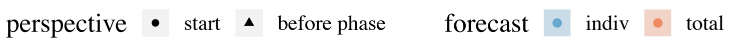
(b) Gap between requests per minute.



(c) Session length (number of requests).

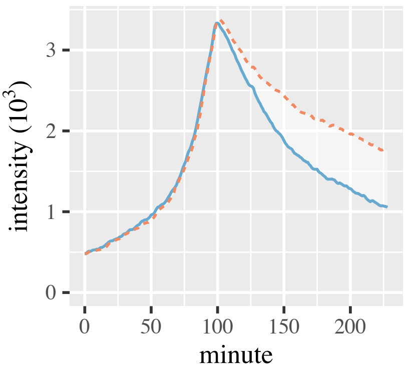


(d) Session duration (seconds).

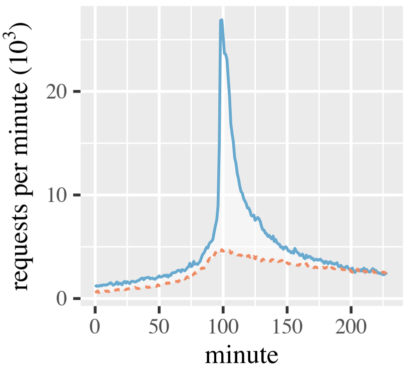


(e) Legend.

Figure 14.16.: Summary metrics of the scenarios with `sharpest-spike` aggregation. The dashed horizontal line and gray area illustrate a baseline, which is the critical value $c(\alpha) = 1.358$ for $\alpha = 0.05$ (session length and duration) or based on the mean and standard deviation of the baseline executions ($\overline{\rho}$ metric and request gap). The phases are arranged by date.



(a) Intensity (number of concurrent sessions).



(b) Number of requests submitted per minute.

type — reference - - - test

(c) Legend.

Figure 14.17.: Intensity and submitted requests for the *enrolment* phase with `sharpest-spike` aggregation, individual forecast, and perspective *before* the phase.

to replay this increase precisely. However, the reference workload contains an increase in the requests submitted at around the intensity peak, which the load tests could not replay. The reason is that the request spike is significantly sharper — with an increase of 2394 % — than the sessions spike — with an increase of 696 %. As the load tests only can vary the number of sessions, the request spike cannot be sharper than the sessions spike. As a consequence, the request gap and $\overline{\rho}$ metric sharply increase during the spike, which is most dominant for the test with the total forecast and start perspective but significant for all.

Table 14.3.: Evaluated Scenarios

phases & aggregations	**forecasts**	**tools**	**data bases**	**total**
2	2	2	2	16
enrolment/highest-spike tuition/percentile	individual total	Telescope Prophet	context pure	

14.5. Experimental Study with Real Forecasting

The second experimental study with the SIS addresses the following research questions.

- RQ3.5: *How representative for future workload scenarios are workload models with forecasted intensities?*
- RQ3.6: *How long does it take to calculate an intensity forecast?*

We generated sixteen further load tests using the forecasting tools Telescope (Bauer et al., 2020) and Prophet (Taylor and Letham, 2018). Hence, as opposed to the previous study, the load tests' workloads were realistically influenced by potentially inaccurate forecasts. Because the SIS dataset is relatively short and, thus, limited regarding time series forecasting, we used the augmented dataset (see Section 14.1.4). We describe the experimental method and present the study results in the following and discuss the research question in Section 14.6.

14.5.1. Experimental Method

For evaluating the ability of workload models with forecasted intensities to predict a reference workload, we reused the experiment setup and a slightly modified version of the process from Section 14.4.1. The modifications are the following.

Load Test Generation ①. We generated a different set of load tests, for which we used the augmented intensities (see Section 14.1.4) for forecasting the workload. The generated load tests varied in the following dimensions, as summarized in Table 14.3.

1. They replayed workload scenarios predicted for two specific days after the end of the (augmented) intensities (May 23, 2019), namely the highest workload during *course enrolment* (Sep. 25, 2019) and the first *tuition* day (Oct. 2, 2019), for which we had access to the request logs. For the tuition, we predicted the 95th `percentile`. Past course enrolment days entailed sharp spikes. Therefore, we used the `highest-spike` aggregation, which is a modification of the `sharpest-spike`. Instead of locating the spike with the sharpest increase, it detects a spike pattern around the peak load. We applied this aggregation instead of `sharpest-spike` because we found that the tools used calculated forecasts that lead to wrongly predicted sharpest spikes. We report more on that in the results section.
2. Similar to before, we forecasted the intensities *individually* and in *total*.
3. We used the *Telescope* (Bauer et al., 2020) and *Prophet* (Taylor and Letham, 2018) tools for computing the intensity forecasts.
4. We expected the context to improve the intensity forecast. For validating this hypothesis, we supported the forecasting tools with the *context* or executed it *pure*ly, i.e., without context. In the enrolment case with context, we defined a special enrolment from 9:30 to 11:00 using the `context` section of the LCtL.

The test with the property combination Telescope/enrolment/highest-spike/individual resulted in a total intensity that was too high for our experiment infrastructure. Therefore, we scaled the workload with a factor of 0.75 and divided the resulting request rates by 0.75 before applying the analysis. For the scaling, we utilized the `users-multiplied` adjustment.

Reference Data Selection ④. For analyzing the load test results, we compared the requests and sessions the tests submitted with the corresponding ones from the two days. For the course enrolment, we aligned the intensity peaks of the load tests and reference. For the tuition, we extracted the 95th percentile.

14.5.2. Results

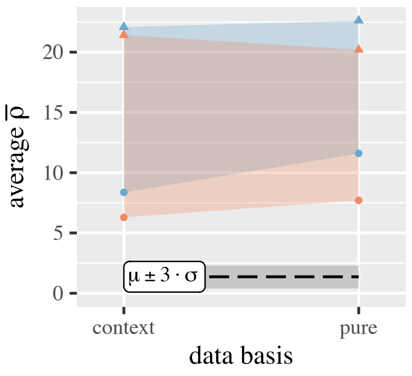
Here, we present the results of the experiment series. First, we consider the load tests predicting the 95th percentile of the first tuition day; then, we provide the results of the load tests simulating the course enrolment spike. Besides, we have identified the duration required to calculate the forecasts to be significantly different among the load tests. Therefore, we analyze it, as well.

14.5.2.1. Tuition Percentile

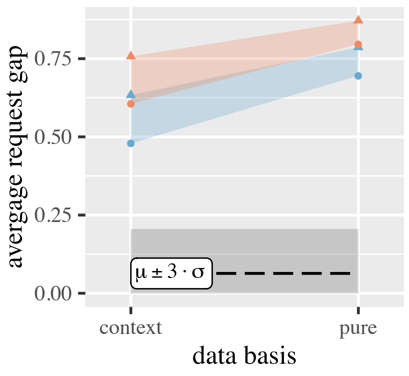
Figure 14.18 provides a summary of the load test results. The request and session metrics are in a similar range as for the tests with perfect forecasting but generally higher, e.g., the average $\overline{\rho}$ for the test with Telescope and individual forecast considering the context is 6.297, as opposed to 2.338 for its counterpart with perfect forecasting. In general, Telescope generates more representative workload predictions than Prophet. An exception is the request gap, which is smaller for Prophet than for Telescope. However, all metric values for the two tools are close to each other.

The individual forecast has the strongest effect on the request metrics. It reduces $\overline{\rho}$ by 48 % to 71 % compared to the total forecast. Also, the request gap is smaller with individual forecasting compared to total forecasting, with differences between 8 % and 25 %. The data basis has a small effect. In all cases except one, using the context reduces $\overline{\rho}$ by 2 % to 39 %. For the test with Telescope applied to the total intensities, the context slightly increases $\overline{\rho}$ by 6 %. However, the context reduces the request gap in all cases.

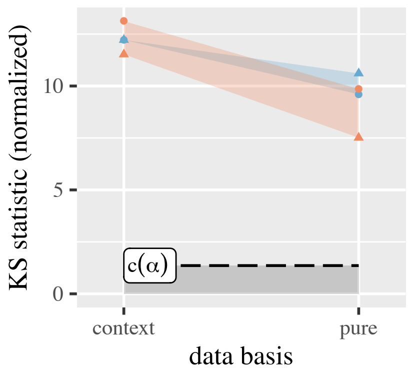
As before, the total forecast produces smaller session metrics, with Q-Q plots similar to Figure 14.15. Besides, Telescope predicts workloads with slightly smaller session metric values than Prophet does in most cases. As all tests' workloads significantly differ from the reference workload, we cannot



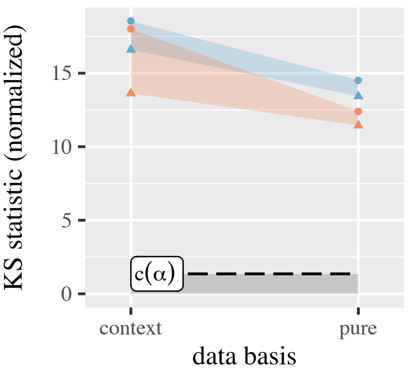
(a) Relative $\overline{\rho}$ metric.



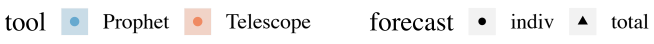
(b) Gap between requests per minute.



(c) Session length (number of requests).



(d) Session duration (seconds).



(e) Legend.

Figure 14.18.: Summary metrics of the tuition scenarios with `percentile` aggregation. The dashed horizontal line and gray area illustrate a baseline, which is the critical value $c(\alpha) = 1.358$ for $\alpha = 0.05$ (session length and duration) or based on the mean and standard deviation of the baseline executions ($\overline{\rho}$ metric and request gap).

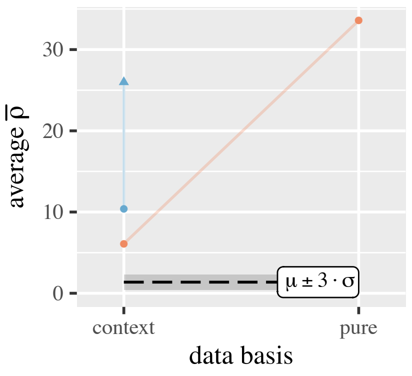
deduce significant differences between Telescope and Prophet regarding the session metrics.

14.5.2.2. Enrolment Spike

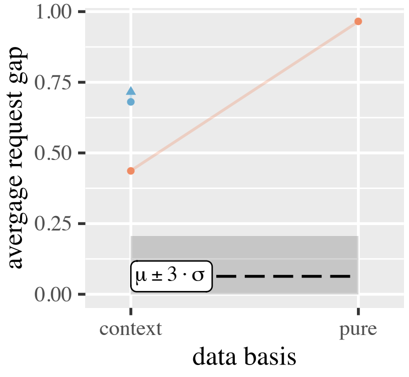
Predicting the sharpest spike during the course enrolment appeared to be challenging for the forecasting tools. In several cases, namely Telescope with total forecasting and Prophet with the pure data basis, the predicted workload did not contain a notable spike. The extracted workload scenarios lasted between ten and twenty hours. Therefore, we only executed the remaining load tests.

Figure 14.19 provides an overview of the resulting metrics. It shows a high variation among the tests. The most representative workload was predicted by Telescope with individual forecasting, using the context. Its metrics are close to the enrolment spike predicted with perfect forecasting. Some metrics, such as the request metrics, are even lower, with $\overline{\rho} = 6.093$ and a request gap of 0.437 as opposed to 9.185 and 0.447, respectively. The prediction without context resulted in a doubled request gap and a $\overline{\rho}$ about as 5.5 times as high as with context. With Prophet, both tests have a similar request gap, but the individual forecast generates a better request mix, as assessed by $\overline{\rho}$. The session metrics are diverse, too. The generally best-predicted session metrics resulted from Telescope with the pure data basis, which, considering the weak request metrics, might be accidental.

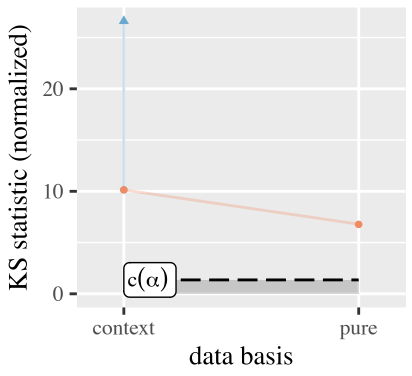
The reason for several inaccuracies regarding the representativeness of the predicted workload scenarios is the shape of the intensity curve, as illustrated in Figure 14.20a. The curve shapes of the numbers of concurrent sessions generated by the tests differ significantly from the reference. Mainly, there are sharp increases shortly after the start, which do not exist in the reference. The increases correlate with the start of the *special* enrolment. Still, Telescope was able to predict the peak intensity accurately when using the context. Remarkably, when not using the context, the intensity curve is not a spike and significantly too low, which explains the high request metrics.



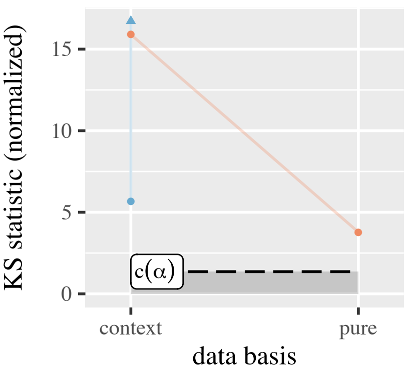
(a) Relative $\overline{\rho}$ metric.



(b) Gap between requests per minute.



(c) Session length (number of requests).

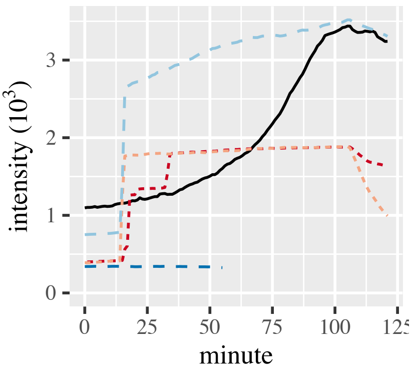


(d) Session duration (seconds).

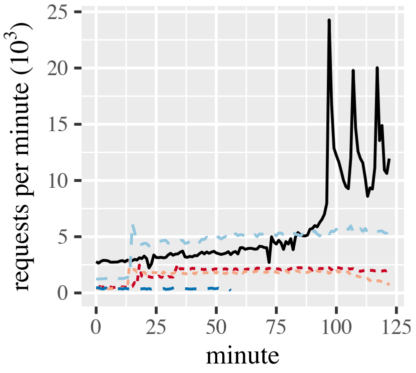
tool
Prophet
Telescope
forecast
indiv
total

(e) Legend.

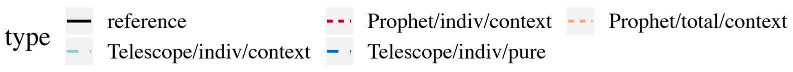
Figure 14.19.: Summary metrics of the course enrolment scenarios with `highest-spike` aggregation. The dashed horizontal line and gray area illustrate a baseline, which is the critical value $c(\alpha) = 1.358$ for $\alpha = 0.05$ (session length and duration) or based on the mean and standard deviation of the baseline executions ($\overline{\rho}$ metric and request gap).



(a) Intensity (number of concurrent sessions).



(b) Number of requests submitted per minute.



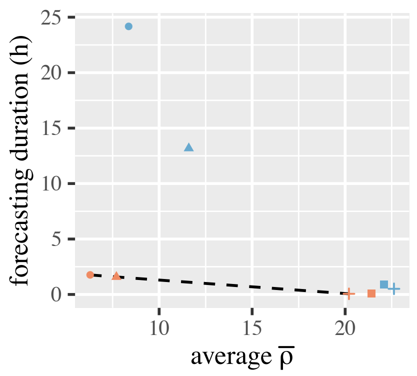
(c) Legend.

Figure 14.20.: Intensity and submitted requests for the spike tests for the *enrolment* phase.

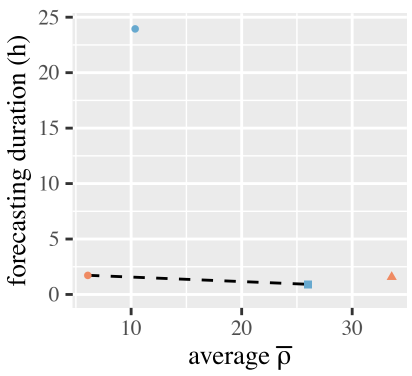
Similar to perfect forecasting, none of the generated load tests were able to replay the spikes in the request rates. As illustrated in Figure 14.20b, the reference request rate contains three sharp spikes that do not correlate with the intensity. Hence, as the load tests only vary the number of users, they cannot generate the request spikes.

14.5.2.3. Forecast Duration

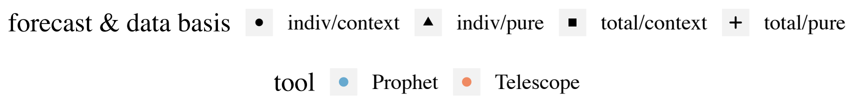
The time needed to compute the intensity forecast differs significantly among the load tests we generated. Figure 14.21 illustrates the forecasting duration in correlation with the $\overline{\rho}$ metric as a measure for the representativeness of the tests. A general trend is that individual forecasting lasts longer than total forecasting, and forecasting considering the context takes more time than pure intensity forecasting. Besides, Telescope is faster than Prophet.



(a) `percentile` aggregation.



(b) `highest-spike` aggregation.



(c) Legend.

Figure 14.21.: Time needed to calculate the intensity forecast correlated with the $\overline{\rho}$ metric per aggregation. The dashed line illustrates the Pareto frontier.

For the tuition percentile tests, the Pareto frontier only contains load tests generated using Telescope. Telescope's pure total forecast is the fastest, with 3.6 minutes, while its per-group individual forecast using the context is the most representative and took 106 minutes. As a comparison, we ran the load tests for three hours. The $\overline{\rho}$ values are 20.203 and 6.297, respectively. Hence, there are significant differences in both dimensions.

For the enrolment spike tests, the Pareto frontier contains Telescope's per-group individual forecast and Prophet's total forecast, both with context. The forecast with Prophet is the fastest with 55 minutes and $\overline{\rho} = 25.990$, while the one with Telescope is the most representative with a duration of 104 minutes and $\overline{\rho} = 6.093$. Here, the difference in the duration is smaller than for the percentile aggregation, as Telescope's forecast took less than twice as long as Prophet, while its $\overline{\rho}$ is only about 23 % of Prophet's $\overline{\rho}$.

14.6. Discussion of Research Questions

In the following, we discuss the research questions RQ3.1 to RQ3.5 defined in Section 5.1. We refer to the context-tailoring approach (Chapter 8) and the studies we have conducted for this purpose, which we presented in the previous sections.

14.6.1. RQ3.1 — Incremental Workload Model Learning

How can we incrementally learn the workload models from observed user sessions for predicting future workload scenarios?

Clustering sessions using an algorithm based on k-means leads to valuable workload models, as our evaluation with the SIS dataset shows. This finding correlates with previous work, which uses k-means and related algorithms for clustering a single bunch of sessions (Menascé et al., 1999; Vögele et al., 2018). Our proposed algorithm applies outlier detection and k-means++ (Arthur and Vassilvitskii, 2007) for the initial clustering and assigns further sessions based on the minimum distance to the cluster centroids and their radiuses. In doing so, it detects up to one new cluster per iteration.

As a crucial aspect of the algorithm, it is fast enough for online execution on a moderately-sized machine. As we intend to apply the algorithm periodically to learn the latest user behavior incrementally, it should not last longer than the time range of sessions clustered. With the SIS dataset, it only needed 27.6 hours for clustering four weeks and up to 2.5 hours for clustering a single week. Hence, it can be applied to an industry-scale application, such as the SIS of a large university, without requiring disproportionate hardware. However, the outlier detection was slow in relation to the actual clustering, which future work might improve.

Limitations of our algorithm are the many parameters to be defined manually and the disability to detect more than one new cluster per iteration. For the SIS dataset, we based on our experience and applied a try-and-improve approach for finding suitable parameter settings. Future work should aim at supporting the user in finding appropriate parameters. As we discuss in Section 8.4.1, alternative algorithms, such as DBSCAN (Schubert et al., 2017), may solve the limitations but do not produce convex clusters. However, we require convexness for selecting a cluster representative, such as the centroid. Hence, future work can also try to adapt our DBSCAN-based algorithm and cope with non-convex clusters by, e.g., selecting different representatives.

14.6.2. RQ3.2 — Influence on Workload Model

> *How much does the incremental learning affect the workload models?*

As assessed in the workload model analysis (Section 14.2), the incremental session clustering resulted in a workload model that slightly fluctuates over time. The behavior models of most of the 26 groups vary in the session duration. Some of them also vary in the session length — i.e., the number of requests submitted per session — or the ratio of requests per endpoint. Most of the fluctuation happens during the course enrolment phase, which has the highest workload.

The fact that new groups occur after several iterations shows the importance of identifying new clusters rather than assigning all sessions to existing groups. Still, the changing request ratio and session length during the course enrolment phase indicate that our clustering algorithm can be improved regarding the assignment of sessions.

Another aspect to be improved in future work is the session duration fluctuation. We observed that it was the most fluctuating session property. We attribute this to the session clustering, which only considers the Markovian request order. The think times between two requests, which dominate the session duration, are added afterward. Future work should decrease the session duration fluctuation by integrating the think times into the clustering.

Finally, we observed that for some groups, the mean session duration minus the standard deviation was negative, stemming from think time specifications with a significant negative portion. As a load test cannot simulate them, it will replace negative think times with zero. Hence, the resulting average session duration is higher than specified. In turn, the request rates are lower than specified. Future work should address this issue in two ways. First, integrating the think times into the clustering might help to separate differently timed sessions and, thus, prevent high think time variances. Second, we propose investigating think time specifications different from normal distributions. One option could be empirical distributions, which also allow for merging think time specifications, as demanded by the incremental updating and the model-based service-tailoring (see Section 7.5).

Concluding, we presume the incremental session clustering to produce valuable workload models if the workload is stable. Highly varying workloads introduce fluctuations, which can impair the representativeness of predicted workload scenarios. Therefore, identifying new clusters, as our algorithm does, is crucial. Future work should investigate improvements of the algorithm, such as detecting more than one cluster per iteration, integrating the think times, and using different think time specifications. The improvements may extend the two algorithms proposed in Section 8.4.1.

14.6.3. RQ3.3 — Expressing Workload Scenarios

How expressive is the Load Test Context-tailoring Language concerning workload scenarios of a production system?

Due to its flexibility, the LCtL is suitably expressive for workload scenarios of the SIS. We conclude this finding from the expressiveness evaluation. First, we were able to express all workload phases we identified using the `timeframe` section. Second, we could define LCtL instances for most of all scenarios the participants of the expert survey specified. Last, we also showed the LCtL is suited for describing complex scenarios that require qualitative forecasting, such as during the COVID-19 pandemic.

The flexibility, i.e., the ability to add further clauses or constructs easily, was a vital feature of the language. For instance, many workload scenarios required an *exists* keyword for selecting the course enrolment phase, which we, therefore, added to the core language. Also, specific concerns, such as selecting weekdays, can be integrated by adding new clauses to the `time-frame` section. For `aggregations` and `adjustments`, we even provide an extension endpoint, which we had to use in few cases of the expert survey.

A limitation of the LCtL is the design of non-representative workloads, such as fault-inducing ones (Jiang and Hassan, 2015), which our approach does not cover by design. However, LCtL instances can describe specific aspects, such as the number of users of specific groups, which an expert can then assemble to a fault-inducing workload specification. Besides, as our approach uses the WESSBAS-DSL, it can be integrated with the work by Vögele (2018), which closes the gap between representative and fault-inducing load tests. Precisely, the approach can select expectedly highly-demanding scenarios from the workload our approach predicts.

14.6.4. RQ3.4 — Describing Future Workload

How well do the continuously learned workload models describe the future workload?

The incrementally learned workload model describes the reference workload accurately, except for two influences. First, as also discussed for RQ3.2, the think time specifications have a too large variance, which induces a significant request gap. As the user groups are affected differently, the request mix and, thus, the $\overline{\rho}$ metric are affected, too. Taking that into account, the latest version of the workload model, in combination with individual forecasting, generates representative load tests, while especially total forecasting gives a worse result. Hence, predicting the workload mix is superior over predicting the total intensity only. The fact that the sessions of the individually forecasted tests tend to be longer than the reference, while the tests with total forecasts have shorter sessions, supports this finding, as Markov chains have generally been found to generate long sessions (Vögele et al., 2018).

Second, the workload model is limited regarding predictions of sharp spikes. While corresponding load tests executed the correct number of concurrent sessions, the request rate and mix were different from the reference. A reason might be that the think times during the spike differ from other phases. Hence, as previously suggested, integrating the think times into the session clustering might improve spike predictions, as the workload model can then separate user groups with different timings. Also, modeling an open instead of closed workload (Schroeder et al., 2007) — i.e., considering the session arrival rates instead of concurrent sessions — might help to predict spikes, which, however, requires careful modeling of the session length and duration. As our and previous evaluations (Vögele et al., 2018) indicate, Markov chains are not suited for that, as they tend to generate too long sessions.

A final aspect we propose to investigate in future work is the fact that the perspective before the predicted phase was more representative than the start perspective in most cases. Applying the incremental session clustering to datasets spanning a longer time frame can reveal whether this effect dissolves, oscillates with the workload's seasonality, or even strengthens over time. The first case is desirable, as it means that the workload model can be used for long-term predictions if it has been learned from a large enough time frame. The second case is acceptable for short-term predictions. In the last case, the clustering algorithm needs to be refined.

14.6.5. RQ3.5 — Representativeness of Forecasted Workload Models

How representative for future workload scenarios are workload models with forecasted intensities?

Load tests with forecasted intensities can be mostly as representative as with real forecasting when considering the context and per-group intensities. For steady-state workloads, the tests are only slightly less representative compared to perfect forecasting. Here, the individual forecasts made the largest difference. When predicting spikes, it is crucial to use the context, as all other tests did not contain a notable spike. We presume that, on the one hand, the context helps the forecasting tool separating the spikes from anomalies, while, on the other hand, the group's individual intensities allow detecting the spike better, as it occurs differently in different groups.

A limitation of the spike forecast is predicting transitions between context facet states, e.g., *open* to *special* course enrolment, which can cause a sharp change in the intensity curve. We presume the reason is that the tools learn the average influence of a specific state in the past and apply it to the whole state in the future. For our purposes, we rather need a forecasting approach that also learns the change in the intensity curve, which needs to be addressed by future work.

Regarding the forecasting tools used, Telescope predicts slightly more representative workloads than Prophet does. For steady-state workloads, this is mainly reflected in the request metrics. For spike workloads, Telescope also predicts intensity curves that are closer to the reference, even if it suffers from the context transitions and the uncorrelated request rates. However, Telescope necessarily requires the context and individual per-group intensities, while Prophet performs better on total intensities.

14.6.6. RQ3.6 — Forecasting Duration

How long does it take to calculate an intensity forecast?

The forecasting duration strongly depends on the tool and input used. In general, Telescope is faster than Prophet. As we also assessed Telescope to predict slightly more representative workload scenarios, we conclude it is preferable over Prophet, especially because some of Prophet's forecast calculations take significantly more time than the generated load tests run. Within each tool, more representative workload scenarios require a longer forecasting duration. For instance, the tuition percentile load test with the shortest forecasting duration only took 3.6 minutes, while the most representative one took 106 minutes. However, the fastest test also has a high $\overline{\rho}$; thus, we cannot consider it to be representative of the reference workload.

Hence, forecasts for representative workload scenario prediction in our specific case require about 100 minutes on the average-sized infrastructure we used. For the load tests we executed, this is more than half the execution time. The reasons why high representativeness requires long-lasting forecasting are the consideration of the context and the per-group individual intensities. Both add information to the forecasting tool, which increases its calculation duration. The individual forecast has a higher impact because each group's intensity needs to be forecasted individually, and the current implementation does not apply parallelization.

Two opportunities for speeding up the forecasting are parallelization and reduction of the number of groups. Parallelization is mainly promising because the per-group intensity forecasts are independent of each other. Hence, in our case, it can decrease the forecasting duration by a factor of almost 26, which is the number of groups. However, this requires a larger infrastructure than we used, especially with more CPU cores. Fewer groups allow for fast forecasting without larger infrastructure. Besides, one of the expert survey participants, who self-assessed as an expert, found that *"26 user group sounds too much."* In order not to reduce the behavior modeling precision, one approach is to forecast groups with similar intensity seasonalities together in total, without merging the behavior models.

14.7. Threats to Validity

We identified the following threats to the validity of our context-tailoring approach and its evaluation, grouped by the validity type.

14.7.1. Conclusion Validity

Typically, statistical tests are an essential means for drawing conclusions from numerical measures. Here, we have not applied statistical testing for assessing the representativeness of the generated load tests. Statistical tests are not suited for comparing predicted workload scenarios with a reference workload, as the predictions naturally deviate from the reference. Instead, we have measured the distance to the reference with metrics introduced and successfully applied in existing work (H. Schulz et al., 2020c; Vögele et al., 2018, and Chapter 12).

14.7.2. Internal Validity

During three short-term (up to few hours) periods of the session clustering, we lost a low percentage ($< 1\,\%$) of data due to platform issues and clock change. These data losses might have influenced our results. However, they are negligible compared to the total size of the dataset and likely to happen in realistic settings. During test execution, we prevented interactions between the load driver and the system under test by using separate machines. Also, we validated that the load driver did not run into overload situations by correlating the request gap with the intensity.

For applying real forecasting, we augmented the SIS dataset, which might have influenced the forecasts. Using a dataset with a longer period of data available would be beneficial. However, to the best of our knowledge, the SIS dataset is unique regarding the information content and extent of real-world session-based workloads. Also, time series augmentation is commonly applied (Wen et al., 2020).

14.7.3. Construct Validity

We identified several limitations of existing workload modeling and forecasting approaches. For preventing limitations due to the tool choice, we used state-of-the-art concepts and tools that have extensively been evaluated (Bauer et al., 2020; Taylor and Letham, 2018; Vögele et al., 2018). Particularly, Markov chains have been assessed a reasonable modeling concept for load tests (Z. Li and Tian, 2003). Also, the limitations do not stem from our implementations, as they are related to the session clustering and Markov-chain-based workload modeling introduced by Menascé et al. (1999) and Vögele et al. (2018), and time-series forecasting, which we have applied as a black box. Besides, our framework allows using different tools, which can be evaluated in future work.

14.7.4. External Validity

The SIS dataset has particular characteristics, such as semester phases sharply differing in the workload. The workloads of other systems can have different characteristics leading to different results. Therefore, future work should aim at finding more datasets related to the one used in this paper and evaluate our context-tailoring approach on them.

14.8. Summary

In this chapter, we have evaluated our context-tailoring approach in four different studies with the SIS of Charles University, Prague. While the general approach was well suited for expressing and generating representative workload scenarios, we have encountered limitations of existing approaches we used as building blocks, which need to be resolved in future work. On the one hand, the prediction accuracy needs to be improved, e.g., by integrating the think times into the session clustering. On the other hand, this likely will lead to an increased number of user groups, which, in turn, will increase the time needed for calculating forecasts. Hence, the challenge is to find a combination of workload modeling concepts and forecasting approaches that capture user behavior accurately while enabling efficient predictions.

In the next chapter, we evaluate our Behavior-driven Load Testing (BDLT) approach, which bases upon the context-tailoring. There, we focus on further expressiveness and usability evaluation.

Chapter 15

Evaluating Load Testing for Non-experts

In this chapter, we provide two case studies evaluating our load testing approach for non-experts (Chapter 9). We have introduced the Behavior-driven Load Testing (BDLT) language, which allows a user to define load tests in template-based natural language. Based on such a definition, we automatically generate and execute the load test leveraging our approaches to automated parameterization, service-tailoring, context-tailoring, and the BenchFlow test execution approach by Ferme and Pautasso (2018). The evaluation of the BDLT approach addresses the sub-questions of RQ4: *How can we leverage automated tailored load test generation and automated load test execution for enabling load testing for non-experts?*

The two case studies presented in this chapter comprise an industrial and laboratory setting. The first study assesses the expressiveness and use cases of the BDLT language for a DevOps team developing and operating an Internet of things (IoT) system. In collaboration with the team, we developed four BDLT definitions and received feedback. The second case study evaluates the expressiveness for complex laboratory load test concerns at the example of microservice scalability assessment. Here, we based on the approach presented in Section 9.3 and experiments conducted in a previous study.

The case studies show that the BDLT language is sufficiently expressive for many load testing concerns. In the industrial case study, we were able to express all concerns of the DevOps team, even though we had to use extension mechanisms for handling outage scenarios, which entailed a high recovery spike. The feedback we received from the team was consistently positive. In particular, they highlighted the benefits of natural language, which appeared to foster collaboration. Furthermore, they suggested using the language in the acceptance criteria of Scrum user stories. In the laboratory case study, we were able to express all experiments from our previous work in a single BDLT definition, but at the expense of executing more experiments than before. The reason is that we were not able to express parameter combinations as concisely as in the previous work, due to limitations of natural language. We provide supplementary material online (H. Schulz et al., 2019d).

The remainder of the chapter is structured as follows. In Section 15.1, we present the industrial case study, followed by the laboratory case study in Section 15.2. In Sections 15.3 and 15.4, we discuss the research questions and lessons we learned, considering the case study results. Section 15.5 discusses threats to the validity of our work. We conclude the chapter with a summary in Section 15.6.

This chapter is a revised version of Chapter 4 of our previous publication (H. Schulz et al., 2019c) and the corresponding supplementary material (H. Schulz et al., 2019b), extended by content from our joint publications below.

- H. Schulz, D. Okanović, A. van Hoorn, V. Ferme, and C. Pautasso (2019c). "Behavior-Driven Load Testing Using Contextual Knowledge — Approach and Experiences." In: *Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE 2019)*. ACM, pp. 265–272
- A. Avritzer, V. Ferme, A. Janes, B. Russo, H. Schulz, and A. van Hoorn (2018). "A Quantitative Approach for the Assessment of Microser-

vice Architecture Deployment Alternatives by Automated Performance Testing." In: *Proceedings of the 12th European Conference on Software Architecture (ECSA 2018)*. Vol. 11048. Lecture Notes in Computer Science. Springer, pp. 159–174

- A. Avritzer, V. Ferme, A. Janes, B. Russo, A. van Hoorn, H. Schulz, D. Menasché, and V. Rufino (2020a). "Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-Based Approach Leveraging Operational Profiles and Load Tests." In: *Journal of Systems and Software* 165, p. 110564

15.1. Industrial Case Study

In this case study, we applied our BDLT approach to an industrial context for evaluating the research questions below.

- RQ4.1: *How expressive is the BDLT language in regards to load test concerns of industrial use cases?*
- RQ4.2: *How would BDLT be used in industrial contexts?*
- RQ4.3: *What are the benefits and limitations of using BDLT in comparison to defining load test scripts?*

We collaborated with an industrial partner from the logistics sector for using the BDLT language to express their relevant load testing concerns. Furthermore, we received feedback. In the following, we describe the case study method we applied and present the results. In Section 15.3, we will discuss the results concerning the research questions.

15.1.1. Case Study Method

In this section, we describe the case study design and planning, the data we have collected, and the analysis we applied to the data.

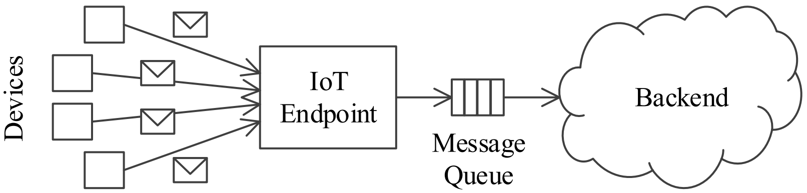


Figure 15.1.: Illustration of the architecture of the IoT system.

15.1.1.1. Design and Planning

We applied our approach to an industrial partner from the logistics sector. Following DevOps practices (Bass et al., 2015), the company developed and operated an IoT system running in a Cloud environment. The underlying Cloud technologies were Docker (Docker Inc., 2020) and Kubernetes (The Linux Foundation, 2020). Figure 15.1 sketches the architecture of the IoT system. Devices are sending messages to an IoT endpoint, which forwards the messages to the backend application via messaging queues. The messages are processed depending on a contained message type. The device IDs and the order, timing, and type of the messages build a structure similar to user sessions.

In order to answer the research questions, we developed BDLT definitions that express the load test concerns the industrial partner had and collected feedback regarding the benefits of these definitions. In doing so, we proceeded as follows. We had five meetings in total with the industrial partner. In the first two meetings, we presented our general research plan and discussed high-level architectural and organizational aspects of their IoT system. As an outcome, we focused on one DevOps team that was working on load testing. In the third meeting, we defined the scope of the collaboration and received production data to use in our case study. After that, we had two iterations, each consisting of the two steps of (1) internally defining BDLT definitions that met the state of knowledge we had at this point and (2) refining the BDLT definitions in another meeting with the DevOps team and collecting feedback. In each of these meetings, we presented the

BDLT definitions as well as representations of the generated load tests to be executed: BenchFlow load tests and graphs visualizing the workload models.

15.1.1.2. Data Collection

Our primary source of information was the BDLT definitions we derived in collaboration with the DevOps team. Furthermore, our partner gave us access to the following data[1], which we incorporated into the discussions with the DevOps team for illustrating the meaning of the BDLT definitions:

- the message logs of a small subset of devices from one week, where each log consists of a timestamp, a device ID, and a message type,
- the load intensity over time, i.e., the number of messages per hour for one year,
- contexts we identified, which influence the load intensities.

Finally, we received oral feedback in the meetings with our partner, which we documented and analyzed.

15.1.1.3. Data Analysis

For transforming the BDLT definitions into the load tests and visualizations we presented our partner, we preprocessed the data. Using the WESSBAS approach (Vögele et al., 2018), we transformed the message logs into a workload model representing the device behaviors as Markov chains and the relative frequencies (mix) of the Markov chains. From the number of messages per hour, we extracted the load test intensities. Regarding the workload context, we identified two specific context facets in the data: public holidays and recoveries from outages of the Cloud infrastructure that can happen irregularly. Public holidays turned out to decrease the intensity. Recoveries, in contrast, significantly increase it, because the devices buffer all messages locally during an outage and send them during the recovery.

[1]For the sake of confidentiality, we do not provide the exact dates or values in the following and add randomly chosen obfuscation factors per hour, day, and week as well as for the global trend to all plots.

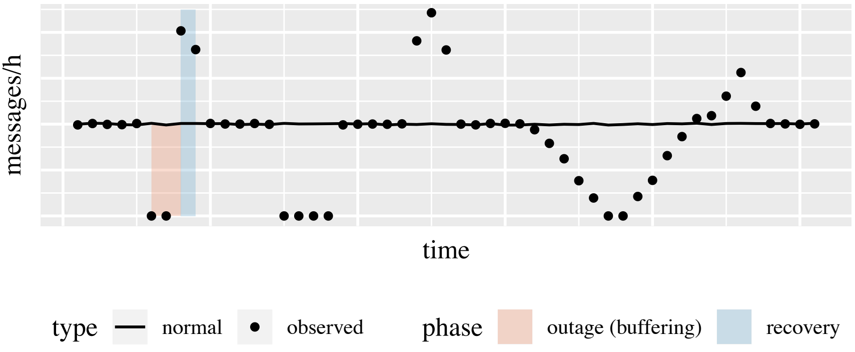


Figure 15.2.: Illustration of the recovery detection. Only the first anomaly is labeled as a recovery: the second misses a spike, the third an outage, and the fourth is de- and increasing too slowly.

Hence, it is essential to prepare for potential future recoveries as a what-if analysis.

We identified past recoveries by processing the message rates, as illustrated in Figure 15.2. We labeled a workload spike as a recovery with a specific severity if the intensity fit the following template: first, it is less than a tenth of the average intensity for that hour and weekday and then, it is higher than two times the average intensity, which constitutes the recovery spike. The recovery severity we calculated as the difference between the messages sent during the outage and the usually sent messages.

For forecasting the workload intensity during expected future outages, we registered an extension, as described in Sections 8.4.2 and 9.2.2.2. If the `event` clause *when an outage happened from <start> to <end>* is specified, the extension determines the expected number of buffered messages by doing separate forecasting. The actual forecasting of the test workload is done in a second step, using the determined number of buffered messages.

Using the data prepared as described above, we were able to transform the BDLT definitions into load tests and visualizations of the tests' workload models. We showed these artifacts to the DevOps team and asked whether they

Table 15.1.: Overview of BDLT Definitions and Corresponding LCtL Instances

name	**timeframe**	**context**	**aggregation**	**adjustments**
configuration exploration	`timerange`	–	`maximum`	–
continuous quality assurance	`timerange`	–	`percentile`	–
recovery spike	`timerange`	*outage*	`as-is`	–
more devices	`timerange`	–	`maximum`	`users-multiplied`

correspond to their expectations. Based on their feedback, we assessed the expressiveness, usefulness, benefits, and limitations of the BDLT language.

15.1.2. Results

Our industrial partner reported four different load test concerns, which we defined using the BDLT language. Table 15.1 provides an overview. The *configuration exploration* test aims at identifying the optimal configuration of the system under test. After that configuration has been established, the *continuous quality assurance* test is executed continuously, e.g., every night, to detect whether the performance of the system remains stable. For preparing for future outage recoveries, our partner demanded the *recovery spike* test. Finally, the *more devices* test covers the planned scenario of adding more devices to the system. In the following, we present the BDLT definitions and describe their transformation to BenchFlow tests using our context-tailoring approach. As we did not make use of the `services` clause — which did not exist at that time —, the service-tailoring approach was not involved.

Please note that we conducted this case study before introducing the LCtL. Therefore, we performed parts of the transformation of a BDLT definition to a BenchFlow test manually. Using the LCtL, the transformation would have been automated. Thus, we provide details about the LCtL instances derived from the BDLT definitions as an addition to our previous publication.

```
Given the next three months
and the number of users set to the maximum,
when varying the CPU cores between 0.5 and 4 in steps of 0.5
and varying the number of instances between 1 and 5
and varying the RAM among (1GB, 2GB, 4GB),
then run each experiment for 1 hour
and ensure the average CPU load is less than 15%
and ensure the message latency is less than 2 seconds.
```

Listing 15.1: BDLT definition: configuration exploration.

15.1.2.1. Configuration Exploration

Listing 15.1 provides the BDLT definition for the configuration exploration test. It uses the maximum expected intensity of the next three months to assess the performance of the system under test (SUT) under different configurations of the CPU, the number of instances, and the RAM. Each experiment should last one hour, and CPU load and message latency thresholds define whether an experiment passes or fails.

The context-tailoring for transforming the BDLT definition to a BenchFlow test is illustrated in Figure 15.3. The LCtL instance contains a `timerange` clause corresponding to the first *given* clause and a `maximum` aggregation corresponding to the second clause in *given*. Hence, our approach calculates a forecast, as illustrated in the figure, and extracts the maximum value. The result is a steady-state workload model with a constant intensity.

Therefore, the `load_function` of the BenchFlow test is a constant value. Because the *varying* keyword is used in the *when* clause to vary system configurations, the BenchFlow goal `exhaustive_exploration` is used, testing all configuration combinations. The *then* clause defines each experiment to run for at most one hour, defined using the `steady_state` field. Besides, the *ensure* keywords define `qualiy_gates` on CPU and message latency metrics.

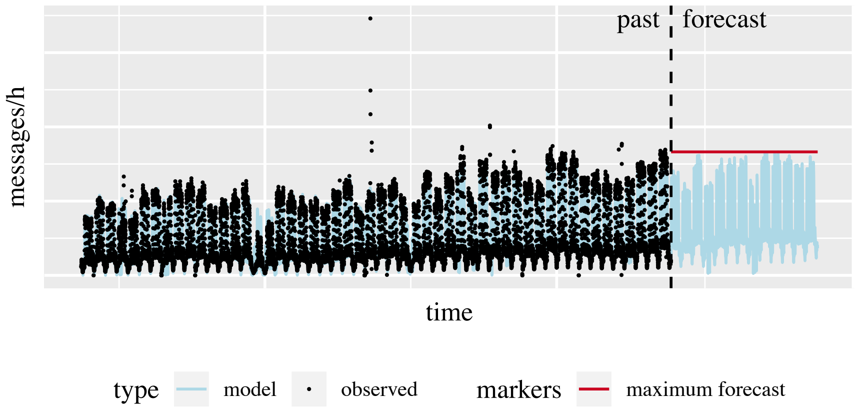


Figure 15.3.: Observed intensities and forecast for the configuration exploration test (obfuscated).

```
Given 2018
and the number of users set to the 95th percentile,
then run the experiment for 1h
and ensure the number of instances is less than 3
and ensure the summarized cost is less than X.
```

Listing 15.2: BDLT definition: continuous quality assurance.

15.1.2.2. Continuous Quality Assurance

The continuous quality assurance test is a simple load test and is easily expressible in the BDLT language, as shown in Listing 15.2. Instead of relying on a workload forecast, the *given* clause states to use the 95$^{\text{th}}$ percentile of the number of users for the year 2018. Hence, repeated generations of this test result in the same workload model, allowing for comparison. Because it is a simple load test, there is no *when* clause. The *then* clause defines to execute the test for one hour and to compare its results with the number of instances and cost thresholds. These metrics are of interest because the test is to be executed in an environment with auto-scaling in place.

The corresponding LCtL instance contains a `timerange` clause for the year 2018 and a `percentile` aggregation. Hence, the context-tailoring uses the past intensities for calculating a steady-state workload. The resulting BenchFlow test is similar to several aspects of the configuration exploration test, but with the difference that the test goal is `load`, denoting a single load test. Besides, it holds different metrics in the `quality_gates` field, namely the number of instances and cost, which is retrieved from the Cloud services.

15.1.2.3. Recovery Spike

The recovery spike test has the goal of preparing for load intensity spikes that might happen in the future because of outages. Listing 15.3 provides the BDLT definition. Because the current intensity strongly influences the number of messages that get buffered and, thus, the spike height, the test focuses on a specific date in the *given* clause. The *when* clause defines the expected outage by utilizing the *event* statement. The *then* clause defines to execute the test for two hours, which is one hour for the spike and one hour for the usual load, and the queue length at the end of the test as the pass criterion.

The LCtL holds the defined time range in a `timerange` clause and additionally sets the *outage* context facet to *true*. Because there is no different statement in the BDLT definition, the aggregation is `as-is`. The context-tailoring predicts the spike load curve for the test in a two-step forecast using the registered extension, as illustrated in Figure 15.4. First, the extension calculates the *recovery severity* as the expected number of buffered messages. For that, it calculates a forecast assuming that no outage happened. Then,

Given *2018/10/15 9:00,*
when *an outage happened from 2018/10/15 7:00 to 2018/10/15 9:00,*
then *run the experiment for 2 hours*
and *ensure the final queue length is less than 100.*

Listing 15.3: BDLT definition: recovery spike.

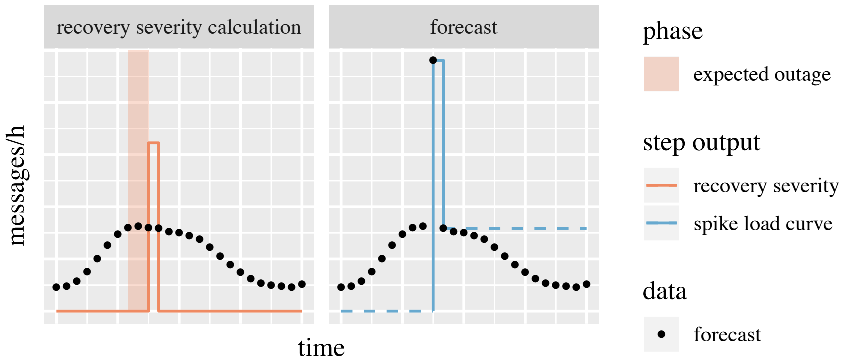


Figure 15.4.: Two-step forecasting of the recovery spike load curve using the *outage* extension, which calculates the recovery severity.

it determines the number of buffered messages as the number of sent ones during the outage and sets the *recovery severity* facet accordingly. In the second step, the regular forecasting is done, incorporating the recovery severity. The `as-is` aggregation extracts the spike load curve.

The resulting BenchFlow test is a simple load test because there is no configuration exploration. The `load_function` is a step function replaying the expected spike curve. The `steady_state` is two hours, as defined, and the queue length threshold is the single `quality_gate`.

15.1.2.4. More Devices

We provide the last BDLT that was relevant for our industry partner in Listing 15.4. It covers the scenario when more devices are added to the system at a known point in time in the future. This circumstance is unknown to the forecaster and has to be added as a user-defined input. For that, the *given* clause defines the date in the future when the devices will be added. The number of users is defined as the maximum forecasted intensity increased by a given percentage, e.g., 30 %. Besides, a custom configuration of the system is used, which our industry partner planned to use on that date. Here we use the number of instances as an example. Because there is

```
Given calendar week 5 in 2019
and the number of users set to the maximum increased by 30%
and the number of instances is 4,
then run the experiment for 5 hours
and ensure the average CPU load is less than 20%.
```

Listing 15.4: BDLT definition: more devices.

only a fixed set of test parameters, there is no *when* clause. The *then* clause defines to run the experiment for five hours and to use the CPU load as the pass criterion.

The context-tailoring predicts the steady-state maximum workload using the same forecasting mechanism as for the configuration exploration test. Additionally, it multiplies the determined intensity with 1.3, to account for the *increased by* 30 % statement. For that, the LCtL holds a `timerange` clause defining the *calendar week 5 in 2019* and the `maximum` aggregation. Besides, it contains a `multiply-users` adjustment with a factor of 1.3. Again, the generated BenchFlow test has the `load` goal and a constant `load_function`. The `steady_state` is five hours, and there is a `quality_gate` on the CPU load metric.

15.2. Laboratory Case Study

In this second case study, we investigate the expressiveness of the BDLT language for laboratory concerns, addressing RQ4.4: *How expressive is the BDLT language regarding the load test concerns coming from laboratory experiments?* We consider an experiment series from our previous work (Avritzer et al., 2020a), which assesses the scalability of different deployment alternatives of a microservice application. In Section 9.3, we have introduced a template for defining such experiments. In this study, we investigate how accurately we can express the specific experiments. Below, we provide the case study method and results. The research question we will discuss in Section 15.3.

15.2.1. Case Study Method

In this section, we describe the method we applied for this case study.

15.2.1.1. Design and Planning

For investigating the expressiveness of the BDLT language for complex load test concerns, such as from laboratory contexts, we have selected a series of experiments from our previous work. These experiments belong to the scalability assessment approach we described in Section 9.3. Due to two reasons, the experiments are adequate for challenging the expressiveness of our language: they vary the configuration of the SUT and comprise multiple load tests per configuration.

The SUT we have utilized in the experiments is the Sock Shop (Weaveworks, Inc., 2020), which we have introduced in Chapter 13 already. We decided to use this SUT, as it is known to be a representative microservice application (Aderaldo et al., 2017) and allows for different deployment alternatives. As summarized in Table 15.2, we have executed load tests for ten different configurations comprising different amounts of memory, numbers of CPU shares, and numbers of instances of the *carts* service. Furthermore, we have executed load tests with six different intensities ranging from 50 to 300. Hence, there are 60 load test executions overall, which we aimed to express in a single BDLT definition.

15.2.1.2. Data Collection

The data we collected for the study comprises the experiment configuration and results from our previous work. While we have conducted multiple experiment series in different environments, we focus on the bare-metal environment (HPI in the publication). The data we base on is:

- the SUT configurations as shown in Table 15.2,
- the load tests executed,

- the experiment results (Domain-based metric values), as also shown in Table 15.2.

Furthermore, we attempted to define the same experiments using the BDLT language, resulting in a BDLT definition. As we were involved in the experiment design and execution, we were suitably able to decide whether the definition corresponds to the experiments.

15.2.1.3. Data Analysis

We assessed how accurately the BDLT definition describes the experiments by deriving the hypothetically executed — given we would apply our approach to the definition — experiments. Then, we compared the hypothetical with the executed experiments, checking for missing or additional ones. Furthermore, we investigated the implications of potential differences on the scalability assessment.

15.2.2. Results

We were able to define all experiments from our previous work in a single BDLT definition but had to include a few further experiments. In the following, we introduce and explain the BDLT definition and provide highlights from the scalability assessment.

```
Given the <time range>,
when varying the number of users between the minimum and the maximum
      in steps of 50
and varying the RAM among (0.5GB, 1GB)
and varying the CPU shares among (0.25, 0.5)
and varying the number of instances among (1, 2, 4),
then run each experiment for 30 minutes
and collect the Domain–based metric.
```

Listing 15.5: BDLT definition: scalability assessment of microservice deployment alternatives.

Table 15.2.: Microservice Deployment Alternatives and Corresponding Domain-based Metric Values (Avritzer et al., 2020a)

RAM	CPU	#instances	Domain-based metric
0.5 GB	0.25	1	0.6150
1 GB	***0.25***	***1***	***0.7763***
1 GB	0.5	1	0.5356
0.5 GB	0.5	1	0.5154
0.5 GB	0.5	2	0.5100
1 GB	0.25	2	0.7408
1 GB	0.5	2	0.5340
0.5 GB	0.5	4	0.5053
1 GB	0.25	4	0.3716
1 GB	0.5	4	0.5672

15.2.2.1. BDLT Definition

Listing 15.5 provides the BDLT definition comprising all experiments. It defines using a workload from a specific time range in the *given* clause. As we did not define the time range in our previous work, we leave it open here, too. The *when* clause defines the different levels of numbers of users in steps of 50. Given the minimum and maximum are 50 and 300, it results in the same load tests as in the experiments. However, it is more flexible, as our approach determines the boundaries automatically. Alternatively, we could have set the boundaries explicitly to 50 and 300. Furthermore, the *when* clause defines the variations of the configurations RAM, CPU shares, and number of instances. In the *then* clause, we defined the run time and specified to collect the Domain-based metric, which is calculated based on the number of requests and the response times of the requests.

For transforming the BDLT definition into a BenchFlow test, the generated LCtL would hold the time range as a `timerange` clause and an `intensity-range` aggregation extracting the different intensity levels. The BenchFlow test would have the `exhaustive_exploration` goal with CPU, RAM, and number of instances parameters as defined above. The load function would be `constant` and vary between the different user levels in subsequent experiments. The Domain-based metric — which we needed to pre-configure in the `data_collection` section — would be collected using the `observe` keyword.

As the BDLT definition defines six different intensities, two RAM configurations, two CPU configurations, and three different numbers of service instances, it results in 72 experiments. These are 12 more than we executed originally. The reason is that we did not test the configurations (0.5 GB, 0.25 CPU shares, 2 instances) and (0.5 GB, 0.25 CPU shares, 4 instances) because we knew they would result in poor performance. However, the BDLT language does not allow for such non-trivial combinations of configuration parameters. Therefore, we had to include them in the definition. As a consequence, the experiment execution lasts six hours longer than the required 30 hours (not considering SUT deployment, ramp-up, and cooldown).

15.2.2.2. Scalability Assessment

The Domain-based metric values per deployment alternative resulting from the scalability assessment are contained in Table 15.2. Also, we show the values per workload situation for selected deployments in Figure 15.5. Surprisingly, the generally best alternative has relatively few resources with 1 GB of RAM, 0.25 CPU shares, and one instance of the *carts* service. Using the same configuration with more *carts* instances results in a lower Domain-based metric value. At least, the deployment with the most resources performed best for low workloads. This finding indicates that careful scalability assessment is strongly preferable compared to adding more resources. In a poorly designed application, more resources can even reduce performance under high workloads, e.g., due to synchronization effects.

Transforming the BDLT definition into experiments and executing them would have led us to the same result. As they only comprise two additional deployment alternatives, which we knew to perform worse, the optimal deployment would remain the same. However, the experiments would have lasted 20 % longer (36 instead of 30 hours).

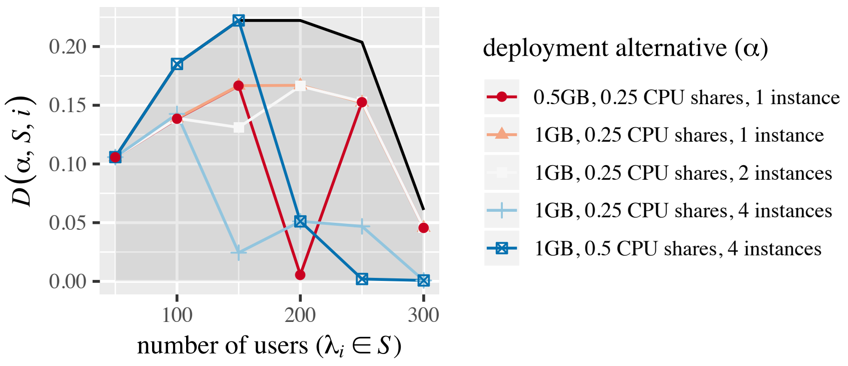


Figure 15.5.: Domain-based metric $D(\alpha, S, i)$ (Avritzer et al., 2020a).

15.3. Discussion of Research Questions

In this section, we discuss the sub-questions of RQ4, which we defined in Section 5.1, considering the results of the two case studies.

15.3.1. RQ4.1 — Expressiveness for Industrial Concerns

How expressive is the BDLT language in regards to load test concerns of industrial use cases?

Using the BDLT language, we were able to express all load test concerns our industrial partner named. In doing so, we covered a broad scope of available features, such as steady-state and spike workloads, past and expected future workload, what-if analyses (recovery spike test), qualitative forecasts added on top (more devices test), specific and varying system configurations, and different pass/fail criteria. However, we had to make use of the extension mechanism for custom events. For predicting recovery spikes, we had to add a custom implementation calculating the *recovery severity* from the outage length. Because there is no silver bullet for custom events, such custom implementations are inevitable. We conclude that the BDLT language is suitably expressive, while many of the provided features are required.

15.3.2. RQ4.2 — Use in Industry

How would BDLT be used in industrial contexts?

The members of the DevOps team we collaborated with reported about several use cases of the BDLT language. First, they stated they would use it instead of defining load tests manually, as they currently do. Besides, they noticed that the usage of the natural language makes the test definitions easily understandable for non-experts such as product owners. Hence, also non-experts could specify BDLT definitions, or at least participate in the specification. Furthermore, a BDLT definition could be used as an acceptance criterion of a Scrum user story (Schwaber and Beedle, 2001). In general, in the meetings with the DevOps team, we noticed that the BDLT definitions had stimulated a discussion, also with team members with a less technical background.

15.3.3. RQ4.3 — Benefits and Limitations

What are the benefits and limitations of using BDLT in comparison to defining load test scripts?

In general, our industrial partner found that our BDLT approach "has potential," and they were interested in further development. Primarily, they rated the use of natural language positively, as mentioned before, which especially eases discussing load testing concerns with less technically experienced team members.

The identified limitations of BDLT are the need for extensions for custom events, such as the outage recovery, and the current focus of our approach to HTTP APIs. For this reason, executing the generated tests in the context of the industrial case study requires additional implementations. One option would be to extend BenchFlow to support the used messaging protocol and add a transformation to Gatling load tests (Gatling Corp, 2020), which our industrial partner used already. Another alternative would be to implement a REST proxy that receives requests from the current implementation of our approach and forwards them as messages to the IoT endpoint.

As feedback for future improvements, our industrial partner mentioned the requirement to compare test executions, such as the BDLT definition for continuous quality assurance. Furthermore, we discussed that in some cases, the *and* keyword means a logical *or* instead. For instance, if a *then* clause contains a `break` and a `run` statement, the test should be executed for the defined duration *or* stopped if the `break` condition is fulfilled. However, we decided not to introduce new keywords but to stick to the ones already used in Behavior-driven Development (BDD). Instead, we paid attention to separating related statements such as `ensure` and `break`, which both define acceptance criteria.

15.3.4. RQ4.1 — Expressiveness for Laboratory Concerns

How expressive is the BDLT language regarding the load test concerns coming from laboratory experiments?

As the laboratory case study shows, the BDLT language can also express more complex load test concerns, such as the scalability assessment of different microservice deployment alternatives. In doing so, a single BDLT definition can comprise multiple experiments and, thus, ease selecting the optimal alternative. However, we also noticed that the language has limitations regarding non-trivial parameter combinations. While it only allows for testing all combinations of all specified parameters, our previous experiments only contained a subset. Hence, we introduced additional test overhead. Future work might investigate possibilities for specifying parameter subsets using the BDLT language. However, we also presume that natural language is limited. At the same time, BDLT definitions should be kept simple to be usable by non-experts.

15.4. Lessons Learned

In the following, we present the lessons we learned when conducting the case studies. In general, we found the template-based natural language to be well understood and well-received by practitioners. Furthermore, it was able to express more complex load test concerns. However, there are also some open challenges motivating future research in BDLT and load test definition in natural language in general.

15.4.0.1. Natural Language is Easy to Understand

In the meetings we had with our industrial partner, they understood the BDLT definitions we presented well, which we attribute to the natural language. Primarily, the team members were able to extend the definitions and express load test concerns on their own. Furthermore, they rated the language to be understandable for non-experts such as product owners, allowing utilization as acceptance criteria in Scrum user stories.

15.4.0.2. Natural Language Helps Identifying Load Testing Concerns

We also noticed that discussing load tests defined in BDLT reveals new concerns. In our meetings, the precise but well understandable load test definitions in natural language have formed a good basis for discussing the test's concern and shaping it. Also, our industrial partner came up with new concerns that arose based on BDLT definitions we already had. These discussions appeared to be beneficial in themselves, regardless of the automated generation and execution of load tests.

15.4.0.3. Special Concerns Require Extensibility

An important finding of our industrial case study is that there are specific load testing concerns that cannot be expressed through standardized language templates. An example is the outage recovery, which requires custom preprocessing for determining the recovery severity. We conclude that BDLT or related languages cannot be universal but need to be extensible. Our approach is to enable users to register extensions, which react on a particular keyword. Notably, adding such extensions requires expert knowledge. Hence, experts need to set up our BDLT approach — and most likely any related approach —, such that non-experts can use it.

15.4.0.4. Natural Language is Limited

There can be constructs such as load test parameter combinations where natural language lacks in concise descriptions. For example, it is hard to concisely describe the deployment alternatives from the scalability assessment experiments, because they are a non-trivial subset of all combinations of the parameters RAM, CPU shares, and number of instances. Hence, future works should focus on expressing such complex constructs and assessing the limitations of the natural language for load test definition. Still, the BDLT language is valuable for less experienced users, as we were able to comprise all experiments in one definition.

15.4.0.5. Scalability Assessment is Crucial

As reported by Avritzer et al. (2020a), carefully assessing the scalability of different deployment alternatives of a microservice application is inevitable and superior over adding more resources. In our experiments, the Domain-based metric shows that adding more resourced can even reduce the application's ability to handle high workloads. With the BDLT language, we ease scalability assessment also for non-experts. However, to overcome the limitations discussed above, expert users might prefer using our context-tailoring approach (Chapter 8) and BenchFlow (Ferme and Pautasso, 2018) directly, without the BDLT language as an additional level of abstraction.

15.5. Threats to Validity

In the following, we discuss the threats to the validity of our work we identified, structured by conclusion, internal, construct, and external validity. In general, we have focused on gaining experience in an industrial context rather than applying rigorous methods. Future works should extend our experience with profound measures.

15.5.1. Conclusion Validity

In the industrial case study, we derived our conclusions from oral discussions in meetings. Hence, we missed a structured interview protocol, which might have influenced our findings. However, we found that such unstructured meetings are most valuable for gaining experience from an industrial partner. The team we collaborated with could add their opinion and feedback without being asked explicitly. Due to the same reason, we did not collect quantitative measures but focused on qualitative feedback. Future work should add quantitative measures, e.g., through surveys or experiments.

15.5.2. Internal Validity

The fact that we selected the industrial case study subject from our existing contacts bears the threat that they might have responded more positively than other subjects would have. However, conducting case studies that include several personal meetings requires considerable acceptance from the subject, which can be achieved by mutual familiarity. Also, because they were intrinsically interested in the topic, we presume they rated our approach critically.

Another threat is the discussions in which we might have influenced the team. However, we aimed at a bilateral exchange such that we can learn from each other and gain as much experience as possible. Still, we cannot preclude an influence on the team. Therefore, future works should evaluate our approach in more rigorous settings.

15.5.3. Construct Validity

In the laboratory case study, we have applied the BDLT language to the experiments from our own previous work. Hence, we could have had this load testing concern in mind when designing the language, and the concern might not be representative of general complex concerns. However, the case study still shows that the language is capable of expressing laboratory concerns. Also, the BDLT language does not contain clauses we exclusively used in the laboratory case study. For evaluating the language regarding its general applicability, we presume it has to be applied to various contexts. In doing so, it might be extended by adding more clauses.

15.5.4. External Validity

As we only evaluated the BDLT language with one industrial and one laboratory subject, we cannot reason the current version of the language is generally expressible. Other subjects can have further load testing concerns. However, we have shown that template-based natural language has relevant use cases, which was our primary goal. Furthermore, the BDLT language is easily extensible by adding further clauses. Future works should apply our approach to further subjects from industrial and laboratory contexts to extend it by the required clauses to bring it into a generally usable state.

15.6. Summary

In this chapter, we evaluated the BDLT language in two case studies. We found that expressing load tests in template-based natural language entails several benefits, such as including technically less experienced team members in a discussion. The concepts from the automated generation of tailored load tests, as presented in this thesis, and the text execution automation by Ferme and Pautasso (2018) abstract from technical details.

This chapter concludes the evaluation of our approach. For an overview of all conducted studies, we refer to Chapter 11. In the next chapter, we discuss existing works that are related to ours.

Chapter 16

Related Work

In this chapter, we discuss existing work that is related to ours. We structure it according to the research plan we introduced in Section 5.3. Section 16.1 focuses on work that relates to our whole approach or multiple work packages. Section 16.2 discusses the related work of WP1, i.e., the automated parameterization of load tests. In Section 16.3, we collect work that is mainly related to the service-tailoring of WP2. Section 16.4 belongs to WP3, which targets the context-tailoring of load tests. Finally, in Section 16.5, we focus on Behavior-driven Load Testing (BDLT) for non-experts, as defined by WP4. Section 16.6 summarizes the chapter.

16.1. Overall Approach

In this dissertation, we introduce an approach to the automated generation of tailored load tests. We automate the parameterization of load tests required for fully automated load test generation. Also, we tailor the load tests to specific (micro-) services, to reduce the required hardware, and to relevant contexts, e.g., an upcoming sales event. Based on these contributions, our BDLT language for non-experts integrates automated load test generation and execution leveraging BenchFlow (Ferme and Pautasso, 2018).

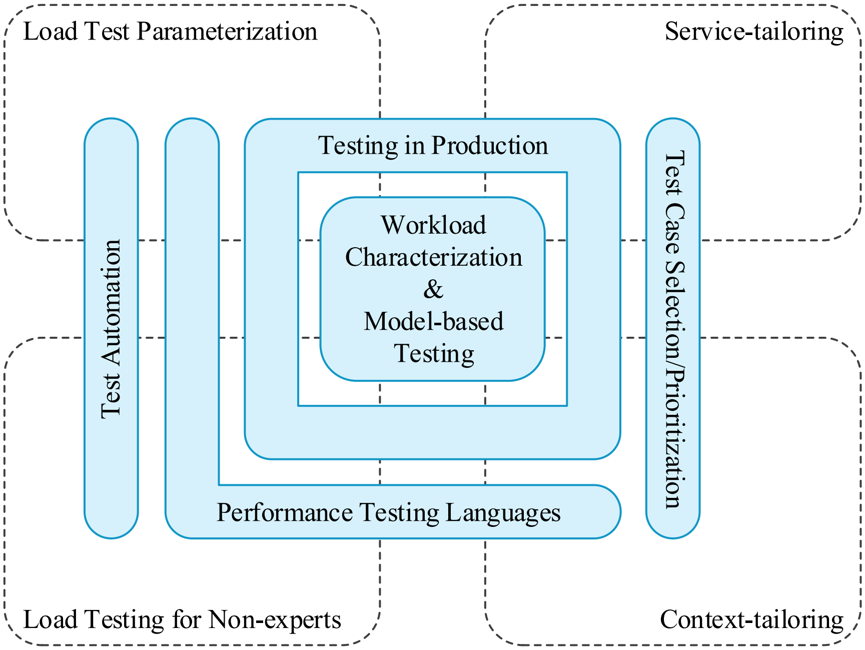

Load Test Parameterization
Service-tailoring
Testing in Production
Workload Characterization & Model-based Testing
Test Automation
Test Case Selection/Prioritization
Performance Testing Languages
Load Testing for Non-experts
Context-tailoring

Figure 16.1.: Overview of work related to parts of our approach.

As illustrated in Figure 16.1, numerous research fields are related to our overall approach or several work packages. In the following, we discuss these fields. Two essential techniques we base upon in all work packages are workload characterization and model-based testing (Sections 16.1.1 and 16.1.2). An alternative approach to representative load testing is testing in production (Section 16.1.3). Related to service- and context-tailoring are general test case selection or prioritization approaches (Section 16.1.4). The automation of (load) tests (Section 16.1.5) is a goal we also target with the automated load test parameterization and the BDLT approach. Finally, we propose several languages to be used for load or performance testing (Section 16.1.6). Besides, there is related work specific to each work package, which we discuss in Sections 16.2 to 16.5.

16.1.1. Workload Characterization

Deriving a representative load test requires characterizing the workload. For instance, Ferrari (1972) emphasizes that careful characterization is mandatory for reasonable performance evaluation. The literature contains many works that contribute to this field, including summarizing methodologies by Calzarossa et al. (2016), Calzarossa and Serazzi (1993), Feitelson (2002), Menascé and Almeida (2002), and Menascé et al. (1999). These authors agree on two fundamental steps: measuring the basic workload units and extracting a workload model, e.g., by clustering the basic units. The derived model should represent the expected workload, which is also referred to as an operational profile (Musa, 1993).

In this research, we focus on session-based workloads, i.e., the basic units are user sessions consisting of requests to the system under test (SUT). Z. Li and Tian (2003) have assessed Markov chains as a suitable modeling formalism for this type of workload, and several approaches leverage it (Barros et al., 2007; Menascé, 2002; Menascé et al., 1999; Ruffo et al., 2004; van Hoorn et al., 2008; Vögele et al., 2018). An often-cited formalism is the Customer Behavior Model Graph by Menascé et al. (1999), which uses the endpoints of the SUT as states connected with transition probabilities. Related are probabilistic timed automata (Abbors et al., 2013a,b, 2014), which differentiate between probabilistic and timed transitions. A drawback of Markov-chain-based approaches is that they are not able to model so-called inter-request dependencies. These dependencies denote requests that are only valid after another request was made, e.g., a logout from a website has to be preceded by a login. Addressing this limitation, Krishnamurthy et al. (2006) and Shams et al. (2006) handle inter-request dependencies using sequences of requests and extended infinite state machines but lack the probabilistic modeling capabilities of Markov chains. A further formalism is the stochastic form-oriented workload model (Cai et al., 2007; Draheim et al., 2006; Lutteroth and Weber, 2008), which merges probabilistic transitions with reactions to the SUT's responses. Finally, Zhou et al. (2014a,b) use a formalism named sequential action model.

We highlight the work by van Hoorn et al. (2008) and Vögele et al. (2018), which merges the probabilistic Markov chains and inter-request dependency handling. The introduced WESSBAS approach extracts workload models from recorded sessions while automatically finding so-called guards and actions describing the inter-request dependencies. In our work, we leverage the WESSBAS approach and extend its model extraction algorithm to be incremental and specific to microservices. Notably, we leave the integration with guards and actions for future work.

When modeling a session-based workload, it is relevant to capture the session's behavior and quantify it. The quantity is referred to as workload intensity (Menascé and Almeida, 2002) and defines the number of concurrently active sessions or the session arrival rate of a closed or open workload, respectively (Schroeder et al., 2007). Several studies characterize real workloads (Arlitt and T. Jin, 2000; Goseva-Popstojanova et al., 2006; Menascé et al., 2003), with the conclusion that the grouping of requests into sessions is crucial. von Kistowski et al. (2014b, 2017) propose the Descartes Load Intensity Model (DLIM) that can express arrival rates decomposed into trends, seasonalities, bursts, and noise with the LIMBO (von Kistowski et al., 2014a) tool support. Also, DLIM instances can be extracted from recorded logs. In our research, we incrementally learn the number of concurrent sessions per group and represent it as a time series. This format is suitable for forecasting and, therefore, we would not benefit from extracting DLIM models from the time series. However, the extraction could be used for integrating our incremental learning with the tooling by von Kistowski et al.

16.1.2. Model-based Testing

Model-based testing (Utting and Legeard, 2007) aims at improving testing practices by leveraging models of the SUT. Abbors et al. (2013a) discuss the differences between system models and test models, which describe the SUT's behavior from different perspectives: the former focus on the SUT's internals while the latter see the SUT as a black box from a user's or environment's viewpoint. According to this terminology, the workload models we use for load testing are test models, and representative load testing is a specific form of model-based testing. While we have discussed the most related works in Section 16.1.1, we refer to Jiang and Hassan (2015) for a comprehensive overview of (model-based) load testing approaches.

Apart from load testing, the applicability of model-based testing is manifold. Dias Neto et al. (2007) and Javed et al. (2016) provide literature surveys of existing approaches that differ in multiple dimensions. The first is the size of the tested software part, i.e., at the unit (Kamma and Maruthi, 2014; Padgham et al., 2013; Usman et al., 2020), service (Bertolino et al., 2008b), integration (Basanieri et al., 2002), and system level (Menascé et al., 1999; Shams et al., 2006; Vögele et al., 2018). Second, works focus on different quality characteristics, such as functional correctness (Basanieri et al., 2002; Mahali and Mohapatra, 2018), performance (Abbors et al., 2013b; Shams et al., 2006; Usman et al., 2020; Vögele et al., 2018), reliability (Avritzer and Weyuker, 2009), and security (Botella et al., 2019; Felderer et al., 2016; Peroli et al., 2018). Finally, the type of models used varies. In addition to the models listed in Section 16.1.1, Unified Modeling Language (UML) diagrams are frequently used, including class (Basanieri et al., 2002), activity (Ahmad et al., 2019), and sequence diagrams (Mahali and Mohapatra, 2018). In our work, we use Markov-chain-based models for testing both functional and non-functional attributes, with a particular focus on performance, at the system, integration, and service levels.

16.1.3. Testing in Production

With the increasing adoption of continuous integration and delivery (CI/CD) (Humble and Farley, 2010) and cloud technologies, a trend in applying software testing in the production environment has started. Schermann et al. (2018) summarize different applied strategies when deploying new versions of a software or service. These strategies have in common that both the new and old versions are deployed simultaneously. Canary releasing (Humble and Farley, 2010) redirects a small portion of the traffic to the new version, to test whether it behaves as intended. With A/B testing (Kohavi et al., 2013), both versions are compared regarding functional and non-functional properties. Gradual rollouts (Humble and Farley, 2010) start with a small portion of traffic directed to the new version, which gradually increases. Furthermore, there are sophisticated approaches, e.g., for shifting user traffic between datacenters (Veeraraghavan et al., 2016) or executing specific experiments (Schermann et al., 2016). These testing strategies allow investigating the tested system's behavior under the real load and, therefore, can be seen as an alternative to representative load testing. However, as Feitelson (2002) argues, the redirection of live traffic has several drawbacks compared to workload modeling. As the live traffic is arbitrary, it cannot be generalized to the overall workload. Also, the comparability of multiple testings, e.g., for regression detection, is not ensured. For these reasons, we learn workload models from live traffic and apply testing in a dedicated environment.

Further approaches apply testing after the release. For instance, Pietrantuono et al. (2018, 2020) test the reliability of microservice applications in their operational phase. For that, they use a combination of monitoring, adaptive sampling-based testing, and estimation. Gerostathopoulos et al. (2016) systematically conduct experiments with a distributed system based on declarative specifications. While consistently repeated testing might overcome some limitations described above, it still can only test with the current workload. Hence, it is not suitable for preparing for a rarely-occurring workload-influencing event, such as a yearly special sale. Our approach, in contrast, leverages the long-term recorded workload to cover such scenarios.

A finally worth mentioning production testing technique is *chaos engineering* (Basiri et al., 2016). Related to resilience benchmarking (Vieira et al., 2012), but performed in the production environment, it randomly conducts experiments with the application *"to build confidence in its capability to withstand turbulent conditions"* (Basiri et al., 2016). A core concept of these experiments is the injection of faults (Natella et al., 2016), e.g., by shutting down service instances or introducing software bugs, with the support of various tools, such as Simian Army (Izrailevsky and Tseitlin, 2011), Gremlin (Heorhiadi et al., 2016), Chaos Monkey (Chang et al., 2015), Screwdriver (Nagarajan and Vaddadi, 2016), and *Chaos Toolkit* (2020). For investigating the resilience with respect to performance, Keck et al. (2016) propose injecting performance problems based on antipatterns. Fault injection is also used in test environments, e.g., Meinke and Nycander (2015) use it combined with incremental learning to test microservice applications. To improve the low efficiency of chaos engineering, which stems from its randomness, van Hoorn et al. (2018) investigate the more systematic execution of resilience experiments in a test environment. Here, we can contribute by supplying representative workload models to be used in the experiments.

16.1.4. Test Case Selection and Prioritization

Test case selection is a long-standing discipline for reducing the testing time and effort. As summarized in different surveys (Biswas et al., 2011; Engström et al., 2010, 2008; Kazmi et al., 2017), many approaches focus on functional testing while using various methods, including model-based testing (see Section 16.1.2). In our research, we focus more on non-functional properties, such as performance. Therefore, we discuss related work in this domain in the following.

Much work has been done for selecting or prioritizing performance regression tests. For instance, Huang et al. (2014) apply prioritization based on static code analysis, which they name performance risk analysis. Mostafa et al. (2017) extend this technique with profiling while specializing in collection-intensive software. The approaches by de Oliveira et al. (2017) and Alcocer et al. (2020) utilize static analysis and prior test executions for selecting or prioritizing tests. Related to that, Reichelt and Kühne (2018) and Reichelt et al. (2019) leverage functional unit tests for performance testing and apply static code analysis and comparison of invocation traces to select the tests to execute. The main difference between these approaches and ours is the tested piece of software. While they focus on the code, i.e., unit level, we test the software at higher levels with a representative workload.

At this level, the size of the space of workloads that could be executed is almost infinitely large. Therefore, several works aim at generating test workloads that fulfill given objectives. Avritzer and Weyuker (1995a,b) model telecommunication systems as large Markov chains and select only the most likely occurring states for load testing. Other approaches focus on fault-inducing or resource-saturating workloads. J. Zhang and Cheung (2002) use symbolic execution to generate test suites that maximize performance measures. The method by Penta et al. (2007) is based on genetic algorithms that create workloads violating service-level agreements (SLAs). Briand et al. (2005) follow a similar goal, but for real-time systems with deadlines instead of SLAs.

A large number of works utilize feedback from test executions to reach their goal iteratively, using different techniques to improve the tests in each iteration. Barna et al. (2011a,b) use two-layer queuing models to identify performance bottlenecks. Having the same goal, the FOREPOST approach by Grechanik et al. (2012) and Q. Luo et al. (2017) applies a machine learning classification algorithm. P. Zhang et al. (2011a,b) utilize Petri nets to stress multimedia applications as highly as possible. The approach by Bayan and Cangussu (2006, 2008) contains a feedback controller that drives an embedded system to a defined utilization of one or several resources. Works specifically focusing on cloud applications are by Segall and Tzoref-Brill (2015), who use a combinatorial test design and feedback from cloud monitoring, and Gambi et al. (2013), who test the elasticity with a model-based approach.

While the listed approaches are designed to efficiently identify performance or other issues, they do not take into account the likelihood of the occurrence of the executed workloads. In contrast, our load test tailoring concept allows generating those workloads the application likely needs to handle in the future. The work by Vögele (2018) closes the gap between these objectives. He uses representative workload models extracted by the WESSBAS approach (Vögele et al., 2018) and a Palladio Component Model (Becker et al., 2009) of the SUT. Then, he applies an evolutionary multi-objective optimization process to derive workload specifications that satisfy the given objectives. The objectives can include, among others, resource utilization, response time, number of test cases, and workload representativeness. Here, our tailoring approach can deliver a set of likely occurring workloads modeled with the WESSBAS-DSL, of which, e.g., the most resource-demanding ones are selected. Furthermore, our service-tailoring can choose only those tests that are relevant for a specific microservice.

16.1.5. Test Automation

For automating the whole load testing process, several steps need to be considered, including the workload characterization (see Section 16.1.1), selection of workloads to be executed (see Section 16.1.4), deployment of the SUT, workload execution, and analysis of the test results (Jiang and Hassan, 2015). While our approach mainly contributes to the first two steps, the remaining ones are also required for, e.g., integrating the load testing into a CI/CD pipeline.

In particular, our approach needs to interact with works that execute the generated load tests automatedly. Several proposed approaches leverage the automatable capabilities of cloud environments. Astyrakakis et al. (2019) automate the deployment of the SUT to validate it, e.g., using stress tests. Barve et al. (2018) propose a model-driven approach to automate performance evaluation tasks, including profiling and testing. Most relevant for us is the BenchFlow framework by Ferme and Pautasso (2017, 2018), which automates the end-to-end process of executing performance tests based on inputs formulated in a domain-specific language (DSL). Hence, it can deliver the test results for the load tests we generate without manual intervention. For that, Palenga (2018) has implemented a transformation of WESSBAS (Vögele et al., 2018) workload models to BenchFlow DSL instances, which we have used to develop the BDLT approach (see Chapter 9).

Load tests tend to need much time to execute (T.-H. Chen et al., 2017). Therefore, it is beneficial to reduce the execution time to a minimum. To this end, Alghamdi et al. (2020, 2016) introduce approaches that stop a load test automatically as soon as the measurements are reliable.

When a load test finished its execution, the results need to be analyzed. For automating this final step, we refer to the work by Jiang et al. (2008, 2009) and Malik et al. (2010a,b, 2013, 2010c). Another approach comes from Okanović et al. (2019), who generate load test reports tailored to the user's concern. Their concept complements well our BDLT approach, to automate the end-to-end testing process starting with a tailored workload characterization and ending with a concern-tailored report.

16.1.6. Performance Testing Languages

In this work, we have developed three DSLs for different aspects of the load testing process. The Input Data and Properties Annotation (IDPA) parameterizes generated load tests for being executed. Using the Load Test Context-tailoring Language (LCtL), users can describe workload scenarios based on their context. Finally, the BDLT language reuses LCtL concepts more conveniently and integrates load test generation with execution. The literature contains further DSLs relevant for performance testing.

Related to the IDPA are workload modeling languages, which we discuss in Section 16.2.1. One of these is the BenchFlow DSL by Ferme and Pautasso (2017, 2018), which allows specifying, apart from the workload, the input data and details for the test execution. Therefore, we transform a BDLT description into an LCtL instance and a BenchFlow DSL instance to automate the test generation and execution process (Chapter 9). Also, we transform IDPA instances into the BenchFlow input data (Section 6.5.3.3). Hence, all DSLs complement each other.

Furthermore, there are approaches related to BenchFlow, which specifically focus on Cloud environments. Michael et al. (2017) test the cloud infrastructure according to SLAs negotiated between the provider and tenant. The Crawl language by Cunha et al. (2013) describes performance tests in Infrastructure-as-a-Service (IaaS) clouds. Scheuner et al. (2015, 2014) specify cloud benchmarks based on the Infrastructure-as-Code principle.

Primarily related to the BDLT language is the Gherkin language (Wynne et al., 2017) used in Behavior-driven Development (BDD) (North, 2006). We adopted the *given–when–then* structure. While initially designed for functional unit testing, the concept is also used for other testing purposes, such as safety analysis (Y. Wang and Wagner, 2018) and acceptance testing

(Rahman and J. Gao, 2015). Bernardino et al. (2016) introduce Canopus, which is a DSL for modeling performance tests. In some parts, it also uses the BDD structure. Further natural languages are proposed by Okanović et al. (2020, 2019), for generating load test reports tailored to a user concern and describing load tests to a chatbot interface.

To the best of our knowledge, there is no other DSL that describes a workload scenario on the same level of abstraction as the LCtL and BDLT language do. In contrast to the description based on the context, existing DSLs require the definition of the workload model directly.

16.2. Load Test Parameterization

We describe our contribution to the automated load test parameterization in Chapter 6. We propose the IDPA that separates manually created input data specifications and adjustments of static properties from generated workload models and load tests. Parameterized and, thus, executable load tests can be generated from the workload model and an IDPA without manual intervention. Furthermore, IDPAs can be evolved over changing APIs with little manual effort, which is only due at development time. Hence, the IDPA allows generating and executing load tests in CI/CD pipelines. In the following, we discuss works related to our approach. We group them into input data and properties in existing workload models, continuous validation of generated load tests, generation of test data, change propagation, model-driven engineering techniques, and commercial approaches.

16.2.1. Input Data and Properties in Workload Models

Many of the existing workload models for load testing comprise input data the load test should use when submitting requests. By nature, these modes contain the static properties of the IDPA `Overrides`, too. However, all workload models lack different aspects regarding the automated evolution of parameterizations.

Several proposed approaches model input data as intrinsic elements of the workload model or the request generation process. The main drawback of such approaches is the lack of separation of the automatically generated user behavior model and manual parameterizations like the input data. If the user behavior needs to be changed, the whole workload model and the finally executed load test have to be generated again, overwriting the parameterizations. As one such approach, Ruffo et al. (2004) append a query string and a POST body to an HTTP request when submitting it. The query string consists of a list of name-value pairs that are to be specified by an analyst before executing a load test. Similarly, Krishnamurthy et al. (2006) store information about name-value pairs in system-specific URL formation rules and generate query strings dynamically. Based on this, Shams et al. (2006) use extended finite state machines (EFSMs) for handling data dependencies explicitly. That is, if a request requires parameter values contained in the response of a different request, the EFSM ensures appropriate request orders. Scientific and open-source load testing frameworks such as BenchFlow (Ferme and Pautasso, 2018) and Taurus (BlazeMeter, 2015) allow modeling input data as a part of the load test definition. Even though these definitions are independent of the load test executor, the input data are not separated from the user behavior model.

The WESSBAS workload model (van Hoorn et al., 2008; Vögele et al., 2018) stores input data in a dedicated application model as possible parameter values. Also, the approach extracts the parameter values from the input request logs. However, as load tests are typically executed in dedicated test environments with databases differing from the production environment, the extracted input data are unlikely usable for the test and, thus, have to be adjusted manually. Besides, the workload model does not support data dependencies, which have to be configured in the finally generated load test (Vögele et al., 2018). In contrast to the presented approaches, our IDPA separates the manual parameterizations from automatically generated models. By its extensible design, all required input data can be specified, avoiding manual changes to the generated load tests. Hence, workload models can be regenerated automatically without overwriting manual specifications.

Opposed to intrinsic input data modeling, other approaches separate the input data from the user behavior. The stochastic form-oriented models by Draheim et al. (2006) and Lutteroth and Weber (2008) comprise a separated data model that is used for both defining the parameter values and determining the next state based on the previous request's response. Hence, the data model is specific to the stochastic form-oriented model. Furthermore, the authors do not provide a means for generating workload models from monitoring data. In contrast to this, the IDPA is independent of the used workload model and is explicitly designed for workload model generation. Zhou et al. (2014b) introduce a workload parameter specification language that separates run-time specifications from the modeled user behavior. Similar to the stochastic form-oriented model, the approach is not designed for the usage in combination with various workload models. Generating workload models is left for future work, as well. Finally, the authors do not provide an approach for evolving the run-time specifications over API changes, which we do for the IDPA.

16.2.2. Continuous Validation of Generated Load Tests

In order to limit the manual effort required for evolving generated parameterized load tests over workload changes, approaches to continuous representativeness validation have been proposed. Such approaches determine whether a modeled workload differs from the actual workload in the production environment and whether corresponding load tests need to be updated. T.-H. Chen et al. (2017) and Syer et al. (2017) validate the representativeness of load tests by comparing test and production logs. They identify execution events that explain the workload differences, which can be used to update the load tests manually. The authors claim to apply their approach regularly (e.g., every few months). Hence, manual effort is still entailed regularly each time the workloads differ sufficiently. In contrast to that, the IDPA allows generating and parameterizing new load tests automatically without manual intervention. Hence, we reduce the manual effort, and the generated load tests can be better integrated into CI/CD pipelines.

16.2.3. Test Data Generation

Another field related to load test parameterization is the automatic generation of test data, i.e., parameter values the load driver should use for the requests submitted. Howden (1975) and Ince (1987) propose some of the first approaches focusing on functional tests. Later, approaches explicitly targeting performance or load tests have been proposed as well. Barros et al. (2007) and Farahbod and Dadashi (2017) introduce related approaches that generate test data based on the composition and the relationships of the production data. Because production data typically cannot be used in test environments due to privacy regulations, the authors apply obfuscation techniques, e.g., data sanitization (Barros et al., 2007). Grechanik et al. (2010) introduce a similar approach, which anonymizes a production database based on the k-anonymity algorithm, aiming to test database-centric applications under strong privacy regulations. As an alternative to obfuscation or anonymization techniques, Bainbridge et al. (2009) propose to create test databases from scratch. However, their creation process is application-specific and cannot be generalized. The approaches differ from our IDPA approach, as we do not investigate the generation of test data, but the specification of them in a load test. Hence, the test data generation and IDPA approaches complement each other.

16.2.4. Change Propagation

In the literature, several approaches to change propagation are proposed, which are related to the IDPA evolution. The primary objective of change propagation is to resolve inconsistencies introduced by changes to certain software entities (e.g., classes, methods, or models), respecting the inconsistencies these secondary changes cause (Gwizdala et al., 2003; Rajlich, 1997). A related technique is change impact analysis, which solely detects or predicts the impact of a change (Alam et al., 2015). Selected approaches to change propagation rely on graph rewriting techniques using snapshots (Rajlich, 1997), incremental and interactive resolution of inconsistencies (Gwizdala et al., 2003), heuristics (Hassan and Holt, 2004), and machine-learning techniques using dependency networks (Lee and Hong, 2018). Applied to the IDPA evolution, the primary changes are made to the application model, introducing inconsistencies in the annotation model. These inconsistencies have to be resolved by secondary manual changes. The main difference to change propagation techniques is the lack of inconsistencies introduced by the secondary changes. Because changes to the annotation model cannot introduce further inconsistencies, we do not need to rely on such complex change propagation approaches.

16.2.5. Model-Driven Engineering Techniques

In the field of model-driven software engineering (MDSE) (Brambilla et al., 2017; Stahl et al., 2006; Steinberg et al., 2008), several approaches are related to ours. Similar to the context of the IDPA, a typical MDSE technique is the generation of program code from models. The generated code is often enriched by manually created code. The similarity to the IDPA approach is the merge of generated and manually created models (in this case, program code). In MDSE, it is recommended to separate generated and non-generated code. Similarly, our approach separates parameterizations in the form of IDPAs from generated workload models.

Fritzsche and Gilani (2012) propose another approach that is analogous to ours. While our approach enriches load tests generated from workload models with IDPAs, they use model annotations to enrich the transformation of development models to resulting models.

Finally, the evolution of workload models, API models, and the IDPA can be seen as a particular case of evolving parallel models, also known as co-evolution (Getir et al., 2018; Kramer et al., 2013; Milovanovic and Milicev, 2015). Approaches to co-evolution take care of adapting associated models as soon as one of them changes. Also, approaches utilizing the change propagation mentioned above exist (Demuth et al., 2016). However, the IDPA evolution is less complex than model co-evolution in general, because only changes in the API model require updating the other models. Besides, the workload model can always be easily updated by generating a new one.

16.2.6. Commercial Approaches

Apart from scientific approaches, several commercial load testing tools are used in practice, e.g., LoadRunner (Micro Focus, 2020[a]) and Silk Performer (Micro Focus, 2020[b]). Such tools provide a means for recording load scripts and assisting the specification of IDPA-related parameterizations. Often, it is possible to correlate IDs that are to be extracted from responses and placed into parameters of subsequent requests. However, these tools do not support the generation of load tests based on production monitoring data and, thus, do not need to evolve once created load scripts over changes in the production workload.

With particular regards to integration into CI/CD pipelines, *continuous load testing arose* (BlazeMeter, 2017; Dunne, 2018; Tricentis, 2020). Like us, such approaches aim to integrate load testing in CI/CD pipelines by reducing the test overhead and adding automation. However, representative load testing based on generated workload models is rarely applied in practice, and commercial tools do not natively support it. Hence, they do not support the evolution of load tests over workload changes, either.

16.3. Service-tailoring

Our service-tailoring approach (see Chapter 7) modifies the standard load test generation process, such that the generated tests directly target the (micro-) services under test. For that, we introduce two alternative algorithms that change specific intermediate artifacts of the generation process: *log-based* tailoring operates on the input request logs, while *model-based* tailoring modifies the extracted workload model.

We structure the related work according to the test pyramid by Cohn (2009). He suggests executing few long-running system-level tests, some more service-level tests, and a high number of unit tests. Related work on the system level we discussed in the previous Sections 16.1.1 and 16.1.2. For performing the service-tailoring, we use techniques similar to workload transformation for performance prediction, which we explain in the following.

Integration testing is in between of system-level and service-level testing and can be performed top-down or bottom-up (Myers, 2004). With our approach, we support bottom-up testing by simulating the requests of consuming services. Complementing this, we discuss service stubs, which mimic the behavior of provider services for top-down testing. Besides, we elaborate on further microservice testing approaches. Finally, we focus on the lowest level, namely non-functional unit tests.

16.3.1. Workload Transformation for Performance Prediction

From a certain point of view, our service-tailoring approach tackles the challenge of transforming a workload specification from the system level to the service level. The field of software performance modeling and prediction faces similar challenges, as it needs to transform workload specifications between different levels and representations, e.g., user-oriented and resource-oriented descriptions (Graf, 1987). As surveyed by Balsamo et al. (2004), different approaches exist that transform user-oriented models such as UML into analytical models to be solved for performance prediction, including queuing networks, Petri nets, stochastic processes, and also Markov chains. An example is a work by Cortellessa and Mirandola (2000, 2002), which transforms UML models such as activity diagrams into queuing networks. Bernardo et al. (2011) derive queuing networks from architectural descriptions in the Æmilia language. The main difference between our work and the listed ones is that ours does not include user-oriented models, but only operates on analytical models such as Markov chains. Furthermore, the listed approaches focus on the system-level while we tailor to specific services.

A subset of performance prediction approaches that focus on smaller units of an application is those for component-based systems, where it is essential to model inter-component dependencies (Becker et al., 2004), related to call dependencies between services. Denning and Buzen (1978) formalize the general problem in the *Forced Flow Law*, which states that the throughput X_i of a particular component or resource i with visit count V_i originates from the throughput X of the entire system as $X_i = X \cdot V_i$.

The Palladio Component Model (Becker et al., 2009) is one approach for component-based performance prediction. For transforming instances of this model into analytical models, Koziolek (2008) has introduced the *Dependency Solver*, which calculates the resource usage originating from inter-component dependencies and a workload specification. Utilizing the Dependency Solver, transformations to stochastic regular expressions (Koziolek, 2008), layered queuing networks (Koziolek and Reussner, 2008), and queueing Petri nets (Meier et al., 2011) exist. Ciancone et al. (2014) propose an intermediate language to be transformed into Markov chains. While these works utilize similar mechanisms as we do — using a system-level workload description and inter-component or -service dependencies for calculating the workload arriving at a specific component or service —, they base upon different models. While we utilize individually measured traces, they rely on aggregate dependency models. Also, the domain of usage is different: we generate load tests to be executed, while they derive performance measures by model solving or simulation.

Some modeling approaches, such as the previously mentioned Dependency Solver and works by Bondarev et al. (2005) and Eismann et al. (2018), put particular emphasis on capturing input-parameter dependencies. As the behavior of a component can differ for different input parameters, they consider the parameter values for determining the workload arriving at a particular component. Remarkably, we do not consider parameter values, but rather focus on the traces as instances of individual calls propagating through the system.

16.3.2. Service Stubs for Load Testing

In terms of integration testing, our service-tailoring approach generates *drivers* that can be used for bottom-up integration. The opposite approach — top-down integration — requires *stubs* that replace the services the tested ones call. There are several works in this field. A bunch of approaches addresses the early or ongoing development phase, where parts of the developed application are not available yet. Denaro et al. (2004) provide stubs for missing components, which they tailor to the specific test case. Becker et al. (2008) generate so-called performance prototypes from performance models, which can be executed similarly to a real component. Baltas and Field (2012) and Field et al. (2018) create mock objects based on performance models and tackle the synchronization of the models' virtual time with the performance tests' real time. Besides, they integrate their mock objects with performance unit tests (Chatley et al., 2019), i.e., operate at a lower level of the testing pyramid than we do. Due to the focus on early development phases, these approaches provide stubs with limited accuracy for the real implementation. As we focus on a later phase, approaches that learn the stubs' performance and functional behavior from production data would be preferable. Remarkably, the approach by Becker et al. uses the Palladio Component Model, which can be learned from production measurements (e.g., Brunnert et al., 2013; Mazkatli et al., 2020).

Another set of approaches originates from the service-oriented architecture (SOA) field, where multiple testbeds have been introduced. Typically, testbeds are created based on high-level models or other descriptions and contain both performance tests and stubs for services that are not available, e.g., external services. The stub definition ranges from high-level architectural descriptions (Grundy et al., 2005), over scripts — such as the Genesis2 approach by Juszczyk and Dustdar (2010) —, and models — including work by Grundy et al. (2006) and the SOABench approach by Bianculli et al. (2010) — to service contracts. The PUPPET approach by Bertolino et al. (2008a) falls into the latter category, which integrates SLAs for non-functional behavior (Bertolino et al., 2007) and contracts for functional behavior (Bertolino et al., 2008b) into the stubs. While the listed approaches focus on the service-level, which is the level we focus on, too, they still lack support for learning stub behavior from production measurements.

To this end, approaches that emulate individual services or even whole systems, which then can be called by other services, are relevant. Hine et al. (2009) emulate enterprise software based on deterministic finite state machines with a particular focus on scalability, which makes the approach interesting for load testing. Yu et al. (2017) follow a similar approach using

Petri nets. An approach that integrates production measurements is the one by Cui et al. (2006), which captures and replays the client- and server-side of sessions. Versteeg et al. (2016) emulate services by learning their behavior from message traces with an approach more elaborate than capture and replay. Besides, their approach is integrated into the commercial CA Service Virtualization tool (Michelsen and English, 2013). Hence, we presume it to be production-ready, making it a perfect complement to our service-tailoring approach. In combination, both approaches can completely isolate one or several services in a load test by learning the behavior of the calling services or users and the called services from production measurements.

16.3.3. Testing of Microservices

Many testing approaches not explicitly designed for microservices can be used in this context. For instance, SOA testbeds, which we describe in Section 16.3.2, are suited for testing distributed applications. Besides, some approaches explicitly target microservice applications. In Section 16.1.3, we discuss testing in production, which assumes microservices in most cases, e.g., regarding the deployment strategies (Humble and Farley, 2010) or resilience experimentation (Basiri et al., 2016). We also refer to the test automation frameworks (see Section 16.1.5): the approaches by de Camargo et al. (2016), Ferme and Pautasso (2018), and Portillo-Dominguez et al. (2014) support the testing of microservices applications.

Further related approaches originate from the functional testing domain. Lehvä et al. (2019) introduce microservice testing based on contracts between consumer and producer services. Hence, consumers can mock their providers, while providers can replay the requests of the consumers. Similarly, we replay the requests of consumers but rely on monitored data instead of contracts. G. Luo et al. (2019) tackle the challenge of verifying whether a test succeeded, which they state to be particularly challenging for microservice systems, using metamorphic testing.

Regarding the testing of non-functional properties, such as performance, microservices bear opportunities and challenges compared to monolithic systems. Heinrich et al. (2017) argue that the neat collaboration of operations can improve testing practices by using monitoring data from production. At the same time, they highlight the unfeasibility of long-lasting performance tests, which requires careful selection. Eismann et al. (2020) further highlight that performance metrics, such as response times measured during a test, can be less stable than with monoliths. In our research, we leverage production monitoring data, as suggested by Heinrich et al., to generated tailored load tests. Hence, we also contribute to the selection of relevant test cases.

A specific approach related to ours is the one by Grambow et al. (2020). The authors propose benchmarking microservices that expose a Representational State Transfer (REST) API. The benchmark workload is described in abstract patterns by users. In contrast, in our work, we generate workload models based on recorded data. In joint work with Avritzer et al. (2018, 2020a), we introduce the Domain-based scalability assessment described in Section 9.3. Here, we utilize the observed production workload to generate a series of load tests, e.g., using our approach. The Production and Performance Testing Based Application Monitoring (PPTAM) tool (Avritzer et al., 2019) is a prototypical implementation of the approach. Janes and Russo (2019) introduce PPTAM+ for supporting the transition from monolithic applications to microservices.

16.3.4. Non-functional Unit Tests

Most of the work regarding non-functional unit tests focus on performance. Also, there is extensive tool support, such as by Java Microbenchmarking Harness (JMH) (Oracle Corporation, 2020), Caliper (Google, 2015), ContiPerf (Carro, 2019), or JUnitPerf (noconnor, 2017). However, the adoption of performance unit testing is low. Stefan et al. (2017) found that only 3.4% of all open-source projects apply it. The authors attribute this finding to existing challenges concerning test automation, implementation, and execution duration.

To this end, several works aim at solving some of these challenges. For instance, Bulej et al. (2017, 2012) introduce the Stochastic Performance Logic (SPL), which allows expressing performance requirements mathematically. Based on such expressions, their proposed framework executes performance unit tests automatically and evaluates the results. Horký et al. (2013) use SPL to generate performance documentation, aiming at increasing the awareness for performance. Reichelt et al. (2019) leverage existing functional unit tests to conduct performance testing automatically. In doing so, they lower the execution time by test case selection (Reichelt and Kühne, 2018). Hill et al. (2009) are not limited to performance, but apply unit testing to applications in their early development stages regarding general non-functional properties. Also, there exist performance unit test approaches for specific domains, such as mobile applications (Kim et al., 2009; Usman et al., 2020).

The main difference between all unit testing approaches and ours is the granularity of the tested piece of software. While we focus on specific services, unit tests address smaller units that can be part of a service. As a result, unit tests tend to be faster regarding their execution, while our approach evaluates the performance and further quality attributes under more realistic conditions.

16.4. Context-tailoring

In Chapter 8, we describe our approach to the context-tailored generation of load tests. We continuously learn a workload model by clustering the sessions incrementally and enriching them with contextual information. The result is behavior models for multiple user groups with an intensity — i.e., the number of concurrent sessions per time unit — time series each. A user of our approach can trigger the generation of a tailored load test by submitting an LCtL instance. We then use multivariate time series forecasting to predict the expected workload.

Related work can be found in various fields, such as workload characterization, model-based testing, testing in production, test case selection or prioritization, and performance testing languages (see Sections 16.1.1

to 16.1.4 and 16.1.6). In this section, we elaborate on incremental clustering and workload forecasting, specifically relevant to the context-tailoring.

16.4.1. Incremental Session Clustering

Existing workload characterization approaches cluster the user sessions with algorithms such as k-means (Menascé et al., 1999) or X-means (Vögele et al., 2018). However, these algorithms can only process a single dataset as a whole, while our continuous workload model learning requires repeated updates. Therefore, we have extended the k-means-based clustering to be incremental and have designed a further algorithm based on DBSCAN (Schubert et al., 2017). T. Wang et al. (2014) introduce a similar k-means-based algorithm to be used for workload-aware anomaly detection. Like us, they assign new sessions to existing clusters based on the minimum distance and detect new clusters based on a distance threshold. However, they recalculate the threshold in each iteration and require all clusters to have the same radius while we maintain per-cluster radiuses. We suggest comparing both algorithms experimentally in future work.

Regarding general-purpose clustering, which can also be applied to sessions, there are many algorithms for incremental (Joshi and Kulkarni, 2012) or stream (Silva et al., 2013) processing. To provide some examples, IncrementalDBSCAN by Ester et al. (1996) is an often-cited one that incrementally updates the clustering. Regarding k-means, Lloyd's Algorithm (Lloyd, 1982) could be executed online, which Ailon et al. (2009), however, have found to perform poorly. Therefore, they propose an improved algorithm based on k-means++ by Arthur and Vassilvitskii (2007). Related to that are the works by Ackermann et al. (2012) and Shindler et al. (2011). However, most of the proposed algorithms miss some of our requirements, e.g., they might change the cluster membership of an already processed session or miss newly occurring clusters. The DBSCAN-based algorithms additionally suffer from the missing convexness, as we discuss in Section 8.4.1. Still, as much effort has been investigated already, these algorithms might be considered for improving ours.

16.4.2. Workload Forecasting

Menascé and Almeida (2002) argue that workload forecasting is an essential part of workload characterization. They propose predicting the expected workload intensity via basic techniques such as linear or non-linear regression, moving average, and exponential smoothing. Besides, there are more complex forecasting techniques, including the use of autoregressive integrated moving average models, support vector machines, and artificial neural networks, as surveyed by Mahalakshmi et al. (2016). Abstracting from such technical details, Bauer et al. (2020) and Taylor and Letham (2018) introduce the tools Telescope and Prophet, which chose an appropriate forecasting method autonomously. A common use case of workload forecasting is the proactive provisioning of cloud resources (Herbst et al., 2013). According to the forecast, hardware resources can be added before the workload intensity increases. Our use case is the load test generation. While we leverage Telescope and Prophet, the main difference is that we also predict the workload mix, while the tools are limited to the intensity. As our evaluation shows, this results in more representative load tests.

A technique with a related but different goal is the prediction of user behavior. For instance, Nguyen and Cho (2020) predict online users' next actions to support, e.g., recommendation and personalization systems. A similar goal has resource planning for business processes (Verenich et al., 2019). As opposed to these approaches, we predict the overall workload of the application. Still unpublished but more related work is by Albertetti and Ghorbel (2020), which predicts the (human) workloads of business processes using process mining and recurrent neural networks. However, it only aims at mid-term predictions, while we also require long-term horizons.

16.5. Load Testing for Non-experts

To ease load testing for unexperienced users, our BDLT approach integrates the tailored load test generation with BenchFlow (Ferme and Pautasso, 2018), as depicted in Chapter 9. The BDLT language acts as the interface and allows describing the workload, runtime configurations, including SUT deployment options, and quality gates. We also integrate the scalability assessment approach we have developed in joint work with Avritzer et al. (2018, 2020a).

Several research fields we discussed in previous sections are related to the BDLT approach (see Sections 16.1.1 to 16.1.3, 16.1.5, and 16.1.6). Mainly related to this part of our work are BDD, whose concept we leverage, and declarative performance engineering (DPE). In the following, we elaborate on these fields.

16.5.1. Behavior-driven Development

BDD (North, 2006) is an extension of test-driven development (TDD) (Beck, 2003) and aims at specifying the intended behavior of software before implementing it. This behavior is defined in the Gherkin (Wynne et al., 2017) language and automatically transformed into software tests. Gherkin consists of natural language templates starting with the keywords *given*, *when*, and *then* to describe the initial state, changes to the state, and expected outcomes. BDD has been widely adopted for different use cases, such as microservice testing (Rahman and J. Gao, 2015) and safety analysis (Y. Wang and Wagner, 2018). An approach that is related to BDLT is Canopus by Bernardino et al. (2016), which uses BDD syntax to describe performance tests. Oruç and Ovatman (2016) transform BDD statements into JMeter (Apache Software Foundation, 2020[a]) load tests. The main difference between these works and ours is the level of abstraction. While they have to specify the workload model directly, we can rely on our tailoring approach and refer to the context and services to be tested.

While we have not focused on the specification process of BDD definitions, works investigating this direction exist. Soeken et al. (2012) allow users to specify BDD definitions in a dialog with the computer. N. Gao and Z. Li (2016) provide a web tool for collecting requirements models from multiple stakeholders, transforming them into BDD user scenarios. The case study we have conducted (Chapter 15) indicates that BDD — or in this case, BDLT — is easy to understand for non-technical stakeholders, such as product owners. Sarinho (2019) builds on the same insight and uses BDD in game-based teaching.

16.5.2. Declarative Performance Engineering

DPE (Walter et al., 2016) aims at hiding complexity from the user when conducting performance engineering tasks, such as model-based prediction, monitoring, or testing. It separates a user's concern, i.e., *what* question they want to answer, from the selection of a specific solution approach, i.e., *how* to solve the task. For that, DPE requires a high-level description language, e.g., Descartes Query Language (DQL) by Gorsler et al. (2014), and decision support regarding the applied performance engineering approach. We have previously discussed several works that contribute to DPE, such as BenchFlow by Ferme and Pautasso (2017, 2018), the Crawl language by Cunha et al. (2013), and the concern-driven reporting by Okanović et al. (2019). Walter et al. (2018) collect further tools to be used in this context.

Our BDLT approach conforms to DPE as well. We mainly achieved that by integrating BenchFlow for declarative test execution. Besides, we add a high-level and, thus, declarative description of the workload. As users only need to describe the context and services under test, we hide complexity from them and automatically choose the appropriate workload model.

16.6. Summary

Our research relates to existing works in many aspects: it bases upon existing workload characterization and model-based testing techniques, complements test automation, test data generation, and service stubbing approaches, uses concepts from model-driven software engineering (MDSE), and conforms to principles from declarative performance engineering (DPE). The next part concludes this dissertation with a summary and suggestions for future work.

Part IV

Conclusion

Chapter 17 Summary

In this dissertation, we have developed an approach for the automated generation of tailored representative load tests. The automation enables the integration with continuous software engineering (CSE) concepts, such as continuous integration and delivery (CI/CD) pipelines (Humble and Farley, 2010). We can also generate load tests that focus on the relevant services and context, being time- and resource-efficient. For instance, the DevOps team developing a specific microservice of a webshop can test its service with the expected Christmas shopping workload in early December, while putting less emphasis on the Black Friday workload. Finally, we have integrated our tailoring approach with automated test execution, easing load testing for non-experts.

In the following, we summarize our work and draw conclusions according to the research questions we have postulated in Chapter 1. Furthermore, we summarize the implementations and supplementary material we provide.

17.1. Automated Load Test Parameterization

RQ1: *How can load test parameterizations be evolved without manual intervention at test generation or execution time?*

Our solution is a user-specified parameterization model named Input Data and Properties Annotation (IDPA). Users can define IDPAs in advance, and our approach automatically parameterizes generated load tests by transforming the IDPAs. To further reduce the maintenance effort, we developed feedback-based evolution mechanisms regarding the API changes collected in the literature. Hence, we reduce the manual effort and prevent manual intervention when generating or executing load tests.

Our evaluation, which comprises effort estimation models, experimental studies, and a case study, shows that the IDPA reduces the cumulative maintenance effort from a quadratic to a linear function of time. Furthermore, we found that proper parameterization is inevitable for reasonable load testing. Unparameterized load tests may fail to, e.g., login to the system under test (SUT) and, thus, lead to significantly different performance behavior of the SUT. An IDPA can prevent this, providing extensible parameterization concepts ready for industrial use. The benefit of using an IDPA is the highest if the sessions dominate the workload, i.e., the order and timing of requests influence the SUT more than the choice of parameter value from the correct input data set does. Otherwise, the IDPA can still be used but requires more careful modeling of the input data.

17.2. Service-tailoring

RQ2: *How can representative load tests be tailored to specific services of a session-based application?*

By analyzing the load test extraction process used in existing work (Vögele et al., 2018), we identified two possible modifications that tailor load tests to services. In both cases, we used recorded traces documenting how a request propagates through the distributed application (Okanović et al., 2016). The *log-based service-tailoring* algorithm modifies the recorded logs, i.e., operates on the traces at the instance level. The *model-based service-tailoring* algorithm changes the workload model based on the aggregate call behavior, which it extracts from the traces.

Both algorithms provenly fulfill the requirements we have defined. An experimental study shows that the model-based algorithm generates load tests that are slightly more representative than log-based generated tests. However, the difference decreases with an increasing number of SUT endpoints involved, and both algorithms lead to significantly higher representativeness than simple request-based tailoring does. The model-based approach is preferable for integration with context-tailoring (see next section) because it can tailor load tests to services specified on demand. Log-based tailoring requires defining the services in advance.

17.3. Context-tailoring

> RQ3: *How can representative load tests automatically be tailored to the contexts of a session-based workload?*

We split the load test generation process into a continually repeated part and a part executed on demand. In the first part, we cluster the user sessions incrementally, resulting in a continually updated workload model and workload intensity. Here, we introduced two algorithms, whereas we chose to use the one based on k-means (Arthur and Vassilvitskii, 2007). Furthermore, we enrich the workload model with contextual information, e.g., the presence of special sales or the temperature. The second part generates a load test on-demand by predicting future intensities using time-series forecasting (Bauer et al., 2020; Taylor and Letham, 2018) and extracting a workload scenario from the prediction. For that, users describe the scenario in the Load Test Context-tailoring Language (LCtL). For instance, they can describe the expected spike workload during the next sales event, potentially modifying the prediction with qualitative information, such as the effect of extraordinarily high temperatures.

Our evaluation — consisting of an analysis of the incrementally learned workload model, a case study, and two experiment series with the student information system (SIS) of Charles University, Prague — indicates that this approach is suitable for generating context-tailored load tests. Except for particular influences, the load tests we generated were representative, and the LCtL could specify the relevant scenarios accurately. The influences, however, reveal challenges to be addressed in future work. First, we found strongly fluctuating think time specifications in the workload model stemming from the session clustering, which ignores the session timings, similar to existing work (Vögele et al., 2018). The fluctuations affected the duration of the simulated sessions and, thus, decreased representativeness. Furthermore, the predictions of sharp spikes were inaccurate due to the inability of the forecasting tools used to predict the time series shape. Apart from the representativeness, the workload prediction duration can be improved.

17.4. Load Testing for Non-experts

> RQ4: *How can we leverage automated tailored load test generation and automated load test execution for enabling load testing for non-experts?*

We enable load testing for non-experts by integrating load test generation and execution with the Behavior-driven Load Testing (BDLT) language. The language uses the *given–when–then* template from Behavior-driven Development (BDD) (North, 2006) and is based on natural language. Hence, it constitutes a lower barrier for technically inexperienced users. Showing that the language still can be used for complex load testing concerns, we integrated the Domain-based microservice scalability assessment from joint work with Avritzer et al. (2018, 2020a).

The two case studies we conducted show that the BDLT language is expressive enough for industrial use cases. Similar to the IDPA, extensibility was a key feature. The case study participants particularly highlighted that natural language is easy to understand and fosters collaboration also with non-technical stakeholders, such as product owners. An insight we gained is that the explicit specification of load test concerns in natural language forms a helpful basis for discussions that lead to further relevant concerns. A limitation of natural language is complex expressions, which can hinder the sharp definition of advanced load testing concerns. Here, we suggest using the LCtL as an alternative, which is more technical and better suited for advanced specifications.

17.5. Implementations & Supplementary Material

We have implemented our approach as a prototypical distributed application, which we have used in the evaluations. The source code is available online on GitHub and archived for permanent access (H. Schulz, 2020a; H. Schulz et al., 2020a; H. Schulz and Dang, 2020), with container images being available on DockerHub (H. Schulz, 2020d) for easy deployment. Besides, we made additional implementations, such as an SUT mock and a trace converter service, publicly available (H. Schulz, 2020b,c).

For replication of our evaluations, we published the following supplementary material:

- Evaluation of automated load test parameterization: H. Schulz et al. (2019f)
- Evaluation of service-tailoring: H. Schulz et al. (2019b)
- Evaluation of context-tailoring: H. Schulz et al. (2020b)
- Evaluation of load testing for non-experts: H. Schulz et al. (2019d)

Chapter 18

Future Work

There are several directions we suggest investigating in future work, which we describe in this chapter. These directions comprise improvements and enhancements of the automation of the load test parameterization, think time modeling, workload model learning, and workload forecasting. Besides, we propose exploring further use cases of our approach and conducting additional evaluations.

18.1. Extended Automation of Parameterization

In Chapter 6, we have introduced the automated parameterization of generated load tests with Input Data and Properties Annotations (IDPAs) and the evolution of parameterizations over API changes. Hence, we have reached our goal of automating the load test generation process. However, the degree of automation when maintaining IDPAs can be further increased. For instance, OpenAPI specifications (OpenAPI Initiative, 2020) can also hold information about the allowed parameter inputs and return type of an endpoint. This information could be leveraged to decide whether API changes of the type *Change Input* or *Change Response Behavior* are present. For that, the IDPA application model would need to be extended.

Furthermore, the IDPA could be extended by further types of parameterization, e.g., sanity checks. While the BenchFlow DSL (Ferme and Pautasso, 2018) and our Behavior-driven Load Testing (BDLT) language (see Chapter 9) already cover quality gates that are evaluated after the test execution, e.g., based on performance measures, IDPAs could be used to define endpoint-level checks. These checks would focus on functional requirements, e.g., whether the response to a request contains specific fields.

Finally, we propose modeling endpoint types other than HTTP in the IDPA application model. For instance, message queuing protocols, such as Advanced Message Queuing Protocol (AMQP) and Kafka, are often used in continuous software engineering (CSE) projects. The application model offers an extension point for new endpoint types; thus, they can be added easily.

18.2. Different Think Time Specification

In this research, we utilized the WESSBAS-DSL (Vögele et al., 2018) for modeling workloads, with normal distributions as think time specifications. We have chosen normal distributions because existing work uses them as well. However, in mostly all parts of our approach, we encountered think-time-related issues: the experiments with the automated parameterization (Chapter 12) showed too low request rates; the model-based service tailoring algorithm (Section 7.5) has to approximate think time convolutions; the incremental session clustering (Section 8.4) has the same problem; the evaluation of the context-tailoring (Chapter 14) suffered from negative think time portions.

Therefore, we suggest using a different type of think time specification, such as empirical distributions related to the resource demand specification of the Palladio Component Model (Becker et al., 2009). Empirical distributions allow for more accurate merging than normal distributions do and also prevent negative portions. Adding this specification type does not require to change the WESSBAS-DSL, as its think time specification is designed to be extended.

18.3. Improved Workload Model Learning

The most relevant direction for future work is improving the continuous workload model learning (Section 8.4). Similar to existing work (Vögele et al., 2018), we calculate the distance between sessions only based on the Markovian request order, ignoring the inter-request think times. When applying one-time clustering, this proceeding is sufficient. However, incremental updates of the workload model can lead to strongly fluctuating session durations (see Section 14.2). As a solution, we suggest integrating the think times into the clustering — in addition to changing the type of specification — to respect the varying timing behavior of the users. A challenge is that the number of clusters can increase, which can lead to scalability issues, e.g., when forecasting the cluster's intensities individually.

For better predicting sharp workload spikes, we propose switching to an open workload model (Schroeder et al., 2007). In contrast to a closed workload model, which we utilized, it defines the session arrival rate instead of the number of concurrent users. We hypothesize that the arrival rate correlates with the request rates better than the number of users does (see Section 14.4). However, open workloads require accurate modeling of the session length and duration. As our and the evaluation of Vögele et al. (2018) show, Markov chains are not suited for that. In contrast, they tend to generate too long sessions, leading to an increased request rate. Here, using the guards and actions from the WESSBAS-DSL, which we excluded in this research, might improve the accuracy.

Finally, the incremental session clustering can be further improved. One aspect is detecting multiple new clusters in one iteration. The algorithm we introduced only can detect one. Furthermore, for better describing group-based qualitative intensity forecasts — e.g., increasing the number of students checking their exam results by 20 % —, we suggest labeling the workload model's groups accordingly. For that, interactive clustering (Bae et al., 2020) might be helpful, as an expert might be able to identify specific types of users and influence the clustering. However, the interaction must restrict to a calibration phase, without blocking continuous learning.

18.4. Improved Workload Intensity Forecasting

Regarding the forecasting of the workload intensities, we see two challenges to be addressed. First, the tools we used (Bauer et al., 2020; Taylor and Letham, 2018) were not able to predict the shape of the intensity curve accurately (see Section 14.5). Instead, the forecasts contained sharp changes at context state transitions. Hence, future work should find a way of predicting the intensity curve. One option is to adopt the tools. Another, maybe more promising option is to preprocess the context for preventing sharp context changes. For instance, the context during a sharp intensity spike could be changed from a boolean specification to a smooth in- and decrease according to the spike's slope.

The second challenge is the forecasting duration, i.e., the time the tools need to calculate a forecast. Especially for per-group intensity predictions, which require applying the tools on the groups individually, the duration was too high in our evaluation. We see two approaches to accelerating the forecasts. First, per-group predictions can be parallelized. However, for high numbers of groups, this requires large-scale machines with a high number of CPU cores. Therefore, another option is to predict specific groups together. If the intensities of some groups correlate, they could be summed up, forecasted in total, and split again afterward.

18.5. Further Use Cases

In this work, we have used our approach to generate tailored load tests automatically and, in combination with BenchFlow (Ferme and Pautasso, 2018), executed automatically. By integrating further approaches, additional use cases could be covered. First, our parameterization approach could be integrated with test data generation approaches, resulting in an even higher degree of automation. Second, the service-tailoring, combined with load-test-capable stubs, e.g., the ones by Versteeg et al. (2016), enables load testing a microservice in isolation. Further integrations could ease the specification of load tests and evaluation of their results. Here, we refer to Okanović et al.

(2020, 2019), who have introduced concern-driven reporting and a software engineering chatbot. These approaches fit particularly well with our BDLT language (Chapter 9).

Furthermore, an intelligent test case selection approach could be added on top of our approach. With the Load Test Context-tailoring Language (LCtL) (Chapter 8) and the service-tailoring algorithms (Chapter 7), we enable generating load tests that fit a given context and set of services. Hence, our approach could be used to generate load tests for all expected workload scenarios, from which only the most relevant ones that can be executed within a given time frame could be selected. Determining the relevance could be done based on the expected number of faults caused by the workload, e.g., using the work by Vögele (2018). Combining all these integrations may lead to an intelligent load testing platform, related to testing as a service (TaaS) (Yan et al., 2012).

18.6. Further Evaluation

Finally, we propose conducting further studies complementing ours by evaluating the following aspects.

- The expressiveness of the languages we have introduced (IDPA, LCtL, and BDLT) in different domains than our evaluation was.
- A quantitative assessment of the maintenance effort when using IDPAs to parameterize the effort estimation models introduced in Section 12.1. For that, a study related to the one by Benestad et al. (2010) can be appropriate.
- The representativeness of service-tailored load tests for industrial applications.
- The representativeness of load tests generated using service-tailoring combined with context-tailoring. Here, we hypothesize that the model-based service-tailoring is preferable over log-based tailoring because the call behavior between services can frequently change. However, this hypothesis should be evaluated experimentally.

- The behavior and applicability of the context-tailoring applied to a further dataset. This dataset should come from a different domain than education, and should preferably be large enough to apply forecasting without intensity augmentation.
- A quantitative evaluation of the BDLT approach, as we mostly focused on qualitative aspects.

As several aspects require large amounts of data or a realistic evaluation setting, we recommend conducting one large case study that covers several or all of them.

Bibliography

Abbors, F., T. Ahmad, D. Truscan, I. Porres (2013a). “Model-Based Performance Testing in the Cloud Using the Mbpet Tool.” In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013)*. ACM, pp. 423–424 (cit. on pp. 27, 435, 436).

– (2013b). “Model-Based Performance Testing of Web Services Using Probabilistic Timed Automata.” In: *Proceedings of the 9th International Conference on Web Information Systems and Technologies (WEBIST 2013)*. SciTePress, pp. 99–104 (cit. on pp. 27, 435, 437).

Abbors, F., D. Truscan, T. Ahmad (2014). “An Automated Approach for Creating Workload Models from Server Log Data.” In: *Proceedings of the 9th International Conference on Software Engineering and Applications (ICSOFT-EA 2014)*. SciTePress, pp. 14–25 (cit. on pp. 27, 435).

Ackermann, M. R., M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, C. Sohler (2012). “StreamKM++: A Clustering Algorithm for Data Streams.” In: *ACM Journal of Experimental Algorithmics* 17.1 (cit. on p. 455).

Aderaldo, C. M., N. C. Mendonça, C. Pahl, P. Jamshidi (2017). “Benchmark Requirements for Microservices Architecture Research.” In: *Proceedings of the 1st IEEE/ACM International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE@ICSE 2017)*. IEEE, pp. 8–13 (cit. on pp. 147, 317, 353, 421).

Ahmad, T., J. Iqbal, A. Ashraf, D. Truscan, I. Porres (2019). “Model-Based Testing Using UML Activity Diagrams: A Systematic Mapping Study.” In: *Computer Science Review* 33, pp. 98–112 (cit. on p. 437).

Ailon, N., R. Jaiswal, C. Monteleoni (2009). “Streaming K-Means Approximation.” In: *Proceedings of the 23rd Annual Conference on Neural Information Processing Systems 2009*. Curran Associates, Inc., pp. 10–18 (cit. on p. 455).

Alam, K. A., R. B. Ahmad, A. Akhunzada, M. H. N. M. Nasir, S. U. Khan (2015). “Impact Analysis and Change Propagation in Service-Oriented Enterprises: A Systematic Review.” In: *Information Systems* 54, pp. 43–73 (cit. on p. 446).

Albertetti, F., H. Ghorbel (2020). “Workload Prediction of Business Processes - an Approach Based on Process Mining and Recurrent Neural Networks.” In: *CoRR* abs/2002.11675 (cit. on p. 456).

Alcocer, J. P. S., A. Bergel, M. T. Valente (2020). “Prioritizing Versions for Performance Regression Testing: The Pharo Case.” In: *Scienceof Computer Programming* 191, p. 102415 (cit. on p. 439).

Alghamdi, H. M., C.-P. Bezemer, W. Shang, A. E. Hassan, P. Flora (2020). “Towards Reducing the Time Needed for Load Testing.” In: *Journal of Software: Evolution and Process* (cit. on pp. 42, 441).

Alghamdi, H. M., M. D. Syer, W. Shang, A. E. Hassan (2016). “An Automated Approach for Recommending When to Stop Performance Tests.” In: *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME 2016)*. IEEE Computer Society, pp. 279–289 (cit. on pp. 42, 272, 351, 441).

Angerstein, T. (2018). “Modularization of Representative Load Tests for Microservice Applications.” Master’s Thesis (cit. on pp. 64, 79, 89, 90).

Apache Software Foundation (2020[a]). *Apache JMeter*. URL: `http://jmeter.apache.org/` (visited on 07/16/2020) (cit. on pp. 6, 32, 42, 44–46, 58, 64, 73, 77, 84, 116, 127, 131, 251, 283, 291, 308, 340, 385, 457, 547).

– (2020[b]). *Apache JMeter API*. URL: `https://jmeter.apache.org/api/index.html` (visited on 07/16/2020) (cit. on pp. 45, 131).

– (2020[c]). *Log Files - Apache HTTP Server*. URL: `https://httpd.apache.org/docs/1.3/logs.html` (visited on 07/16/2020) (cit. on p. 254).

Arlitt, M., T. Jin (2000). “A Workload Characterization Study of the 1998 World Cup Web Site.” In: *IEEE Network* 14.3, pp. 30–37 (cit. on p. 436).

Arnold, K., J. Gosling, D. Holmes (2005). *The Java Programming Language*. Addison Wesley Professional (cit. on p. 248).

Arthur, D., S. Vassilvitskii (2007). “K-Means++: The Advantages of Careful Seeding.” In: *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2007)*. SIAM, pp. 1027–1035 (cit. on pp. 186, 400, 455, 463).

Astyrakakis, N., Y. Nikoloudakis, I. Kefaloukos, C. Skianis, E. Pallis, E. K. Markakis (2019). “Cloud-Native Application Validation & Stress Testing through a Framework for Auto-Cluster Deployment.” In: *Proceedings of the 24th IEEE International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD 2019)*. IEEE, pp. 1–5 (cit. on p. 441).

Avritzer, A., V. Ferme, A. Janes, B. Russo, H. Schulz, A. van Hoorn (2018). “A Quantitative Approach for the Assessment of Microservice Architecture Deployment Alternatives by Automated Performance Testing.” In: *Proceedings of the 12th European Conference on Software Architecture (ECSA 2018)*. Vol. 11048. Lecture Notes in Computer Science. Springer, pp. 159–174 (cit. on pp. VIII, 7, 20, 59, 66, 223, 239, 262, 410, 453, 457, 464).

Avritzer, A., V. Ferme, A. Janes, B. Russo, A. van Hoorn, H. Schulz, D. Menasché, V. Rufino (2020a). “Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-Based Approach Leveraging Operational Profiles and Load Tests.” In: *Journal of Systems and Software* 165, p. 110564 (cit. on pp. VII, 7, 20, 59, 66, 221–223, 239, 240, 243, 262, 411, 420, 423, 425, 429, 453, 457, 464).

– (2020b). *Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-Based Approach Leveraging Operational Profiles and Load Tests - Reproducibility Package*. Zenodo. `http://doi.org/10.5281/zenodo.3689500` (cit. on pp. 8, 560).

Avritzer, A., D. S. Menasché, V. Rufino, B. Russo, A. Janes, V. Ferme, A. van Hoorn, H. Schulz (2019). “PPTAM: Production and Performance Testing Based Application Monitoring.” In: *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE 2019)*. ACM, pp. 39–40 (cit. on pp. VIII, 66, 453).

Avritzer, A., E. J. Weyuker (1995a). “Correction: "The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software".” In: *IEEE Trans. Software Eng.* 21.11, p. 927 (cit. on p. 439).

– (1995b). “The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software.” In: *IEEE Transactions on Software Engineering* 21.9, pp. 705–716 (cit. on p. 439).

– (2009). “The Automated Generation of Test Cases Using an Extended Domain Based Reliability Model.” In: *Proceedings of the 4th International Workshop on Automation of Software Test (AST 2009)*. IEEE Computer Society, pp. 44–52 (cit. on p. 437).

Bae, J., T. Helldin, M. Riveiro, S. Nowaczyk, M.-R. Bouguelia, G. Falkman (2020). "Interactive Clustering: A Comprehensive Review." In: *ACM Computing Surveys (CSUR)* 53.1, 1:1–1:39 (cit. on pp. 193, 469).

Bainbridge, D., I. H. Witten, S. J. Boddie, J. Thompson (2009). "Stress-Testing General Purpose Digital Library Software." In: *Proceedings of the 13th European Conference on Research and Advanced Technology for Digital Libraries (ECDL 2009)*. Vol. 5714. Lecture Notes in Computer Science. Springer, pp. 203–214 (cit. on pp. 44, 446).

Balsamo, S., A. Di Marco, P. Inverardi, M. Simeoni (2004). "Model-Based Performance Prediction in Software Development: A Survey." In: *IEEE Transactions on Software Engineering* 30.5, pp. 295–310 (cit. on p. 449).

Balsamo, S., A. Marin (2007). "Queueing Networks." In: *Proceedings of the 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM 2007)*. Vol. 4486. Lecture Notes in Computer Science. Springer, pp. 34–82 (cit. on p. 24).

Baltas, N., T. Field (2012). "Continuous Performance Testing in Virtual Time." In: *Proceedings of the 9th International Conference on Quantitative Evaluation of Systems (QEST 2012)*. IEEE Computer Society, pp. 13–22 (cit. on pp. 148, 172, 350, 451).

Barford, P., M. Crovella (1998). "Generating Representative Web Workloads for Network and Server Performance Evaluation." In: *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1998 / PERFORMANCE 1998)*. ACM, pp. 151–160 (cit. on p. 146).

Barna, C., M. Litoiu, H. Ghanbari (2011a). "Autonomic Load-Testing Framework." In: *Proceedings of the 8th International Conference on Autonomic Computing (ICAC 2011)*. ACM, pp. 91–100 (cit. on p. 440).

– (2011b). "Model-Based Performance Testing." In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*. Ed. by R. N. Taylor, H. C. Gall, N. Medvidovic. ACM, pp. 872–875 (cit. on p. 440).

Barros, M. D., J. Shiau, C. Shang, K. Gidewall, H. Shi, J. Forsmann (2007). "Web Services Wind Tunnel: On Performance Testing Large-Scale Stateful Web Services." In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007)*. IEEE Computer Society, pp. 612–617 (cit. on pp. 2, 44, 56, 67, 146, 175, 314, 435, 446).

Barve, Y. D., S. Shekhar, S. Khare, A. Bhattacharjee, A. S. Gokhale (2018). "UPSARA: A Model-Driven Approach for Performance Analysis of Cloud-Hosted Applications."

In: *Proceedings of the 11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2018)*. IEEE Computer Society, pp. 1–10 (cit. on p. 441).

Basanieri, F., A. Bertolino, E. Marchetti (2002). “The Cow_Suite Approach to Planning and Deriving Test Suites in Uml Projects.” In: *Proceedings of the 5th International Conference on the Unified Modeling Language (UML 2002)*. Vol. 2460. Lecture Notes in Computer Science. Springer, pp. 383–397 (cit. on p. 437).

Basiri, A., N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, C. Rosenthal (2016). “Chaos Engineering.” In: *IEEE Software* 33.3, pp. 35–41 (cit. on pp. 20, 438, 452).

Bass, L. J., I. M. Weber, L. Zhu (2015). *DevOps - A Software Architect’s Perspective*. SEI Series in Software Engineering. Addison-Wesley (cit. on pp. 2, 14, 17, 50, 145, 171, 412).

Bauer, A., M. Züfle, N. Herbst, S. Kounev, V. Curtef (2020). “Telescope: An Automatic Feature Extraction and Transformation Approach for Time Series Forecasting on a Level-Playing Field.” In: *Proceedings of the 36th International Conference on Data Engineering (ICDE 2020)* (cit. on pp. 35–37, 63, 176, 214, 251, 392, 393, 407, 456, 463, 470).

Bayan, M. S., J. W. Cangussu (2006). “Automatic Stress and Load Testing for Embedded Systems.” In: *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC 2006)*. IEEE Computer Society, pp. 229–233 (cit. on p. 440).

– (2008). “Automatic Feedback, Control-Based, Stress and Load Testing.” In: *Proceedings of the 23rd ACM Symposium on Applied Computing (SAC 2008)*. ACM, pp. 661–666 (cit. on p. 440).

Beck, K. L. (2003). *Test-Driven Development - by Example*. The Addison-Wesley Signature Series. Addison-Wesley (cit. on pp. 16, 226, 457).

Becker, S., T. Dencker, J. Happe (2008). “Model-Driven Generation of Performance Prototypes.” In: *Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW 2008)*. Vol. 5119. Lecture Notes in Computer Science. Springer, pp. 79–98 (cit. on pp. 148, 172, 350, 451).

Becker, S., L. Grunske, R. Mirandola, S. Overhage (2004). “Performance Prediction of Component-Based Systems - a Survey from an Engineering Perspective.” In: *Architecting Systems with Trustworthy Components, Revised Selected Papers from the International Seminar, Dagstuhl Castle*. Vol. 3938. Lecture Notes in Computer Science. Springer, pp. 169–192 (cit. on p. 449).

Becker, S., H. Koziolek, R. H. Reussner (2009). “The Palladio Component Model for Model-Driven Performance Prediction.” In: *Journal of Systems and Software* 82.1, pp. 3–22 (cit. on pp. 440, 450, 468).

Benestad, H. C., B. Anda, E. Arisholm (2010). “Understanding Cost Drivers of Software Evolution: A Quantitative and Qualitative Investigation of Change Effort in Two Evolving Software Systems.” In: *Empirical Software Engineering* 15.2, pp. 166–203 (cit. on pp. 275, 471).

Ben-Kiki, O., C. Evans, I. döt Net (2009). *YAML Ain't Markup Language (YAML™) Version 1.2*. URL: `http://yaml.org/spec/1.2/spec.html` (visited on 07/16/2020) (cit. on p. 523).

Bernardino, M., A. F. Zorzo, E. d. M. Rodrigues (2016). “Canopus: A Domain-Specific Language for Modeling Performance Testing.” In: *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST 2016)*. IEEE Computer Society, pp. 157–167 (cit. on pp. 443, 457).

Bernardo, M., V. Cortellessa, M. Flamminj (2011). “TwoEagles: A Model Transformation Tool from Architectural Descriptions to Queueing Networks.” In: *Proceedings of the 8th European Performance Engineering Workshop (EPEW 2011)*. Vol. 6977. Lecture Notes in Computer Science. Springer, pp. 265–279 (cit. on p. 449).

Bertolino, A., G. D. Angelis, F. Lonetti, A. Sabetta (2008a). “Let the Puppets Move! Automated Testbed Generation for Service-Oriented Mobile Applications.” In: *Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2008)*. IEEE Computer Society, pp. 321–328 (cit. on p. 451).

Bertolino, A., G. De Angelis, L. Frantzen, A. Polini (2008b). “Model-Based Generation of Testbeds for Web Services.” In: *Testing of Software and Communicating Systems: Proceedings of the 20th IFIP TC 6/WG 6.1 International Conference on Testing Communicating Systems (TestCom 2008) and the 8th International Workshop on Formal Approaches to Testing of Software (FATES 2008)*. Vol. 5047. Lecture Notes in Computer Science. Springer, pp. 266–282 (cit. on pp. 437, 451).

Bertolino, A., G. De Angelis, A. Polini (2007). “A QoS Test-Bed Generator for Web Services.” In: *Proceedings of the 7th International Conference on Web Engineering (ICWE 2007)*. Vol. 4607. Lecture Notes in Computer Science. Springer, pp. 17–31 (cit. on p. 451).

Bezemer, C.-P., S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villavicencio, J. Walter, F. Willnecker (2019). “How Is Performance Addressed in DevOps?” In: *Proceedings of the 10th ACM/SPEC International*

Conference on Performance Engineering (ICPE 2019). ACM, pp. 45–50 (cit. on pp. 2, 4, 18, 50, 221).

Bianculli, D., W. Binder, M. L. Drago (2010). “Automated Performance Assessment for Service-Oriented Middleware: A Case Study on BPEL Engines.” In: *Proceedings of the 19th International Conference on World Wide Web (WWW 2010)*. ACM, pp. 141–150 (cit. on p. 451).

Billington, J. (2014). *Ellen DeGeneres Selfie at Oscars 2014 Breaks Twitter*. URL: `https://www.news.com.au/technology/online/ellen-degeneres-selfie-at-oscars-2014-breaks-twitter/news-story/da9c8e1fe7d8e73ed791143872f10bbe` (visited on 07/16/2020) (cit. on pp. 178, 180, 200).

Biswas, S., R. Mall, M. Satpathy, S. Sukumaran (2011). “Regression Test Selection Techniques: A Survey.” In: *Informatica (Slovenia)* 35.3, pp. 289–321 (cit. on p. 439).

BlazeMeter (2015). *Taurus: A New Star in the Test Automation Tools Constellation*. URL: `https://www.blazemeter.com/blog/taurus-new-star-test-automation-tools-constellation` (visited on 11/27/2018) (cit. on pp. 42, 444).

– (2017). *How to Include Load Testing in Your Continuous Integration Environment*. URL: `https://www.blazemeter.com/blog/how-include-load-testing-your-continuous-integration-environment-0` (visited on 11/27/2018) (cit. on p. 448).

Bondarev, E., P. H. N. de With, M. R. V. Chaudron, J. Muskens (2005). “Modelling of Input-Parameter Dependency for Performance Predictions of Component-Based Embedded Systems.” In: *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA 2005)*. IEEE Computer Society, pp. 36–43 (cit. on p. 450).

Bosch, J. (2014). “Continuous Software Engineering: An Introduction.” In: *Continuous Software Engineering*. Springer, pp. 3–13 (cit. on p. 13).

Bosch, J., H. H. Olsson, J. Björk, J. Ljungblad (2013). “The Early Stage Software Startup Development Model: A Framework for Operationalizing Lean Principles in Software Startups.” In: *Proceedings of the 4th International Conference on Lean Enterprise Software and Systems (LESS 2013)*. Vol. 167. Lecture Notes in Business Information Processing. Springer, pp. 1–15 (cit. on p. 14).

Botella, J., J.-F. Capuron, F. Dadeau, E. Fourneret, B. Legeard, F. Schadle (2019). “Complementary Test Selection Criteria for Model-Based Testing of Security Components.” In: *International Journal on Software Tools for Technology Transfer* 21.4, pp. 425–448 (cit. on p. 437).

Box, G. E. P., D. R. Cox (1964). “An Analysis of Transformations.” In: *Journal of the Royal Statistical Society: Series B (Methodological)* 26.2, pp. 211–243 (cit. on p. 35).

Box, G. E. P., G. M. Jenkins, G. C. Reinsel, G. M. Ljung (2015). *Time Series Analysis: Forecasting and Control*. 5th ed. 712 pp. (cit. on p. 36).

Bradley Web Group (2016). *Does Your Website Traffic Go down at the Weekend?* URL: `https://bradleywebgroup.com/website-traffic-go-weekend/` (visited on 07/16/2020) (cit. on pp. 178–180).

Brambilla, M., J. Cabot, M. Wimmer (2017). *Model-Driven Software Engineering in Practice, Second Edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool (cit. on p. 447).

Briand, L. C., Y. Labiche, M. Shousha (2005). “Stress Testing Real-Time Systems with Genetic Algorithms.” In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*. ACM, pp. 1021–1028 (cit. on p. 440).

Broadleaf Commerce, LLC (2017). *DemoSite*. URL: `https://github.com/BroadleafCommerce/DemoSite` (visited on 07/16/2020) (cit. on pp. 71, 262, 274, 289, 290).

– (2020). *Editions and Solutions Overview*. URL: `https://www.broadleafcommerce.com/editions/overview` (visited on 07/16/2020) (cit. on p. 290).

Brunnert, A., A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. R. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, A. Koziolek, S. Spinner, J. Kroßand, C. Vögele, J. Walter, A. Wert (2015). *Performance-Oriented Devops: A Research Agenda*. SPEC-RG-2015-01 (cit. on pp. 18, 42).

Brunnert, A., C. Vögele, H. Krcmar (2013). “Automatic Performance Model Generation for Java Enterprise Edition (EE) Applications.” In: *Proceedings of the 10th European Workshop on Computer Performance Engineering (EPEW 2013)*. Vol. 8168. Lecture Notes in Computer Science. Springer, pp. 74–88 (cit. on p. 451).

Bulej, L., T. Bures, V. Horký, J. Kotrc, L. Marek, T. Trojánek, P. Tůma (2017). “Unit Testing Performance with Stochastic Performance Logic.” In: *Automated Software Engineering* 24.1, pp. 139–187 (cit. on p. 454).

Bulej, L., T. Bures, J. Keznikl, A. Koubková, A. Podzimek, P. Tuma (2012). “Capturing Performance Assumptions Using Stochastic Performance Logic.” In: *Proceedings of the 3rd Joint WOSP/SIPEW International Conference on Performance Engineering (ICPE 2012)*. ACM, pp. 311–322 (cit. on p. 454).

Cai, Y., J. Grundy, J. G. Hosking (2007). "Synthesizing Client Load Models for Performance Engineering via Web Crawling." In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. ACM, pp. 353–362 (cit. on pp. 2, 27, 67, 146, 175, 435).

Calzarossa, M. C., L. Massari, D. Tessera (2016). "Workload Characterization: A Survey Revisited." In: *ACM Computing Surveys* 48.3, 48:1–48:43 (cit. on pp. 2, 6, 16, 23, 24, 26, 159, 435).

Calzarossa, M. C., G. Serazzi (1993). "Workload Characterization: A Survey." In: *Proceedings of the IEEE* 81.8, pp. 1136–1150 (cit. on p. 435).

Camargo, A. de, I. L. Salvadori, R. d. S. Mello, F. Siqueira (2016). "An Architecture to Automate Performance Tests on Microservices." In: *Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services (iiWAS 2016)*. ACM, pp. 422–429 (cit. on p. 452).

Carro, R. L. (2019). *ContiPerf*. URL: `https://github.com/javatlacati/contiperf` (visited on 07/16/2020) (cit. on p. 453).

Chair of Software Engineering, University of Würzburg (2020). *DescartesResearch/Telescope*. GitHub. URL: `https://github.com/DescartesResearch/telescope/tree/test_multivariate` (visited on 07/16/2020) (cit. on pp. 36, 215).

Chandola, V., A. Banerjee, V. Kumar (2009). "Anomaly Detection: A Survey." In: *ACM Computing Surveys* 41.3, 15:1–15:58 (cit. on pp. 3, 50, 175).

Chang, M. A., B. Tschaen, T. Benson, L. Vanbever (2015). "Chaos Monkey: Increasing SDN Reliability through Systematic Network Destruction." In: *Computer Communication Review* 45.5, pp. 371–372 (cit. on p. 438).

Chaos Toolkit (2020). *Chaos Toolkit*. URL: `https://chaostoolkit.org/` (visited on 07/16/2020) (cit. on p. 438).

Charles University, Faculty of Mathematics and Physics (2017). *Academic Calendar 2017/2018*. URL: `https://www.mff.cuni.cz/en/students/academic-calendar/academic-calendar-2017-2018` (visited on 07/16/2020) (cit. on p. 359).

– (2018). *Academic Calendar 2018/2019*. URL: `https://www.mff.cuni.cz/en/students/academic-calendar/academic-calendar-2018-2019` (visited on 07/16/2020) (cit. on p. 359).

– (2020). *Measures Regarding the Coronavirus SARS-CoV-2 and the COVID-19 Disease*. URL: `http://mff.cuni.cz/coronavirus` (visited on 07/16/2020) (cit. on p. 378).

Chatley, R., T. Field, D. Wei (2019). “Continuous Performance Testing in Virtual Time.” In: *Companion of the IEEE International Conference on Software Architecture (ICSA Companion 2019)*. IEEE, pp. 109–115 (cit. on p. 451).

Chen, T., C. Guestrin (2016). “XGBoost: A Scalable Tree Boosting System.” In: *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD 2016)*. ACM, pp. 785–794 (cit. on p. 36).

Chen, T.-H., M. D. Syer, W. Shang, Z. M. Jiang, A. E. Hassan, M. N. Nasser, P. Flora (2017). “Analytics-Driven Load Testing: An Industrial Experience Report on Load Testing of Large-Scale Systems.” In: *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP 2017)*. IEEE, pp. 243–252 (cit. on pp. 2, 50, 67, 68, 441, 445).

Ciancone, A., M. L. Drago, A. Filieri, V. Grassi, H. Koziolek, R. Mirandola (2014). “The KlaperSuite Framework for Model-Driven Reliability Analysis of Component-Based Systems.” In: *Software and Systems Modeling* 13.4, pp. 1269–1290 (cit. on p. 450).

Cleveland, R. B., W. S. Cleveland, J. E. McRae, I. Terpenning (1990). “STL: A Seasonal-Trend Decomposition Procedure Based on Loess.” In: *Journal of Official Statistics* 6.1, pp. 3–73 (cit. on p. 35).

Cleveland, W. S., E. Grosse, W. M. Shyu (1991). “Local Regression Models.” In: *Statistical Models in S*. CRC Press, Inc., pp. 309–376 (cit. on p. 362).

Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional (cit. on p. 448).

ContinuITy (2020). *ContinuITy: Automated Performance Testing in Continuous Software Engineering*. URL: `https://continuity-project.github.io/` (visited on 07/16/2020) (cit. on pp. 65, 247).

Conway, M. E. (1968). “How Do Committees Invent.” In: *Datamation* 14.4, pp. 28–31 (cit. on p. 19).

Cortellessa, V., R. Mirandola (2000). “Deriving a Queueing Network Based Performance Model from UML Diagrams.” In: *Workshop on Software and Performance*, pp. 58–70 (cit. on p. 449).

– (2002). “PRIMA-UML: A Performance Validation Incremental Methodology on Early UML Diagrams.” In: *Sci. Comput. Program.* 44.1, pp. 101–129 (cit. on p. 449).

Cui, W., V. Paxson, N. Weaver, R. H. Katz (2006). “Protocol-Independent Adaptive Replay of Application Dialog.” In: *Proceedings of the 13th Network and Distributed System Security Symposium (NDSS 2006)*. The Internet Society (cit. on p. 452).

Cunha, M., N. C. Mendonça, A. Sampaio (2013). "A Declarative Environment for Automatic Performance Evaluation in IaaS Clouds." In: *Proceedinsg of the IEEE 6th International Conference on Cloud Computing (CLOUD 2013)*. IEEE Computer Society, pp. 285–292 (cit. on pp. 442, 458).

Davis, A. (2019). "DevOps." In: *Mastering Salesforce DevOps: A Practical Guide to Building Trust While Delivering Innovation*. Berkeley, CA: Apress, pp. 27–64 (cit. on p. 17).

Demeyer, S., B. Verhaeghe, A. Etien, N. Anquetil, S. Ducasse (2018). "Evaluating the Efficiency of Continuous Testing during Test-Driven Development." In: *Proceedings of the 2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST@SANER 2018)*. IEEE, pp. 21–25 (cit. on p. 16).

Demuth, A., M. Riedl-Ehrenleitner, R. E. Lopez-Herrejon, A. Egyed (2016). "Co-Evolution of Metamodels and Models through Consistent Change Propagation." In: *Journal of Systems and Software* 111, pp. 281–297 (cit. on p. 447).

Denaro, G., A. Polini, W. Emmerich (2004). "Early Performance Testing of Distributed Software Applications." In: *Proceedings of the 4th International Workshop on Software and Performance (WOSP 2004)*. ACM, pp. 94–103 (cit. on p. 450).

Denning, P. J., J. P. Buzen (1978). "The Operational Analysis of Queueing Network Models." In: *ACM Computing Surveys* 10.3, pp. 225–261 (cit. on p. 449).

Dias Neto, A. C., R. Subramanyan, M. Vieira, G. H. Travassos (2007). "A Survey on Model-Based Testing Approaches: A Systematic Review." In: *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech 2007) Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. WEASELTech '07. Atlanta, Georgia: Association for Computing Machinery, pp. 31–36 (cit. on p. 437).

Docker Inc. (2020). *Docker - Build, Ship, and Run Any App, Anywhere*. URL: `https://www.docker.com/` (visited on 07/16/2020) (cit. on pp. 46, 77, 239, 247, 412).

DORA, Google Cloud (2019). *Accelerate State of DevOps 2019* (cit. on p. 18).

Draheim, D., J. Grundy, J. Hosking, C. Lutteroth, G. Weber (2006). "Realistic Load Testing of Web Applications." In: *Proceedings of the 10th Conference on Software Maintenance and Reengineering (CSMR 2006)*. IEEE Computer Society, pp. 57–70 (cit. on pp. 159, 435, 445).

Dreyfus, S. E., H. L. Dreyfus (1980). *A Five-Stage Model of the Mental Activities Involved in Directed Skill Acquisition*. ORC-80-2. Operations Research Center, University of California, Berkeley (cit. on p. 374).

Dunne, K. (2018). *Continuous Load Testing: A Journey to Performance at Scale - Flood*. URL: `https://www.flood.io/blog/continuous-load-testing-a-journey-to-performance-at-scale` (visited on 07/16/2020) (cit. on p. 448).

Eismann, S., C.-P. Bezemer, W. Shang, D. Okanović, A. van Hoorn (2020). "Microservices: A Performance Tester's Dream or Nightmare?" In: *Proceedings of the 11th ACM/SPEC International Conference on Performance Engineering (ICPE 2020)*. ACM, pp. 138–149 (cit. on pp. 2, 18, 453).

Eismann, S., J. Walter, J. von Kistowski, S. Kounev (2018). "Modeling of Parametric Dependencies for Performance Prediction of Component-Based Software Systems at Run-Time." In: *Proceedings of the IEEE International Conference on Software Architecture (ICSA 2018)*. IEEE Computer Society, pp. 135–144 (cit. on pp. 18, 450).

Elasticsearch B.V. (2020). *Elasticsearch: The Official Distributed Search & Analytics Engine*. URL: `https://www.elastic.co/elasticsearch` (visited on 07/16/2020) (cit. on p. 250).

Engström, E., P. Runeson, M. Skoglund (2010). "A Systematic Review on Regression Test Selection Techniques." In: *Information and Software Technology* 52.1, pp. 14–30 (cit. on p. 439).

Engström, E., M. Skoglund, P. Runeson (2008). "Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review." In: *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM 2008)*. ACM, pp. 22–31 (cit. on p. 439).

Ester, M., H.-P. Kriegel, J. Sander, X. Xu (1996). "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise." In: *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD 1996)*. AAAI Press, pp. 226–231 (cit. on pp. 184, 455).

Faban (2020). *Faban - Helping Measure Performance*. URL: `http://faban.org/` (visited on 07/16/2020) (cit. on p. 46).

Facebook (2016). *Prophet: Automatic Forecasting Procedure*. URL: `https://github.com/facebook/prophet` (visited on 07/16/2020) (cit. on p. 36).

Farahbod, R., A. Dadashi (2017). "Data Generation for Performance Evaluation." U.S. pat. 9,613,074 B2. SAP SE (cit. on pp. 44, 56, 314, 446).

Fedorov, A. (2017). *Introducing the Random CSV Data Set Config Plugin on Jmeter | Blazemeter*. URL: `https://www.blazemeter.com/blog/introducing-the-random-csv-data-set-config-plugin-on-jmeter` (visited on 07/16/2020) (cit. on pp. 131, 547).

Feitelson, D. G. (2002). "Workload Modeling for Performance Evaluation." In: *Performance Evaluation of Complex Systems: Techniques and Tools, Performance 2002, Tutorial Lectures*. Vol. 2459. Lecture Notes in Computer Science. Springer, pp. 114–141 (cit. on pp. 3, 435, 438).

Felderer, M., P. Zech, R. Breu, M. Büchler, A. Pretschner (2016). "Model-Based Security Testing: A Taxonomy and Systematic Classification." In: *Journal of Software Testing, Verification and Reliability* 26.2, pp. 119–148 (cit. on p. 437).

Ferme, V., C. Pautasso (2017). "Towards Holistic Continuous Software Performance Assessment." In: *Companion of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2017)*. ACM, pp. 159–164 (cit. on pp. 3, 7, 45, 59, 441, 442, 458).

– (2018). "A Declarative Approach for Performance Tests Execution in Continuous Software Development Environments." In: *Proceedings of 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2018)*. ACM (cit. on pp. 3, 5–7, 32, 42, 44–46, 54, 58, 59, 61, 62, 64, 77, 116, 127, 136, 221, 222, 244, 251, 261, 265, 308, 409, 429, 431, 433, 441, 442, 444, 452, 456, 458, 468, 470, 547).

Ferrari, D. (1972). "Workload Charaterization and Selection in Computer Performance Measurement." In: *IEEE Computer* 5.4, pp. 18–24 (cit. on pp. 2, 3, 25, 29, 41, 42, 435).

Field, T., R. Chatley, D. Wei (2018). "Software Performance Testing in Virtual Time." In: *Companion of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018)*. ACM, pp. 173–174 (cit. on pp. 148, 172, 350, 451).

Fitzgerald, B., K.-J. Stol (2017). "Continuous Software Engineering: A Roadmap and Agenda." In: *Journal of Systems and Software* 123, pp. 176–189 (cit. on pp. 14, 15).

Fokaefs, M., R. Mikhaiel, N. Tsantalis, E. Stroulia, A. Lau (2011). "An Empirical Study on Web Service Evolution." In: *Proceedings of the 18th International Conference on Web Services (ICWS 2011)*. IEEE Computer Society, pp. 49–56 (cit. on pp. 100, 104, 105).

Foo, K. C., Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, P. Flora (2015). “An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments.” In: *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE 2015)*. IEEE Computer Society, pp. 159–168 (cit. on p. 56).

Fowler, M. (2006). *Continuous Integration*. URL: `https://martinfowler.com/articles/continuousIntegration.html` (visited on 07/16/2020) (cit. on p. 15).

Fowler, M., J. Highsmith (2001). “The Agile Manifesto.” In: *Software Development* 9.8, pp. 28–35 (cit. on p. 14).

Francesco, P. D., I. Malavolta, P. Lago (2017). “Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption.” In: *Proceedings of the IEEE International Conference on Software Architecture (ICSA 2017)*. IEEE, pp. 21–30 (cit. on p. 18).

Fritzsche, M., W. Gilani (2012). “Non-Intrusive Model Annotation.” U.S. pat. 8,145,468 B2. SAP SE (cit. on p. 447).

Gambi, A., A. Filieri, S. Dustdar (2013). “Iterative Test Suites Refinement for Elastic Computing Systems.” In: *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, pp. 635–638 (cit. on p. 440).

Gao, N., Z. Li (2016). “Generating Testing Codes for Behavior-Driven Development from Problem Diagrams: A Tool-Based Approach.” In: *Proceedings of the 24th IEEE International Requirements Engineering Conference (RE 2016)*. IEEE Computer Society, pp. 399–400 (cit. on p. 457).

Gao, Q., W. Wang, G. Wu, X. Li, J. Wei, H. Zhong (2013). “Migrating Load Testing to the Cloud: A Case Study.” In: *Proceedings of the 7th IEEE International Symposium on Service-Oriented System Engineering (SOSE 2013)*. IEEE Computer Society, pp. 429–434 (cit. on p. 56).

Gatling Corp (2020). *Gatling Open-Source Load Testing – for DevOps and CI/CD*. URL: `http://gatling.io/` (visited on 07/16/2020) (cit. on pp. 42, 77, 248, 300, 426).

Gerostathopoulos, I., T. Bures, S. Schmid, V. Horký, C. Prehofer, P. Tůma (2016). “Towards Systematic Live Experimentation in Software-Intensive Systems of Systems.” In: *Proceedings of the International Colloquium on Software-Intensive Systems-of-*

Systems at 10th European Conference on Software Architecture (SiSoS@ECSA 2016). ACM, 7:1–7:7 (cit. on p. 438).

Getir, S., L. Grunske, A. van Hoorn, T. Kehrer, Y. Noller, M. Tichy (2018). “Supporting Semi-Automatic Co-Evolution of Architecture and Fault Tree Models.” In: *Journal of Systems and Software* 142, pp. 115–135 (cit. on p. 447).

Godlewski, N. (2017). *Nike Site down on Black Friday, How to Place an Order*. URL: `https://www.ibtimes.com/nike-site-down-black-friday-how-place-order-2619427` (visited on 07/16/2020) (cit. on pp. 178–180, 202).

Goessner, S. (2007). *JSONPath - XPath for JSON*. URL: `https://goessner.net/articles/JsonPath/` (visited on 07/16/2020) (cit. on p. 86).

Google (2015). *Caliper*. URL: `https://github.com/google/caliper` (visited on 07/16/2020) (cit. on p. 453).

Gorsler, F., F. Brosig, S. Kounev (2014). “Performance Queries for Architecture-Level Performance Models.” In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*. ACM, pp. 99–110 (cit. on p. 458).

Goseva-Popstojanova, K., A. D. Singh, S. Mazimdar, F. Li (2006). “Empirical Characterization of Session-Based Workload and Reliability for Web Servers.” In: *Empirical Software Engineering* 11.1, pp. 71–117 (cit. on pp. 28, 436).

Graf, I. M. (1987). “Transformation between Different Levels of Workload Characterization for Capacity Planning.” In: *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1987)*. Ed. by R. B. Bunt. ACM, pp. 195–204 (cit. on pp. 146, 449).

Grambow, M., L. Meusel, E. Wittern, D. Bermbach (2020). “Benchmarking Microservice Performance: A Pattern-Based Approach.” In: *Proceedings of the 35th ACM/SIGAPP Symposium on Applied Computing (SAC 2020)*. ACM, pp. 232–241 (cit. on p. 453).

Grechanik, M., C. Csallner, C. Fu, Q. Xie (2010). “Is Data Privacy Always Good for Software Testing?” In: *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering (ISSRE 2010)*. IEEE Computer Society, pp. 368–377 (cit. on pp. 44, 56, 446).

Grechanik, M., C. Fu, Q. Xie (2012). “Automatically Finding Performance Problems with Feedback-Directed Learning Software Testing.” In: *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*. IEEE Computer Society, pp. 156–166 (cit. on p. 440).

Grinstead, C. M., J. L. Snell (2012). *Introduction to Probability*. American Mathematical Society (cit. on pp. 325, 364).

Grundy, J., Y. Cai, A. Liu (2005). “SoftArch/MTE: Generating Distributed System Test-Beds from High-Level Software Architecture Descriptions.” In: *Autom. Softw. Eng.* 12.1, pp. 5–39 (cit. on p. 451).

Grundy, J., J. Hosking, L. Li, N. Liu (2006). “Performance Engineering of Service Compositions.” In: *Proceedings of the International Workshop on Service-Oriented Software Engineering (IW-SOSE 2006)*. SOSE ’06. Shanghai, China: Association for Computing Machinery, pp. 26–32 (cit. on p. 451).

Gwizdala, S., Y. Jiang, V. Rajlich (2003). “JTracker - A Tool for Change Propagation in Java.” In: *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 2003)*. IEEE Computer Society, pp. 223–229 (cit. on p. 446).

Hassan, A. E., R. C. Holt (2004). “Predicting Change Propagation in Software Systems.” In: *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*. IEEE Computer Society, pp. 284–293 (cit. on p. 446).

Hasselbring, W. (2016). “Microservices for Scalability: Keynote Talk Abstract.” In: *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE 2016)*. ACM, pp. 133–134 (cit. on p. 20).

Hasselbring, W., G. Steinacker (2017). “Microservice Architectures for Scalability, Agility and Reliability in E-Commerce.” In: *Proceedings of the IEEE International Conference on Software Architecture Workshops (ICSAW 2017)*. IEEE, pp. 243–246 (cit. on p. 19).

Heckmann, T. (2020). *Twitter Posting*. URL: `https://twitter.com/theckman/status/1212209564084387840?s=03` (visited on 07/16/2020) (cit. on pp. 179, 180).

Heinrich, R., A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, J. Wettinger (2017). “Performance Engineering for Microservices: Research Challenges and Directions.” In: *Companion of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2017)*. ACM, pp. 223–226 (cit. on pp. 18, 453).

Heorhiadi, V., S. Rajagopalan, H. Jamjoom, M. K. Reiter, V. Sekar (2016). “Gremlin: Systematic Resilience Testing of Microservices.” In: *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems (ICDCS 2016)*. IEEE Computer Society, pp. 57–66 (cit. on p. 438).

Herbst, N. R., N. Huber, S. Kounev, E. Amrehn (2013). "Self-Adaptive Workload Classification and Forecasting for Proactive Resource Provisioning." In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013)*. ACM, pp. 187–198 (cit. on pp. 3, 20, 25, 35, 50, 175, 176, 456).

Heyman, J., C. Byström, J. Hamrén, H. Heyman (2020). *Locust - A Modern Load Testing Framework*. URL: `https://locust.io/` (visited on 07/16/2020) (cit. on p. 42).

Hidiroglu, A. (2019). "Context-Aware Load Testing in Continuous Software Engineering." Master's Thesis (cit. on pp. 65, 195).

Hildebrandt, S. (2006). *Analysis 1*. 2nd ed. Springer Berlin Heidelberg (cit. on pp. 164, 327).

Hill, J. H., H. A. Turner, J. R. Edmondson, D. C. Schmidt (2009). "Unit Testing Non-Functional Concerns of Component-Based Distributed Systems." In: *Proceedings of the 2nd International Conference on Software Testing Verification and Validation (ICST 2009)*. IEEE Computer Society, pp. 406–415 (cit. on p. 454).

Hine, C. M., J.-G. Schneider, J. Han, S. Versteeg (2009). "Scalable Emulation of Enterprise Systems." In: *Proceedings of the 20th Australian Software Engineering Conference (ASWEC 2009)*. IEEE Computer Society, pp. 142–151 (cit. on p. 451).

Horký, V., F. Haas, J. Kotrc, M. Lacina, P. Tůma (2013). "Performance Regression Unit Testing: A Case Study." In: *Proceedings of the 10th European Workshop on Computer Performance Engineering (EPEW 2013)*. Vol. 8168. Lecture Notes in Computer Science. Springer, pp. 149–163 (cit. on p. 454).

Howden, W. E. (1975). "Methodology for the Generation of Program Test Data." In: *IEEE Transactions on Computers* 24.5, pp. 554–560 (cit. on p. 446).

Huang, P., X. Ma, D. Shen, Y. Zhou (2014). "Performance Regression Testing Target Prioritization via Performance Risk Analysis." In: *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, pp. 60–71 (cit. on p. 439).

Humble, J., D. Farley (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 1st. Pearson Education (cit. on pp. 2, 15, 16, 175, 437, 438, 452, 461).

Humble, J., J. Molesky (2011). "Why Enterprises Must Adopt Devops to Enable Continuous Delivery." In: *Cutter IT Journal* 24.8 (cit. on p. 17).

Ince, D. C. (1987). "The Automatic Generation of Test Data." In: *The Computer Journal* 30.1, pp. 63–69 (cit. on p. 446).

InfluxData, I. (2020). *InfluxData (InfluxDB) | Time Series Database Monitoring & Analytics*. URL: https://www.influxdata.com/ (visited on 07/16/2020) (cit. on p. 293).

inspectIT (2020). *inspectIT*. URL: https://github.com/inspectIT/inspectIT (visited on 07/16/2020) (cit. on p. 292).

Internet Assigned Numbers Authority (IANA) (2020). *Media Types*. URL: https://www.iana.org/assignments/media-types/media-types.xhtml (visited on 07/16/2020) (cit. on p. 118).

ISO 8601: Data Elements and Interchange Formats — Information Interchange — Representation of Dates and Times (2016). Standard (cit. on pp. 89, 202).

Izrailevsky, Y., A. Tseitlin (2011). *The Netflix Simian Army*. Medium. URL: https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116 (visited on 07/16/2020) (cit. on p. 438).

Janes, A., B. Russo (2019). "Automatic Performance Monitoring and Regression Testing during the Transition from Monolith to Microservices." In: *Proceedings of the 30th IEEE International Symposium on Software Reliability Engineering Workshops (ISSRE Workshops 2019)*. IEEE, pp. 163–168 (cit. on p. 453).

Javed, H., N. M. Minhas, A. Abbas, F. M. Riaz (2016). "Model Based Testing for Web Applications: A Literature Survey Presented." In: *JSW* 11.4, pp. 347–361 (cit. on p. 437).

Jensen, K., N. Wirth (1975). *Pascal User Manual and Report, Second Edition*. Springer (cit. on pp. 199, 227).

Jiang, Z. M., A. E. Hassan (2015). "A Survey on Load Testing of Large-Scale Software Systems." In: *IEEE Transactions on Software Engineering* 41.11, pp. 1091–1118 (cit. on pp. 1, 6, 16, 39–41, 44, 50, 56, 67, 221, 378, 403, 437, 441).

Jiang, Z. M., A. E. Hassan, G. Hamann, P. Flora (2008). "Automatic Identification of Load Testing Problems." In: *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*. IEEE Computer Society, pp. 307–316 (cit. on pp. 42, 441).

– (2009). "Automated Performance Analysis of Load Tests." In: *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM 2009)*. IEEE Computer Society, pp. 125–134 (cit. on pp. 43, 441).

Jin, G., L. Song, X. Shi, J. Scherpelz, S. Lu (2012). "Understanding and Detecting Real-World Performance Bugs." In: *Proceedings of the 33rd ACM SIGPLAN Conference on*

Programming Language Design and Implementation (PLDI 2012). ACM, pp. 77–88 (cit. on pp. 2, 25, 41).

Joshi, P., P. Kulkarni (2012). “Incremental Learning: Areas and Methods-a Survey.” In: *International Journal of Data Mining & Knowledge Management Process* 2.5, p. 43 (cit. on p. 455).

Juszczyk, L., S. Dustdar (2010). “Script-Based Generation of Dynamic Testbeds for Soa.” In: *Proceedings of the 8th IEEE International Conference on Web Services (ICWS 2010)*. IEEE Computer Society, pp. 195–202 (cit. on p. 451).

Kamma, D., P. Maruthi (2014). “Effective Unit-Testing in Model-Based Software Development.” In: *Proceedings of the 9th International Workshop on Automation of Software Test (AST 2014)*. ACM, pp. 36–42 (cit. on p. 437).

Kazmi, R., D. N. A. Jawawi, R. Mohamad, I. Ghani (2017). “Effective Regression Test Case Selection: A Systematic Literature Review.” In: *ACM Computing Surveys* 50.2, 29:1–29:32 (cit. on p. 439).

Keck, P., A. van Hoorn, D. Okanović, T. Pitakrat, T. F. Düllmann (2016). “Antipattern-Based Problem Injection for Assessing Performance and Reliability Evaluation Techniques.” In: *Proceedings of the 27th IEEE International Symposium on Software Reliability Engineering Workshops (ISSRE Workshops 2016)*. IEEE Computer Society, pp. 64–70 (cit. on p. 439).

Kim, H., B. Choi, W. E. Wong (2009). “Performance Testing of Mobile Applications at the Unit Test Level.” In: *Proceedings of the 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2009)*. IEEE Computer Society, pp. 171–180 (cit. on p. 454).

Kistowski, J. von, N. R. Herbst, S. Kounev (2014a). “LIMBO: A Tool for Modeling Variable Load Intensities.” In: *ACM/SPEC International Conference on Performance Engineering, ICPE’14, Dublin, Ireland, March 22-26, 2014*. Ed. by K.-D. Lange, J. Murphy, W. Binder, J. Merseguer. ACM, pp. 225–226 (cit. on pp. 28, 31, 436).

– (2014b). “Modeling Variations in Load Intensity over Time.” In: *Proceedings of the 3rd ACM International Workshop on Large Scale Testing (LT 2014)*. ACM, pp. 1–4 (cit. on pp. 28, 176, 436).

Kistowski, J. von, N. Herbst, S. Kounev, H. Groenda, C. Stier, S. Lehrig (2017). “Modeling and Extracting Load Intensity Profiles.” In: *ACM Transactions on Autonomous and Adaptive Systems* 11.4, 23:1–23:28 (cit. on pp. 28, 436).

Kohavi, R., A. Deng, B. Frasca, T. Walker, Y. Xu, N. Pohlmann (2013). “Online Controlled Experiments at Large Scale.” In: *Proceedings of the 19th ACM SIGKDD*

International Conference on Knowledge Discovery and Data Mining (KDD 2013). ACM, pp. 1168–1176 (cit. on p. 437).

Kohavi, R., R. Longbotham (2007). “Online Experiments: Lessons Learned.” In: *IEEE Computer* 40.9, pp. 103–105 (cit. on p. 1).

Koziolek, H. (2008). “Parameter Dependencies for Reusable Performance Specifications of Software Components.” PhD thesis (cit. on pp. 146, 450).

Koziolek, H., S. Becker, J. Happe (2007). “Predicting the Performance of Component-Based Software Architectures with Different Usage Profiles.” In: *Proceedings of the 3rd International Conference on Quality of Software Architectures (QoSA 2007)*. Vol. 4880. Lecture Notes in Computer Science. Springer, pp. 145–163 (cit. on p. 23).

Koziolek, H., R. H. Reussner (2008). “A Model Transformation from the Palladio Component Model to Layered Queueing Networks.” In: *Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW 2008)*. Ed. by S. Kounev. Vol. 5119. Lecture Notes in Computer Science. Springer, pp. 58–78 (cit. on pp. 146, 450).

Kramer, M. E., E. Burger, M. Langhammer (2013). “View-Centric Engineering with Synchronized Heterogeneous Models.” In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO 2014)*. ACM, pp. 1–6 (cit. on p. 447).

Krishnamurthy, D., J. A. Rolia, S. Majumdar (2006). “A Synthetic Workload Generation Technique for Stress Testing Session-Based Systems.” In: *IEEE Transactions on Software Engineering* 32.11, pp. 868–882 (cit. on pp. 2, 27, 67, 146, 175, 435, 444).

Lee, J., Y. S. Hong (2018). “Data-Driven Prediction of Change Propagation Using Dependency Network.” In: *Engineering Applications of Artificial Intelligence* 70, pp. 149–158 (cit. on p. 447).

Lehvä, J., N. Mäkitalo, T. Mikkonen (2019). “Consumer-Driven Contract Tests for Microservices: A Case Study.” In: *Proceedings of the 20th International Conference on Product-Focused Software Process Improvement (PROFES 2019)*. Vol. 11915. Lecture Notes in Computer Science. Springer, pp. 497–512 (cit. on p. 452).

Lewis, J., M. Fowler (2014). *Microservices*. URL: `https://martinfowler.com/articles/microservices.html` (visited on 11/27/2018) (cit. on p. 18).

Li, J., Y. Xiong, X. Liu, L. Zhang (2013). “How Does Web Service API Evolution Affect Clients?” In: *Proceedings of the 20th International Conference on Web Services (ICWS 2013)*. IEEE Computer Society, pp. 300–307 (cit. on pp. 100, 102, 103).

Li, Z., J. Tian (2003). “Testing the Suitability of Markov Chains as Web Usage Models.” In: *Proceedings of the 27th International Computer Software and Applications Conference (COMPSAC 2003): Design and Assessment of Trustworthy Software-Based Systems*. IEEE Computer Society, pp. 356–361 (cit. on pp. 27, 314, 407, 435).

Lloyd, S. P. (1982). “Least Squares Quantization in PCM.” In: *IEEE Transactions on Information Theory* 28.2, pp. 129–136 (cit. on pp. 188, 455).

Luo, G., X. Zheng, H. Liu, R. Xu, D. Nagumothu, R. Janapareddi, E. Zhuang, X. Liu (2019). “Verification of Microservices Using Metamorphic Testing.” In: *Proceedings of the 19th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2019)*. Vol. 11944. Lecture Notes in Computer Science. Springer, pp. 138–152 (cit. on p. 452).

Luo, Q., A. Nair, M. Grechanik, D. Poshyvanyk (2017). “FOREPOST: Finding Performance Problems Automatically with Feedback-Directed Learning Software Testing.” In: *Empirical Software Engineering* 22.1, pp. 6–56 (cit. on p. 440).

Lutteroth, C., G. Weber (2008). “Modeling a Realistic Workload for Performance Testing.” In: *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (ECOC 2008)*. IEEE Computer Society, pp. 149–158 (cit. on pp. 2, 27, 67, 146, 175, 435, 445).

Mahalakshmi, G., S. Sridevi, S. Rajaram (2016). “A Survey on Forecasting of Time Series Data.” In: *Proceedings of the 2016 International Conference on Computing Technologies and Intelligent Data Engineering (ICCTIDE’16)*. 2016 International Conference on Computing Technologies and Intelligent Data Engineering (ICCTIDE’16), pp. 1–8 (cit. on pp. 35, 456).

Mahali, P., D. P. Mohapatra (2018). “Model Based Test Case Prioritization Using UML Behavioural Diagrams and Association Rule Mining.” In: *International Journal of System Assurance Engineering and Management* 9.5, pp. 1063–1079 (cit. on p. 437).

Malik, H., B. Adams, A. E. Hassan (2010a). “Pinpointing the Subsystems Responsible for the Performance Deviations in a Load Test.” In: *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE 2010)*. IEEE Computer Society, pp. 201–210 (cit. on pp. 43, 441).

Malik, H., B. Adams, A. E. Hassan, P. Flora, G. Hamann (2010b). “Using Load Tests to Automatically Compare the Subsystems of a Large Enterprise System.” In: *Proceedings of the 34th Annual IEEE International Computer Software and Applications Conference (COMPSAC 2010)*. IEEE Computer Society, pp. 117–126 (cit. on pp. 43, 441).

Malik, H., H. Hemmati, A. E. Hassan (2013). “Automatic Detection of Performance Deviations in the Load Testing of Large Scale Systems.” In: *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*. IEEE Computer Society, pp. 1012–1021 (cit. on pp. 43, 441).

Malik, H., Z. M. Jiang, B. Adams, A. E. Hassan, P. Flora, G. Hamann (2010c). “Automatic Comparison of Load Tests to Support the Performance Analysis of Large Enterprise Systems.” In: *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*. IEEE Computer Society, pp. 222–231 (cit. on pp. 43, 441).

Maňásek, M., P. Tůma (2019). *Charles University SIS Access Log Dataset*. Zenodo. `http://doi.org/10.5281/zenodo.3241445` (cit. on pp. 65, 262, 264, 355, 357).

Martinich, J. (2008). *Production and Operations Management: An Applied Modern Approach*. Wiley India Pvt. Limited (cit. on p. 34).

Massey Jr., F. J. (1951). “The Kolmogorov-Smirnov Test for Goodness of Fit.” In: *Journal of the American Statistical Association* 46.253, pp. 68–78 (cit. on pp. 271, 342).

Mazkatli, M., D. Monschein, J. Grohmann, A. Koziolek (2020). “Incremental Calibration of Architectural Performance Models with Parametric Dependencies.” In: *Proceedings of the IEEE International Conference on Software Architecture (ICSA 2020)*. accepted, to appear (cit. on p. 451).

Meier, P., S. Kounev, H. Koziolek (2011). “Automated Transformation of Component-Based Software Architecture Models to Queueing Petri Nets.” In: *Proceedings of the 19th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2011)*. IEEE Computer Society, pp. 339–348 (cit. on p. 450).

Meinke, K., P. Nycander (2015). “Learning-Based Testing of Distributed Microservice Architectures: Correctness and Fault Injection.” In: *Revised Selected Papers of the 13th International Conference on Software Engineering and Formal Methods (SEFM*

*2015) Collocated Workshops: ATSE, HOFM, MoKMaSD, and VERY*SCART*. Vol. 9509. Lecture Notes in Computer Science. Springer, pp. 3–10 (cit. on p. 439).

Menascé, D. A. (2002). “Load Testing of Web Sites.” In: *IEEE Internet Computing* 6.4, pp. 70–74 (cit. on p. 435).

Menascé, D. A., V. A. F. Almeida (2002). *Capacity Planning for Web Services: Metrics, Models and Methods*. 1st. Prentice Hall. 608 pp. (cit. on pp. 2, 24–29, 33–35, 41, 67, 146, 175, 176, 435, 436, 456).

Menascé, D. A., V. A. F. Almeida, R. Fonseca, M. A. Mendes (1999). “A Methodology for Workload Characterization of E-Commerce Sites.” In: *Proceedings of the 1st ACM Conference on Electronic Commerce (EC 1999)*. ACM, pp. 119–128 (cit. on pp. 2, 4, 24, 25, 27, 29, 43, 183, 400, 407, 435, 437, 455).

Menascé, D. A., V. A. F. Almeida, R. H. Riedi, F. Ribeiro, R. C. Fonseca, W. M. Jr (2003). “A Hierarchical and Multiscale Approach to Analyze E-Business Workloads.” In: *Performance Evaluation* 54.1, pp. 33–57 (cit. on p. 436).

Michael, N., N. Ramannavar, Y. Shen, S. Patil, J.-L. Sung (2017). “CloudPerf: A Performance Test Framework for Distributed and Dynamic Multi-Tenant Environments.” In: *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE 2017)*. ACM, pp. 189–200 (cit. on p. 442).

Michelsen, J., J. English (2013). *Service Virtualization: Reality Is Overrated*. Apress (cit. on p. 452).

Micro Focus (2020[a]). *LoadRunner Professional - Application Load Testing Software*. URL: `https://www.microfocus.com/en-us/products/loadrunner-professional/overview` (visited on 07/16/2020) (cit. on pp. 42, 68, 73, 448).

– (2020[b]). *Silk Performer*. URL: `https://www.microfocus.com/en-us/products/silk-performer/overview` (visited on 07/16/2020) (cit. on pp. 42, 68, 73, 448).

Mills, C. (2018). *Amazon’s Website Went down Just as Prime Day 2018 Kicked Off*. URL: `https://bgr.com/2018/07/16/amazon-website-down-prime-day-2018-best-deals/` (visited on 07/16/2020) (cit. on pp. 179, 180).

Milovanovic, V., D. Milicev (2015). “An Interactive Tool for UML Class Model Evolution in Database Applications.” In: *Software and System Modeling* 14.3, pp. 1273–1295 (cit. on p. 447).

Montgomery, D. C., G. C. Runger (2003). *Applied Statistics and Probability for Engineers*. 3rd. John Wiley & Sons, Inc. (cit. on pp. 150, 364).

Mostafa, S., X. Wang, T. Xie (2017). “PerfRanker: Prioritization of Performance Regression Tests for Collection-Intensive Software.” In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, pp. 23–34 (cit. on p. 439).

Musa, J. D. (1993). “Operational Profiles in Software-Reliability Engineering.” In: *IEEE Software* 10.2, pp. 14–32 (cit. on pp. 23, 435).

Myers, G. J. (2004). *The Art of Software Testing*. 2nd ed. John Wiley & Sons, Inc. (cit. on pp. 148, 449).

Nagarajan, A., A. Vaddadi (2016). “Automated Fault-Tolerance Testing.” In: *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICST Workshops 2016)*. IEEE Computer Society, pp. 275–276 (cit. on p. 438).

Natella, R., D. Cotroneo, H. Madeira (2016). “Assessing Dependability with Software Fault Injection: A Survey.” In: *ACM Computing Surveys* 48.3, 44:1–44:55 (cit. on p. 438).

Netflix, Inc. (2012). *Eureka*. URL: `https://github.com/Netflix/eureka` (visited on 07/16/2020) (cit. on p. 248).

– (2013). *Zuul*. URL: `https://github.com/Netflix/zuul` (visited on 07/16/2020) (cit. on p. 250).

Newman, S. (2015). *Building Microservices - Designing Fine-Grained Systems*. 1st. O’Reilly (cit. on pp. 2, 3, 18, 19, 117, 126, 142, 145, 256).

Nguyen, M.-D., Y.-S. Cho (2020). “A Hybrid Generative Model for Online User Behavior Prediction.” In: *IEEE Access* 8, pp. 3761–3771 (cit. on p. 456).

Niedermaier, S., F. Koetter, A. Freymann, S. Wagner (2019). “On Observability and Monitoring of Distributed Systems - an Industry Interview Study.” In: *Proceedings of the 17th International Conference on Service-Oriented Computing (ICSOC 2019)*. Vol. 11895. Lecture Notes in Computer Science. Springer, pp. 36–52 (cit. on p. 16).

noconnor (2017). *JUnitPerf*. URL: `https://github.com/noconnor/JUnitPerf` (visited on 07/16/2020) (cit. on p. 453).

North, D. (2006). “Introducing BDD.” In: *Better Software* 12 (cit. on pp. 54, 222, 225, 226, 244, 442, 457, 464).

Novatec Consulting GmbH (2020[a]). *OpenAPM — Your Custom Open Source APM Solution*. URL: `https://openapm.io` (visited on 07/16/2020) (cit. on p. 16).

– (2020[b]). *Sonatype Nexus*. URL: `https://repository.novatec-gmbh.de/index.html` (visited on 07/16/2020) (cit. on p. 282).

Nygard, M. T. (2018). *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf (cit. on p. 20).

Okanović, D., S. Beck, L. Merz, C. Zorn, L. Merino, A. van Hoorn, F. Beck (2020). "Can a Chatbot Support Software Engineers with Load Testing? Approach and Experiences." In: *Proceedings of the 11th ACM/SPEC International Conference on Performance Engineering (ICPE 2020)*. ACM, pp. 120–129 (cit. on pp. 443, 470).

Okanović, D., A. van Hoorn, C. Heger, A. Wert, S. Siegl (2016). "Towards Performance Tooling Interoperability: An Open Format for Representing Execution Traces." In: *Proceedings of the 13th European Workshop on Computer Performance Engineering (EPEW 2016)*. Springer, pp. 94–108 (cit. on pp. 6, 16, 20, 255, 257, 339, 462).

Okanović, D., A. van Hoorn, C. Zorn, F. Beck, V. Ferme, J. Walter (2019). "Concern-Driven Reporting of Software Performance Analysis Results." In: *Companion of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE 2019)*. ACM, pp. 1–4 (cit. on pp. 221, 225, 442, 443, 458, 470, 471).

Oliveira, A. B. de, S. Fischmeister, A. Diwan, M. Hauswirth, P. F. Sweeney (2017). "Perphecy: Performance Regression Test Selection Made Simple but Effective." In: *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)*. IEEE Computer Society, pp. 103–113 (cit. on p. 439).

OpenAPI Initiative (2020). *OpenAPI Initiative*. URL: `https://www.openapis.org/` (visited on 07/16/2020) (cit. on pp. 6, 21, 61, 75–77, 114, 116, 118, 250, 467).

OpenCensus (2020). *OpenCensus*. URL: `https://opencensus.io/` (visited on 07/16/2020) (cit. on pp. 17, 257).

OpenTelemetry (2020). *OpenTelemetry*. URL: `https://opentelemetry.io/` (visited on 07/16/2020) (cit. on p. 17).

Oracle (2020). *SimpleDateFormat (Java Platform SE 8)*. URL: `https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html` (visited on 07/16/2020) (cit. on p. 89).

Oracle Corporation (2020). *OpenJDK: Jmh*. URL: `http://openjdk.java.net/projects/code-tools/jmh/` (visited on 07/16/2020) (cit. on p. 453).

Oruç, A. F., T. Ovatman (2016). "Testing of Web Services Using Behavior-Driven Development." In: *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016), Volume 2*. SciTePress, pp. 85–92 (cit. on p. 457).

OWASP (2020). *Cross-Site Request Forgery (CSRF)*. URL: `https://owasp.org/www-community/attacks/csrf` (visited on 07/16/2020) (cit. on p. 71).

Padgham, L., Z. Zhang, J. Thangarajah, T. Miller (2013). "Model-Based Test Oracle Generation for Automated Unit Testing of Agent Systems." In: *IEEE Transactions on Software Engineering* 39.9, pp. 1230–1244 (cit. on p. 437).

Pahl, C., P. Jamshidi (2016). "Microservices: A Systematic Mapping Study." In: *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016)*. Vol. 1. SciTePress, pp. 137–146 (cit. on p. 18).

Palenga, M. (2018). "Declarative User Experience Regression Analysis in Continuous Performance Engineering." Master's Thesis (cit. on pp. 46, 65, 136, 225, 441, 551).

Pearson F.R.S, K. (1901). "LIII. On Lines and Planes of Closest Fit to Systems of Points in Space." In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11, pp. 559–572 (cit. on p. 363).

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, É. Duchesnay (2011). "Scikit-Learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12.85, pp. 2825–2830 (cit. on p. 251).

Penta, M. D., G. Canfora, G. Esposito, V. Mazza, M. Bruno (2007). "Search-Based Testing of Service Level Agreements." In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*. ACM, pp. 1090–1097 (cit. on p. 440).

Peroli, M., F. D. Meo, L. Viganò, D. Guardini (2018). "MobSTer: A Model-Based Security Testing Framework for Web Applications." In: *Journal of Software Testing, Verification and Reliability* 28.8 (cit. on p. 437).

Pietrantuono, R., S. Russo, A. Guerriero (2018). "Run-Time Reliability Estimation of Microservice Architectures." In: *Proceedings of the 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*. IEEE Computer Society, pp. 25–35 (cit. on p. 438).

– (2020). "Testing Microservice Architectures for Operational Reliability." In: *Journal of Software Testing, Verification and Reliability* 30.2 (cit. on p. 438).

Pivotal Software, Inc. (2020[a]). *RabbitMQ - Messaging That Just Works*. URL: `https://www.rabbitmq.com/` (visited on 07/16/2020) (cit. on pp. 81, 248).

– (2020[b]). *Spring Boot*. URL: `https://spring.io/projects/spring-boot` (visited on 07/16/2020) (cit. on pp. 250, 285).

– (2020[c]). *Spring Shell*. URL: `https://projects.spring.io/spring-shell/` (visited on 07/16/2020) (cit. on p. 251).

Popa, B. (2018). *GitLab Says It Imported 100,000 Repositories after Microsoft's GitHub Takeover*. softpedia. URL: `https://news.softpedia.com/news/gitlab-says-it-imported-100-000-repositories-after-microsoft-s-github-takeover-521433.shtml` (visited on 07/16/2020) (cit. on pp. 178, 180).

Portillo-Dominguez, A. O., M. Wang, J. Murphy, D. Magoni, N. Mitchell, P. F. Sweeney, E. R. Altman (2014). "Towards an Automated Approach to Use Expert Systems in the Performance Testing of Distributed Systems." In: *Proceedings of the 2nd Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing (JAMAICA@ISSTA 2014)*. ACM, pp. 22–27 (cit. on p. 452).

Prometheus (2020). *Prometheus - Monitoring System & Time Series Database*. URL: `https://prometheus.io/` (visited on 07/16/2020) (cit. on p. 339).

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria (cit. on pp. 36, 198, 207, 248).

Rahman, M., J. Gao (2015). "A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development." In: *Proceedings of the 9th IEEE Symposium on Service-Oriented System Engineering (SOSE 2015)*. IEEE Computer Society, pp. 321–325 (cit. on pp. 443, 457).

Rajlich, V. (1997). "A Model for Change Propagation Based on Graph Rewriting." In: *Proceedings of the 13th International Conference on Software Maintenance (ICSM 1997)*. IEEE Computer Society, pp. 84–91 (cit. on p. 446).

Reichelt, D. G., S. Kühne (2018). "Better Early than Never: Performance Test Acceleration by Regression Test Selection." In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE 2018)*. ACM, pp. 127–130 (cit. on pp. 439, 454).

Reichelt, D. G., S. Kühne, W. Hasselbring (2019). "PeASS: A Tool for Identifying Performance Changes at Code Level." In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. IEEE, pp. 1146–1149 (cit. on pp. 439, 454).

Ruffo, G., R. Schifanella, M. Sereno, R. Politi (2004). "WALTy: A User Behavior Tailored Tool for Evaluating Web Application Performance." In: *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (NCA 2004)*. IEEE Computer Society, pp. 77–86 (cit. on pp. 2, 67, 146, 175, 435, 444).

Saff, D., M. D. Ernst (2004). “An Experimental Evaluation of Continuous Testing during Development.” In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*. ACM, pp. 76–85 (cit. on p. 16).

Sarinho, V. T. (2019). “"BDD Assemble!": A Paper-Based Game Proposal for Behavior Driven Development Design Learning.” In: *Proceedings of the 1st IFIP TC 14 Joint International Conference on Entertainment Computing and Serious Games (ICEC-JCSG 2019)*. Vol. 11863. Lecture Notes in Computer Science. Springer, pp. 431–435 (cit. on p. 458).

Schermann, G., J. Cito, P. Leitner, U. Zdun, H. C. Gall (2018). “We’re Doing It Live: A Multi-Method Empirical Study on Continuous Experimentation.” In: *Information and Software Technology* 99, pp. 41–57 (cit. on p. 437).

Schermann, G., D. Schöni, P. Leitner, H. C. Gall (2016). “Bifrost: Supporting Continuous Deployment with Automated Enactment of Multi-Phase Live Testing Strategies.” In: *Proceedings of the 17th International Middleware Conference (Middleware 2016)*. ACM, p. 12 (cit. on p. 438).

Scheuner, J., J. Cito, P. Leitner, H. C. Gall (2015). “Cloud WorkBench: Benchmarking IaaS Providers Based on Infrastructure-as-Code.” In: *Proceedings of the 24th International Conference on World Wide Web Companion (WWW 2015)*. ACM, pp. 239–242 (cit. on p. 442).

Scheuner, J., P. Leitner, J. Cito, H. C. Gall (2014). “Cloud Work Bench - Infrastructure-as-Code Based Cloud Benchmarking.” In: *Proceedings of the IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom 2014)*. IEEE Computer Society, pp. 246–253 (cit. on p. 442).

Schroeder, B., A. Wierman, M. Harchol-Balter (2007). “Open versus Closed: A Cautionary Tale.” In: *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI 2006)*. USENIX (cit. on pp. 24, 404, 436, 469).

Schubert, E., J. Sander, M. Ester, H.-P. Kriegel, X. Xu (2017). “DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN.” In: *ACM Transactions on Database Systems* 42.3, 19:1–19:21 (cit. on pp. 189, 401, 455).

Schulz, H. (2019). *ContinuITy-Project/Idpa-Demo: IDPA Demo v1.0.0*. Version v1.0.0. Zenodo. `https://doi.org/10.5281/zenodo.2647977` (cit. on pp. 8, 254, 560).

– (2020a). *ContinuITy-Project/Forecastic: Release at Project End (v0.5.3)*. Version v0.5.3. Zenodo. `https://doi.org/10.5281/zenodo.3966834` (cit. on pp. 8, 9, 248, 465, 560).

– (2020b). *ContinuITy-Project/Sutmock: SUT Mock Version v0.2.1*. Version v0.2.1. Zenodo. `https://doi.org/10.5281/zenodo.3966903` (cit. on pp. 258, 465, 560).

– (2020c). *ContinuITy-Project/Zipkin-to-Open-Xtrace-Converter: Zipkin to OPEN.Xtrace Converter 0.0.4*. Version 0.0.4. Zenodo. `https://doi.org/10.5281/zenodo.3966893` (cit. on pp. 17, 257, 465, 560).

– (2020d). *Docker Images for ContinuITy*. Zenodo. `https://doi.org/10.5281/zenodo.3966908` (cit. on pp. 247, 465, 560).

Schulz, H., T. Angerstein, A. van Hoorn (2018). "Towards Automating Representative Load Testing in Continuous Software Engineering." In: *Companion of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018)*. ACM, pp. 123–126 (cit. on pp. VIII, 58, 69, 73).

Schulz, H., T. Angerstein, D. Okanović, A. van Hoorn (2019a). "Microservice-tailored Generation of Session-based Workload Models for Representative Load Testing." In: *Proceedings of the 27th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2019)*. IEEE Computer Society, pp. 323–335 (cit. on pp. VIII, 65, 147, 151, 158, 164, 168, 170, 266, 318).

– (2019b). *Replication Package: Microservice-Tailored Generation of Session-Based Workload Models for Representative Load Testing*. Zenodo. `https://doi.org/10.5281/zenodo.3333367` (cit. on pp. 8, 318, 410, 465, 560).

Schulz, H., T. Angerstein, M. Palenga, A. Hidiroglu (2020a). *ContinuITy-Project/ContinuITy: Release at Project End (v2.9.346)*. Version v2.9.346. Zenodo. `https://doi.org/10.5281/zenodo.3966805` (cit. on pp. 8, 9, 248, 249, 465, 560).

Schulz, H., A. Dang (2020). *ContinuITy-Project/Clustinator: Release at Project End (v0.7.4)*. Version v0.7.4. Zenodo. `https://doi.org/10.5281/zenodo.3966829` (cit. on pp. 8, 9, 248, 465, 560).

Schulz, H., A. van Hoorn (2020). "Representative Load Testing in Continuous Software Engineering: Automation and Maintenance Support." In: *Software Engineering (SE 2020), Fachtagung Des GI-Fachbereichs Softwaretechnik*. Vol. P-300. LNI. Gesellschaft für Informatik e.V., pp. 149–150 (cit. on p. IX).

Schulz, H., D. Okanović, A. van Hoorn, V. Ferme, C. Pautasso (2019c). "Behavior-Driven Load Testing Using Contextual Knowledge — Approach and Experiences." In: *Proceedings of the 10th ACM/SPEC International Conference on Performance*

Engineering (ICPE 2019). ACM, pp. 265–272 (cit. on pp. VIII, 65, 179–181, 198, 214, 223, 226–229, 231, 232, 234, 410, 555–557).

Schulz, H., D. Okanović, A. van Hoorn, V. Ferme, C. Pautasso (2019d). *Behavior-Driven Load Testing Using Contextual Knowledge—Approach and Experiences*. Zenodo. `https://doi.org/10.5281/zenodo.2558279` (cit. on pp. 8, 410, 465, 560).

Schulz, H., D. Okanović, A. van Hoorn, P. Tůma (2021). "Context-tailored Workload Model Generation for Continuous Representative Load Testing." In: *Proceedings of the 12th ACM/SPEC International Conference on Performance Engineering (ICPE 2021)*. To appear. ACM (cit. on pp. VII, 65, 177, 266, 356, 374).

Schulz, H., A. van Hoorn, D. Okanović, S. Siegl, C. Heger, A. Wert, T. Angerstein, A. Hidiroglu, M. Palenga, C. Zorn, V. Ferme, A. Avritzer (2019e). *ContinuITy: Automated Load Testing in DevOps*. Poster presented at the 10th ACM/SPEC International Conference on Performance Engineering (ICPE 2019) (cit. on p. IX).

Schulz, H., A. van Hoorn, P. Tůma, D. Okanović (2020b). *Supplementary Material to Context-Tailored Workload Model Generation for Continuous Representative Load Testing*. Zenodo. `https://doi.org/10.5281/zenodo.4355190` (cit. on pp. 8, 356, 362, 365, 369, 374, 376, 385, 465, 560).

Schulz, H., A. van Hoorn, A. Wert (2019f). *Replication Package: Reducing the Maintenance Effort for Parameterization of Representative Load Tests Using Annotations*. Zenodo. `https://doi.org/10.5281/zenodo.3255388` (cit. on pp. 8, 274, 284, 285, 291, 293, 294, 465, 560).

– (2020c). "Reducing the Maintenance Effort for Parameterization of Representative Load Tests Using Annotations." In: *Journal of Software Testing, Verification and Reliability* 30.1 (cit. on pp. VII, 69, 75, 78, 79, 83, 85–87, 98–100, 102, 105, 106, 266, 275, 406).

Schwaber, K., M. Beedle (2001). *Agile Software Development with Scrum*. 1st. Prentice Hall (cit. on pp. 14, 426).

Segall, I., R. Tzoref-Brill (2015). "Feedback-Driven Combinatorial Test Design and Execution." In: *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR 2015)*. ACM, 12:1–12:6 (cit. on p. 440).

Shams, M., D. Krishnamurthy, B. H. Far (2006). "A Model-Based Approach for Testing the Performance of Web Applications." In: *Proceedings of the 3rd International Workshop on Software Quality Assurance (SOQUA 2006)*. ACM, pp. 54–61 (cit. on pp. 27, 435, 437, 444).

Shindler, M., A. Wong, A. Meyerson (2011). “Fast and Accurate K-Means for Large Datasets.” In: *Proceedings of the 25th Annual Conference on Neural Information Processing Systems 2011*, pp. 2375–2383 (cit. on p. 455).

Silva, J. d. A., E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. P. d. L. F. de Carvalho, J. Gama (2013). “Data Stream Clustering: A Survey.” In: *ACM Computing Surveys* 46.1, 13:1–13:31 (cit. on p. 455).

SmartBear Software (2020). *OpenAPI Specification | Swagger*. URL: `https://swagger.io/specification/` (visited on 07/16/2020) (cit. on p. 117).

Soeken, M., R. Wille, R. Drechsler (2012). “Assisted Behavior Driven Development Using Natural Language Processing.” In: *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns (TOOLS 2012)*. Vol. 7304. Lecture Notes in Computer Science. Springer, pp. 269–287 (cit. on p. 457).

Sohan, S. M., C. Anslow, F. Maurer (2015). “A Case Study of Web API Evolution.” In: *Proceedings of the 11th Congress on Services (SERVICES 2015)*. IEEE computer society, pp. 245–252 (cit. on pp. 100, 104, 105).

Sonatype, Inc. (2013). *REST*. URL: `https://oss.sonatype.org/nexus-restlet1x-plugin/default/docs/rest.html` (visited on 07/16/2020) (cit. on pp. 262, 283).

– (2020[a]). *Nexus Repository | Software Component Management*. URL: `https://www.sonatype.com/product-nexus-repository` (visited on 07/16/2020) (cit. on pp. 71, 273, 276, 281).

– (2020[b]). *Sonatype/Nexus - Docker Hub*. URL: `https://hub.docker.com/r/sonatype/nexus` (visited on 07/16/2020) (cit. on p. 282).

– (2015). *Sonatype/Nexus-Public*. URL: `https://github.com/sonatype/nexus-public` (visited on 07/16/2020) (cit. on p. 282).

– (2020[c]). *The Central Repository Search Engine*. URL: `https://search.maven.org/` (visited on 07/16/2020) (cit. on p. 282).

Soper, T. (2012). *Tons of Traffic: Amazon Dominates Online Retail during Christmas Week*. URL: `https://www.geekwire.com/2012/christmas-day-2012-retail-visits-increase-27-compared-2011/` (visited on 07/16/2020) (cit. on pp. 178–181).

Stahl, T., M. Völter, J. Bettin, A. Haase, S. Helsen (2006). *Model-Driven Software Development - Technology, Engineering, Management*. John Wiley & Sons (cit. on p. 447).

Stefan, P., V. Horký, L. Bulej, P. Tuma (2017). “Unit Testing Performance in Java Projects: Are We There Yet?” In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE 2017)*. ACM, pp. 401–412 (cit. on p. 453).

Steinberg, D., F. Budinsky, E. Merks, M. Paternostro (2008). *EMF: Eclipse Modeling Framework*. 2nd ed. Pearson Education. 949 pp. Google Books: `sA0zOZuDXhgC` (cit. on p. 447).

Syer, M. D., Z. M. Jiang, M. Nagappan, A. E. Hassan, M. N. Nasser, P. Flora (2014). “Continuous Validation of Load Test Suites.” In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*. ACM, pp. 259–270 (cit. on pp. 3, 68).

Syer, M. D., W. Shang, Z. M. Jiang, A. E. Hassan (2017). “Continuous Validation of Performance Test Workloads.” In: *Automated Software Engineering* 24.1, pp. 189–231 (cit. on pp. 3, 68, 445).

Taylor, S. J., B. Letham (2018). “Forecasting at Scale.” In: *The American Statistician* 72.1, pp. 37–45 (cit. on pp. 35–37, 63, 176, 214, 251, 392, 393, 407, 456, 463, 470).

The Linux Foundation (2020). *Production-Grade Container Orchestration - Kubernetes*. URL: `https://kubernetes.io/` (visited on 07/16/2020) (cit. on p. 412).

YAML (2020). *The Official YAML Web Site*. URL: `http://yaml.org/` (visited on 07/16/2020) (cit. on pp. 21, 63, 75, 77, 136, 199).

OpenTracing (2020). *The OpenTracing Project*. URL: `https://opentracing.io/` (visited on 07/16/2020) (cit. on pp. 17, 257).

Tricentis (2020). *Continuous Load Testing: Reinventing Load Testing for DevOps*. URL: `https://www.tricentis.com/resources/continuous-load-testing-reinventing-load-testing-for-devops/` (visited on 07/16/2020) (cit. on p. 448).

Usman, M., M. Z. Iqbal, M. U. Khan (2020). “An Automated Model-Based Approach for Unit-Level Performance Test Generation of Mobile Applications.” In: *Journal of Software Evolution and Process* 32.1 (cit. on pp. 437, 454).

Utting, M., B. Legeard (2007). *Practical Model-Based Testing - a Tools Approach*. Morgan Kaufmann (cit. on p. 436).

Van Rossum, G., F. L. Drake Jr (1995). *Python Reference Manual*. Centrum voor Wiskunde en Informatica Amsterdam (cit. on pp. 36, 248).

Van Hoorn, A., A. Aleti, T. F. Düllmann, T. Pitakrat (2018). "ORCAS: Efficient Resilience Benchmarking of Microservice Architectures." In: *Proceedings of the 29th IEEE International Symposium on Software Reliability Engineering Workshops (ISSRE Workshops 2018)*. IEEE Computer Society, pp. 146–147 (cit. on p. 439).

Van Hoorn, A., M. Rohr, W. Hasselbring (2008). "Generating Probabilistic and Intensity-Varying Workload for Web-Based Software Systems." In: *Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW 2008): Performance Evaluation: Metrics, Models and Benchmarks*. Springer, pp. 124–143 (cit. on pp. 45, 131, 435, 436, 444, 547).

Van Hoorn, A., M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, D. Kieselhorst (2009). *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework*. 0921. Department of Computer Science, Kiel University, Germany (cit. on p. 16).

Veeraraghavan, K., J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, Y. J. Song (2016). "Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services." In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016)*. USENIX Association, pp. 635–651 (cit. on p. 438).

Verenich, I., M. Dumas, M. L. Rosa, F. M. Maggi, I. Teinemaa (2019). "Survey and Cross-Benchmark Comparison of Remaining Time Prediction Methods in Business Process Monitoring." In: *ACM Transactions on Intelligent Systems and Technology* 10.4, 34:1–34:34 (cit. on p. 456).

Versteeg, S., M. Du, J.-G. Schneider, J. Grundy, J. Han, M. Goyal (2016). "Opaque Service Virtualisation: A Practical Tool for Emulating Endpoint Systems." In: *Companion of the 38th International Conference on Software Engineering (ICSE 2016)*. ACM, pp. 202–211 (cit. on pp. 3, 6, 58, 148, 173, 350, 452, 470).

Vieira, M., H. Madeira, K. Sachs, S. Kounev (2012). "Resilience Benchmarking." In: *Resilience Assessment and Evaluation of Computing Systems*. Springer, pp. 283–301 (cit. on p. 438).

Vögele, C. (2018). "Automatic Extraction and Selection of Workload Specifications for Load Testing and Model-Based Performance Prediction." PhD Thesis (cit. on pp. 3, 41, 403, 440, 471).

Vögele, C., A. Brunnert, A. Danciu, D. Tertilt, H. Krcmar (2014). "Using Performance Models to Support Load Testing in a Large SOA Environment." In: *Proceedings* *of the 3rd ACM International Workshop on Large Scale Testing (LT 2014)*. ACM, pp. 5–6 (cit. on p. 27).

Vögele, C., A. van Hoorn, E. Schulz, W. Hasselbring, H. Krcmar (2018). "WESSBAS: Extraction of Probabilistic Workload Specifications for Load Testing and Performance Prediction - a Model-Driven Approach for Session-Based Application Systems." In: *Software and System Modeling* 17.2, pp. 443–477 (cit. on pp. 2, 3, 5, 6, 25, 27–30, 32, 37, 50, 52, 63, 64, 67, 73, 121, 127, 146, 159, 175, 176, 181, 183, 193, 251, 255, 271, 289, 293, 341, 348, 383, 386, 400, 403, 404, 406, 407, 413, 435–437, 440, 441, 444, 455, 462, 464, 468, 469, 547).

Walter, J., S. Eismann, J. Grohmann, D. Okanovic, S. Kounev (2018). "Tools for Declarative Performance Engineering." In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE 2018)*. ACM, pp. 53–56 (cit. on p. 458).

Walter, J., A. van Hoorn, H. Koziolek, D. Okanović, S. Kounev (2016). "Asking "What"?, Automating the "How"?: The Vision of Declarative Performance Engineering." In: *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE 2016)*. ACM, pp. 91–94 (cit. on pp. 221, 458).

Wang, S., I. Keivanloo, Y. Zou (2014). "How Do Developers React to RESTful API Evolution?" In: *Proceedings of the 12th International Conference on Service-Oriented Computing (ICSOC 2014)*. Vol. 8831. Lecture Notes in Computer Science. Springer, pp. 245–259 (cit. on pp. 100, 106, 276, 278, 280, 290, 315).

Wang, T., J. Wei, W. Zhang, H. Zhong, T. Huang (2014). "Workload-Aware Anomaly Detection for Web Applications." In: *Journal of Systems and Software* 89, pp. 19–32 (cit. on p. 455).

Wang, Y., S. Wagner (2018). "Combining STPA and BDD for Safety Analysis and Verification in Agile Development: A Controlled Experiment." In: *Proceedings of the 19th International Conference on Agile Processes in Software Engineering and Extreme Programming (XP 2018)*. Vol. 314. Lecture Notes in Business Information Processing. Springer, pp. 37–53 (cit. on pp. 442, 457).

WeatherAds (2014). *Weather and eCommerce: How Weather Impacts Retail Website Traffic and Online Sales*. URL: `http://www.weatherads.io/blog/2014/august/weather-and-ecommerce-how-weather-impacts-retail-website-traffic-and-online-sales` (visited on 07/16/2020) (cit. on pp. 178–180).

Weaveworks, Inc. (2020). *Sock Shop*. URL: `https://microservices-demo.github.io/` (visited on 07/16/2020) (cit. on pp. 21, 147, 262, 317, 339, 421).

Weise, E. (2016). *Netflix down for about 2.5 Hours Saturday*. URL: `https://eu.usatoday.com/story/tech/news/2016/10/01/netflix-goes-down-saturday-afternoon/91396200/` (visited on 07/16/2020) (cit. on pp. 178–180).

Wen, Q., L. Sun, X. Song, J. Gao, X. Wang, H. Xu (2020). *Time Series Data Augmentation for Deep Learning: A Survey*. arXiv: `2002.12478 [cs, eess, stat]`. URL: `http://arxiv.org/abs/2002.12478` (visited on 04/30/2020) (cit. on p. 407).

Wiedemann, A. M., T. Schulz (2017). "Key Capabilities of DevOps Teams and Their Influence on Software Process Innovation: A Resource-Based View." In: *Proceedings of the 23rd Americas Conference on Information Systems (AMCIS 2017)*. Association for Information Systems (cit. on p. 17).

Wikipedia (2020). *Wikipedia:Pageview Statistics*. URL: `https://en.wikipedia.org/wiki/Wikipedia:Pageview_statistics` (visited on 07/16/2020) (cit. on pp. 179–181).

Wolpert, D. H., W. G. Macready (1997). "No Free Lunch Theorems for Optimization." In: *IEEE Transactions on Evolutionary Computation* 1.1, pp. 67–82 (cit. on p. 36).

Wright, A. (2019). *JSON Schema: A Media Type for Describing JSON Documents*. URL: `http://json-schema.org/draft/2019-09/json-schema-core.html` (visited on 07/16/2020) (cit. on pp. 93, 97, 196, 523).

Wynne, M., A. Hellesoy, S. Tooke (2017). *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf. 476 pp. Google Books: `fA9QDwAAQBAJ` (cit. on pp. 222, 225, 226, 442, 457).

Yan, M., H. Sun, X. Wang, X. Liu (2012). "Building a TaaS Platform for Web Service Load Testing." In: *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2012)*. IEEE Computer Society, pp. 576–579 (cit. on pp. 56, 471).

Yu, J., J. Han, J.-G. Schneider, C. M. Hine, S. Versteeg (2017). "A Petri-Net-Based Virtual Deployment Testing Environment for Enterprise Software Systems." In: *The Computer Journal* 60.1, pp. 27–44 (cit. on p. 451).

Zhang, J., S. C. Cheung (2002). "Automated Test Case Generation for the Stress Testing of Multimedia Systems." In: *Journal of Software Practice and Experience* 32.15, pp. 1411–1435 (cit. on p. 440).

Zhang, P., S. G. Elbaum, M. B. Dwyer (2011a). “Automatic Generation of Load Tests.” In: *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE Computer Society, pp. 43–52 (cit. on p. 440).

– (2011b). “Automatic Generation of Load Tests.” In: *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE Computer Society, pp. 43–52 (cit. on p. 440).

Zhou, J., B. Zhou, S. Li (2014a). “Automated Model-Based Performance Testing for PaaS Cloud Services.” In: *Proceedings of the IEEE 38th Annual Computer Software and Applications Conference (COMPSAC Workshops 2014)*. IEEE Computer Society, pp. 644–649 (cit. on p. 435).

– (2014b). “LTF: A Model-Based Load Testing Framework for Web Applications.” In: *Proceedings of the 14th International Conference on Quality Software (QSIC 2014)*. IEEE, pp. 154–163 (cit. on pp. 435, 445).

Zipkin (2020). *OpenZipkin · A Distributed Tracing System*. URL: `https://zipkin.io/` (visited on 07/16/2020) (cit. on pp. 255, 257, 339).

List of Acronyms

AMQP Advanced Message Queuing Protocol

API application programming interface

APM application performance management

ARIMA autoregressive integrated moving average

BDD Behavior-driven Development

BDLT Behavior-driven Load Testing

CI/CD continuous integration and delivery

CLI command-line interface

CSE continuous software engineering

CSRF cross-site request forgery

CSS Cascading Style Sheets

CSV comma-separated values

DLIM Descartes Load Intensity Model

DPE declarative performance engineering

DQL Descartes Query Language

DSL domain-specific language

EBNF extended Backus–Naur form

EFSM	extended finite state machine
GaAs	guards and actions
GUI	graphical user interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure-as-a-Service
IDPA	Input Data and Properties Annotation
IoT	Internet of things
IQR	inter-quartile range
JAR	Java Archive
JDBC	Java Database Connectivity
JMH	Java Microbenchmarking Harness
JMS	Java Message Service
JSON	JavaScript Object Notation
JWT	Java Web Token
LCtL	Load Test Context-tailoring Language
LOESS	locally estimated scatterplot smoothing
MDSE	model-driven software engineering
PCA	principal component analysis
PCM	Palladio Component Model
POM	Maven Project Object Model
PPTAM	Production and Performance Testing Based Application Monitoring
REST	Representational State Transfer
SaaS	software as a service
SIS	student information system

SLA service-level agreement

SOA service-oriented architecture

SPL Stochastic Performance Logic

STL seasonal and trend decomposition using Loess

SUT system under test

TaaS testing as a service

TDD test-driven development

UI user interface

UML Unified Modeling Language

URL Uniform Resource Locator

UUID universally unique identifier

WMR workload model repository

XGBoost eXtreme Gradient Boosting

XML Extensible Markup Language

YAML YAML Ain’t Markup Language

List of Figures

List of Tables

Appendix

Appendix A

IDPA YAML Schemata

This chapter provides the JavaScript Object Notation (JSON) schemata (Wright, 2019) of the Input Data and Properties Annotation (IDPA). We represent it in the YAML format (Ben-Kiki et al., 2009) for better readability and because there is a direct mapping between the JSON and the YAML format. Furthermore, we utilize a *@type* property denoting YAML tags, e.g., the *@type* value *http* is represented by *!<http>*. We use this workaround that is not part of the JSON schema definition because currently, there is no support for such YAML tags.

There are two schemata: one for the application model and one for the annotation model. We split the respective schemata into meaningful parts for better readability. The subsequently presented parts are also subsequent in the YAML file. Three dots (...) indicate that the subsequent part is a sub item and are not part of the YAML file (see Listing A.1 and Listing A.5).

A.1. Application Model Schema

```
---
$schema: http://json-schema.org/draft-04/schema#
title: Application Model
type: object
additionalProperties: false
properties:
  id:
    type: string
  version:
    type: string
  timestamp:
    type: string
    pattern: \d{4}-\d{2}-\d{2}T\d{2}-\d{2}-\d{2}-\d
        {3}\w* #yyyy-MM-dd'T'HH-mm-ss-SSSX
  endpoints:
    type: array
    items:
      oneOf:
      - $ref: '#/definitions/HttpEndpoint'
required:
- timestamp
definitions:
  ...
```

Listing A.1: Application schema.

```yaml
HttpEndpoint:
  type: object
  additionalProperties: false
  properties:
    "@type":
      type: string
      enum:
      - http
      default: http
    id:
      type: string
    domain:
      type: string
    port:
      type: string
    path:
      type: string
    method:
      type: string
    encoding:
      type: string
      default: <no-encoding>
    headers:
      type: array
      items:
        type: string
    parameters:
      type: array
      items:
        $ref: '#/definitions/HttpParameter'
    protocol:
      type: string
  required:
  - "@type"
  - domain
  - port
  - path
  - method
  - protocol
```

Listing A.2: HttpEndpoint schema.

```yaml
HttpParameter:
  type: object
  additionalProperties: false
  properties:
    id:
      type: string
    name:
      type: string
    parameter-type:
      type: string
      enum:
      - req-param
      - body
      - url-part
      - header
      - form
  required:
  - name
  - parameter-type
```

Listing A.3: HttpParameter schema.

A.2. Annotation Model Schema

```yaml
---
&any_input
oneOf:
- $ref: '#/definitions/ExtractedInput'
- $ref: '#/definitions/DirectListInput'
- $ref: '#/definitions/CsvInput'
- $ref: '#/definitions/CsvInputGroup'
- $ref: '#/definitions/RandomNumberInput'
- $ref: '#/definitions/RandomStringInput'
- $ref: '#/definitions/CounterInput'
- $ref: '#/definitions/DatetimeInput'
- $ref: '#/definitions/EnvironmentInput'
- $ref: '#/definitions/CombinedInput'
- $ref: '#/definitions/JsonInput'
- $ref: '#/definitions/ConciseJsonInput'
```

Listing A.4: List of all provided `Input` implementations, where other schemata can refer to.

```yaml
$schema: http://json-schema.org/draft-04/schema#
title: Annotation Model
type: object
additionalProperties: false
properties:
  id:
    type: string
  overrides:
    type: array
    items:
      oneOf:
      - $ref: '#/definitions/HttpEndpointOverride'
      - $ref: '#/definitions/HttpParameterOverride'
  inputs:
    type: array
    items: *any_input
  endpoint-annotations:
    type: array
    items:
      $ref: '#/definitions/EndpointAnnotation'
definitions:
  ...
```
Listing A.5: ApplicationAnnotation schema.

```
EndpointAnnotation:
  type: object
  additionalProperties: false
  properties:
    id:
      type: string
    endpoint:
      type: string
    overrides:
      type: array
      items:
        oneOf:
        - $ref: '#/definitions/HttpEndpointOverride'
        - $ref: '#/definitions/HttpParameterOverride'
    parameter-annotations:
      type: array
      items:
        $ref: '#/definitions/ ParameterAnnotation'
  required:
  - endpoint
```

Listing A.6: EndpointAnnotation schema.

```yaml
ParameterAnnotation:
  type: object
  additionalProperties: false
  properties:
    id:
      type: string
    parameter:
      type: string
    input: *any_input
    overrides:
      type: array
      items:
        oneOf:
        - $ref: '#/definitions/HttpParameterOverride'
  required:
  - parameter
  - input
```

Listing A.7: ParameterAnnotation schema.

```yaml
HttpEndpointOverride:
  type: object
  additionalProperties: false
  properties:
    HttpEndpoint.domain:
      type: string
    HttpEndpoint.port:
      type: string
    HttpEndpoint.encoding:
      type: string
    HttpEndpoint.protocol:
      type: string
    HttpEndpoint.header:
      type: string
    HttpEndpoint.base-path:
      type: string

HttpParameterOverride:
  type: object
  additionalProperties: false
  properties:
    HttpParameter.encoded:
      type: string
```

Listing A.8: Override schemata.

```yaml
ExtractedInput:
  type: object
  additionalProperties: false
  properties:
    "@type":
      type: string
      enum:
      - extracted
      default: extracted
    id:
      type: string
    initial:
      type: string
    extractions:
      type: array
      items:
        oneOf:
        - $ref: '#/definitions/RegExExtraction'
        - $ref: '#/definitions/JsonPathExtraction'
  required:
  - "@type"
  - extractions
```

Listing A.9: ExtractedInput schema.

```
RegExExtraction:
  type: object
  additionalProperties: false
  properties:
    id:
      type: string
    from:
      type: string
    pattern:
      type: string
    response-key:
      type: string
      enum: [ body, header, status ]
      default: body
    template:
      type: string
      default: (1)
    match-number:
      type: integer
      default: 1
    fallback:
      type: string
      default: NOT FOUND
  required:
  - from
  - pattern
```

Listing A.10: RegExExtraction schema.

```yaml
JsonPathExtraction:
  type: object
  additionalProperties: false
  properties:
    id:
      type: string
    from:
      type: string
    json-path:
      type: string
    response-key:
      type: string
      enum: [ body, header, status ]
      default: body
    match-number:
      type: integer
      default: 1
    fallback:
      type: string
      default: NOT FOUND
  required:
  - from
  - json-path
```

Listing A.11: JsonPathExtraction schema.

```
DirectListInput:
  type: object
  additionalProperties: false
  properties:
    "@type":
      type: string
      enum:
      - direct
      default: direct
    id:
      type: string
    data:
      type: array
      items:
        type: string
    associated:
      type: array
      items:
        oneOf:
        - $ref: '#/definitions/DirectListInput'
        - $ref: '#/definitions/CsvInput'
  required:
  - "@type"
  - data
```

Listing A.12: DirectListInput schema.

```yaml
CsvInput:
  type: object
  additionalProperties: false
  properties:
    "@type":
      type: string
      enum:
      - csv
      default: csv
    id:
      type: string
    file:
      type: string
    column:
      type: integer
    separator:
      type: string
      default: ;
    header:
      type: boolean
      default: false
    associated:
      type: array
      items:
        oneOf:
        - $ref: '#/definitions/DirectListInput'
        - $ref: '#/definitions/CsvInput'
  required:
  - "@type"
  - file
  - column
```

Listing A.13: CsvInput schema.

```
CsvInputGroup:
  type: object
  additionalProperties: false
  properties:
    "@type":
      type: string
      enum:
      - csv
      default: csv
    id:
      type: string
    file:
      type: string
    separator:
      type: string
      default: ;
      header:
        type: boolean
        default: false
    columns:
      type: array
      items:
        type: object
        properties:
          id:
            type: string
  required:
  - "@type"
  - file
  - columns
```

Listing A.14: CsvInputGroup schema.

```yaml
RandomNumberInput:
  type: object
  additionalProperties: false
  properties:
    "@type":
      type: string
      enum:
      - randnum
      default: randnum
    id:
      type: string
    lower:
      type: integer
    lower-input:
      type: *any_input
    upper:
      type: integer
    upper-input:
      type: *any_input
  required:
  - "@type"
```
Listing A.15: RandomNumberInput schema.

```yaml
RandomStringInput:
  type: object
  additionalProperties: false
  properties:
    "@type":
      type: string
      enum:
      - randstring
      default: randstring
    id:
      type: string
    template:
      type: string
  required:
  - "@type"
  - template
```
Listing A.16: RandomStringInput schema.

```yaml
CounterInput:
  type: object
  additionalProperties: false
  properties:
    "@type":
      type: string
      enum:
      - counter
      default: counter
    id:
      type: string
    format:
      type: string
    scope:
      type: string
      enum:
      - global
      - user
      - user-iteration
    start:
      type: integer
    increment:
      type: integer
    maximum:
      type: integer
  required:
  - "@type"
  - scope
  - start
  - increment
  - maximum
```

Listing A.17: CounterInput schema.

```yaml
DatetimeInput:
  type: object
  additionalProperties: false
  properties:
    "@type":
      type: string
      enum:
      - datetime
      default: datetime
    id:
      type: string
    format:
      type: string
    offset:
      type: string
  required:
  - "@type"
  - format
```

Listing A.18: DatetimeInput schema.

```yaml
EnvironmentInput:
  type: object
  additionalProperties: false
  properties:
    "@type":
      type: string
      enum:
      - environment
      default: environment
    id:
      type: string
    property:
      type: string
  required:
  - "@type"
  - property
```

Listing A.19: EnvironmentInput schema.

```yaml
CombinedInput:
  type: object
  additionalProperties: false
  properties:
    "@type":
      type: string
      enum:
      - combined
      default: combined
    id:
      type: string
    format:
      type: string
    inputs:
      type: array
      items: *any_input
  required:
  - "@type"
  - format
  - inputs
```

Listing A.20: CombinedInput schema.

```
JsonInput:
  type: object
  additionalProperties: false
  properties:
    "@type":
      type: string
      enum:
      - json
      default: json
    id:
      type: string
    type:
      type: string
      enum:
      - string
      - number
      - object
      - array
    name:
      type: string
    input:
      type: *any_input
    items:
      type: array
      items: *any_input
  required:
  - "@type"
  - type
```

Listing A.21: JsonInput schema.

```yaml
ConciseJsonInput:
  type: object
  additionalProperties: false
  properties:
    "@type":
      type: string
      enum:
      - json
      default: json
    id:
      type: string
    json:
      type:
        oneOf:
        - $ref: '#/definitions/JsonStaticValue'
        - $ref: '#/definitions/JsonDerivedValue'
        - $ref: '#/definitions/JsonObject'
        - $ref: '#/definitions/JsonArray'
  required:
  - "@type"
  - json
```

Listing A.22: ConciseJsonInput schema.

```yaml
JsonStaticValue:
  type: string

JsonDerivedValue:
  type: *any_input

JsonObject:
  type: object
  additionalProperties:
    type:
      oneOf:
      - $ref: '#/definitions/JsonStaticValue'
      - $ref: '#/definitions/JsonDerivedValue'
      - $ref: '#/definitions/JsonObject'
      - $ref: '#/definitions/JsonArray'

JsonArray:
  type: array
  items:
    oneOf:
    - $ref: '#/definitions/JsonStaticValue'
    - $ref: '#/definitions/JsonDerivedValue'
    - $ref: '#/definitions/JsonObject'
    - $ref: '#/definitions/JsonArray'
```
Listing A.23: JsonItem schemata for the JsonInput.

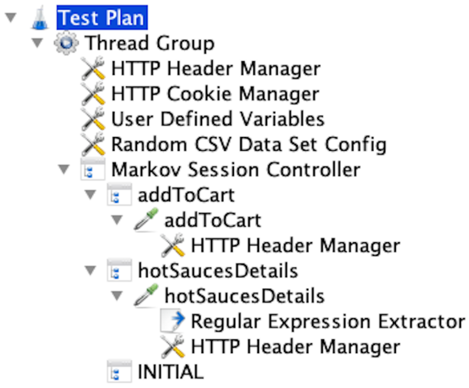
Appendix B

IDPA Transformation Examples

This chapter provides exemplary JMeter (Apache Software Foundation, 2020[a]) and BenchFlow (Ferme and Pautasso, 2018) tests that have been transformed from a WESSBAS (Vögele et al., 2018) workload model and the Input Data and Properties Annotation (IDPA) in Section 6.3.3.1. The workload model specifies to call the two endpoints *hotSaucesDetails* and *addToCart* sequentially.

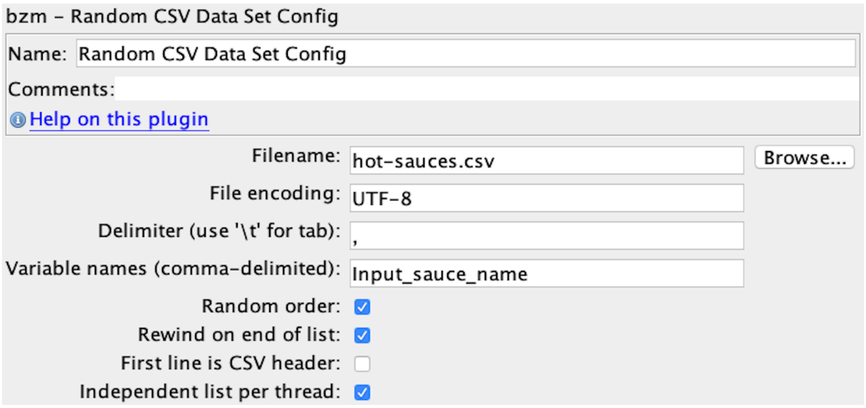
B.1. JMeter

Figures B.1 to B.5 illustrate the generated JMeter test plan as JMeter UI screen shots based on the JMeter version 5.0 with the Markov4JMeter (van Hoorn et al., 2008) and Random CSV Data Set (Fedorov, 2017) plugins installed.



```
▼ Test Plan
  ▼ Thread Group
      HTTP Header Manager
      HTTP Cookie Manager
      User Defined Variables
      Random CSV Data Set Config
    ▼ Markov Session Controller
      ▼ addToCart
        ▼ addToCart
            HTTP Header Manager
      ▼ hotSaucesDetails
        ▼ hotSaucesDetails
            Regular Expression Extractor
            HTTP Header Manager
        INITIAL
```

Figure B.1.: JMeter test plan tree transformed from the IDPA example in Section 6.3.3.1.



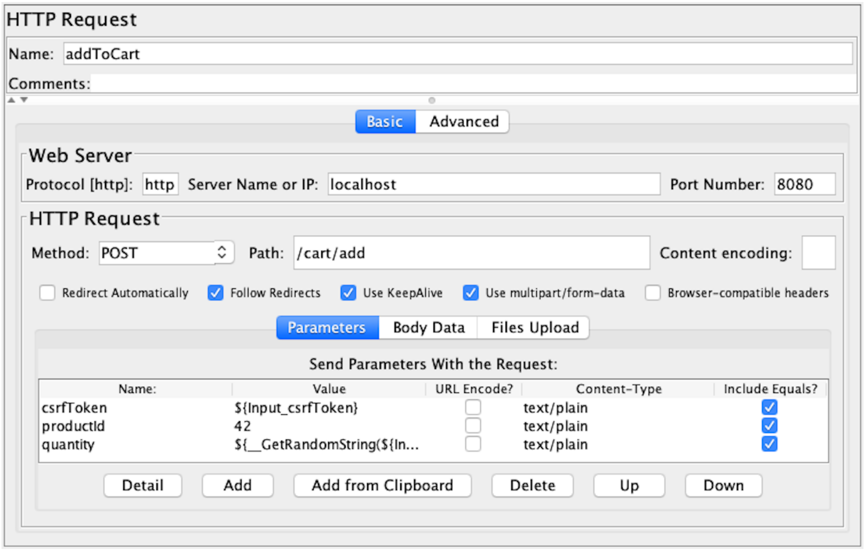
bzm – Random CSV Data Set Config

Name:	Random CSV Data Set Config
Comments:	

Help on this plugin

Filename:	hot-sauces.csv
File encoding:	UTF-8
Delimiter (use '\t' for tab):	,
Variable names (comma-delimited):	Input_sauce_name
Random order:	✓
Rewind on end of list:	✓
First line is CSV header:	☐
Independent list per thread:	✓

Figure B.2.: `RandomCSVDataSetConfig` for the *Input_sauce_name*.



HTTP Request

Name: addToCart

Comments:

Basic | Advanced

Web Server

Protocol [http]: http Server Name or IP: localhost Port Number: 8080

HTTP Request

Method: POST Path: /cart/add Content encoding:

[ ] Redirect Automatically [✓] Follow Redirects [✓] Use KeepAlive [✓] Use multipart/form-data [ ] Browser-compatible headers

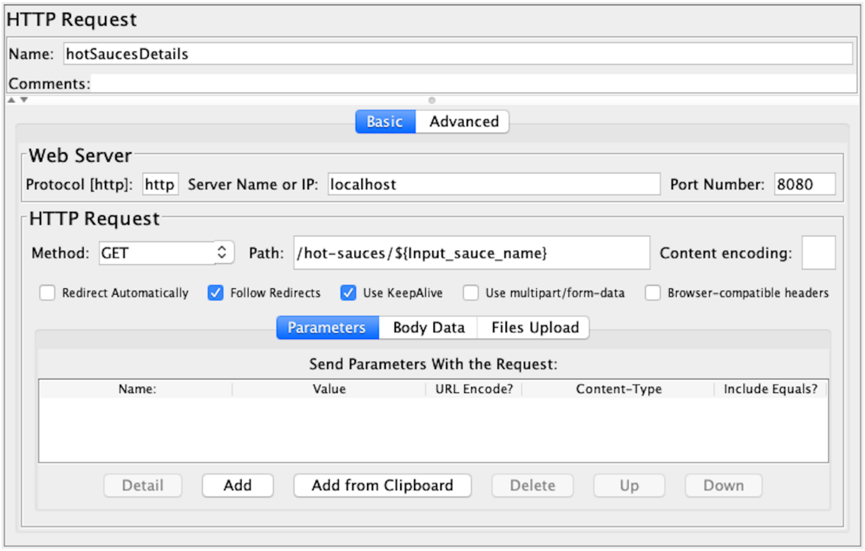
Parameters | Body Data | Files Upload

Send Parameters With the Request:

Name:	Value	URL Encode?	Content-Type	Include Equals?
csrfToken	${Input_csrfToken}	[ ]	text/plain	✓
productId	42	[ ]	text/plain	✓
quantity	${__GetRandomString(${In...	[ ]	text/plain	✓

Detail | Add | Add from Clipboard | Delete | Up | Down

Figure B.3.: `HTTPSamplerProxy` for the *addToCart* request.



HTTP Request

Name: hotSaucesDetails

Comments:

Basic | Advanced

Web Server

Protocol [http]: http Server Name or IP: localhost Port Number: 8080

HTTP Request

Method: GET Path: /hot-sauces/${Input_sauce_name} Content encoding:

[ ] Redirect Automatically [✓] Follow Redirects [✓] Use KeepAlive [ ] Use multipart/form-data [ ] Browser-compatible headers

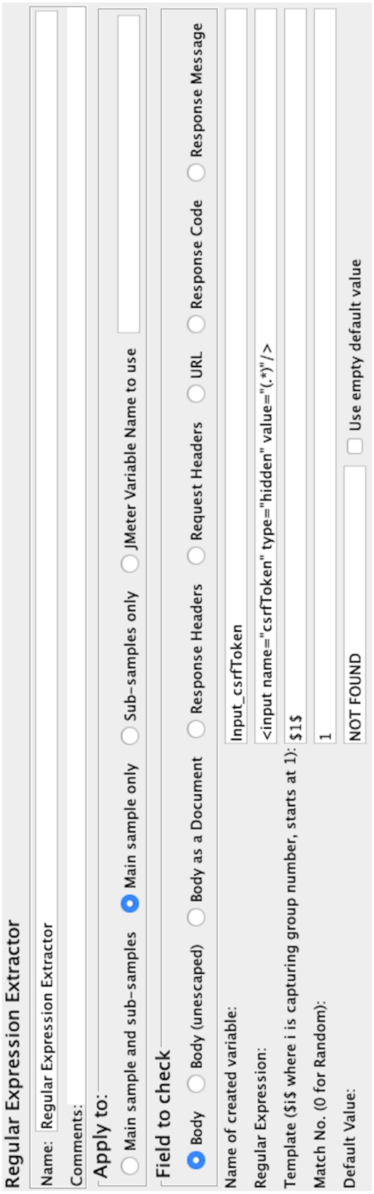
Parameters | Body Data | Files Upload

Send Parameters With the Request:

Name:	Value	URL Encode?	Content-Type	Include Equals?

Detail | Add | Add from Clipboard | Delete | Up | Down

Figure B.4.: `HTTPSamplerProxy` for the *hotSaucesDetails* request.



Regular Expression Extractor

Name: Regular Expression Extractor

Comments:

Apply to:

Main sample and sub-samples [✓ Main sample only] Sub-samples only JMeter Variable Name to use

Field to check

[✓ Body] Body (unescaped) Body as a Document Response Headers Request Headers URL Response Code Response Message

Name of created variable: Input_csrfToken

Regular Expression: <input name="csrfToken" type="hidden" value="(.*)"/>

Template (i where i is capturing group number, starts at 1): 1

Match No. (0 for Random): 1

Default Value: NOT FOUND ☐ Use empty default value

Figure B.5.: `RegexExtractor` for the *Input_csrfToken*.

B.2. BenchFlow

Listings B.1 and B.2 contain the BenchFlow DSL instance based on the version introduced by Palenga (2018). It is assumed that the *hot-sauces.csv* file contains a header *Input_sauce_name*.

```
sut:
  configuration:
    target_service:
      name: sock-shop
      endpoint: localhost:8080/

data-sources:
- path: hot-sauces.csv
  delimiter: ','

workload-items:
  gen_behavior_model0:
    popularity: 100.0%
    inter_operation_timings: fixed-time
    driver_type: http
    mix:
      matrix:
      - [ 0.0%,   0.0%, 100.0% ]
      - [ 0.0%,   0.0%,   0.0% ]
      - [ 0.0%, 100.0%,   0.0% ]
```

Listing B.1: BenchFlow test transformed from the IDPA example in Section 6.3.3.1 (part 1).

```
operations:
- id: INITIAL_STATE
- method: POST
  body:
    csrfToken: ${Input_csrfToken}
    quantity: [ '1', '2', '3', '4', '5' ]
    productId: '42'
  id: addToCart
  endpoint: /cart/add
  think-time:
    mean: 0.0
    deviation: 0.0
  headers:
    X-Requested-With: XMLHttpRequest
    Content-Type: application/x-www-form-urlencoded
  protocol: http
- method: GET
  extract-regexp:
    Input_csrfToken:
      pattern: <input name="csrfToken" type="hidden" value="(.*)"/>
      default: NOT FOUND
      match-number: 1
  id: hotSaucesDetails
  url-parameter:
    sauce: ${Input_sauce_name}
  endpoint: /hot-sauces/${sauce}
  think-time:
    mean: 0.0
    deviation: 0.0
  protocol: http
```

Listing B.2: BenchFlow test transformed from the IDPA example in Section 6.3.3.1 (part 2).

Appendix C

Grammar of Load Test Context-tailoring Language

Below, we provide the formal grammar of the Load Test Context-tailoring Language (LCtL). We express it in extended Backus–Naur form (EBNF). The diagrams presented in Section 8.5 hold the same information but formatted for improved readability. Similar to the diagrams, we use the special terminals '>>' and '<<' for indents and dedents as well as '\n' for newlines respecting the current indentation.

```
scenario ::= ('timeframe:' '\n' timeframe)
    ('context:' '\n' context)?
    ('aggregation:' aggregation)
    ('adjustments:' '\n' adjustments)?
```

Listing C.1: EBNF of the root `scenario` clause of the LCtL.

```
timeframe ::= ’{}’ | (timerange | conditional |
    extended)+

timerange ::= ’- ’ ’!<timerange>’ (’{}’ | ’\n’ ’>>’
    (’from: ’ date ’\n’ (’to: ’ date ’\n’)?
    (’duration: ’ duration ’\n’)?) |
    (’to: ’ date ’\n’ (’duration: ’ duration ’\n’)?) |
    (’duration: ’ duration ’\n’) ’<<’)

conditional ::= ’- ’ ’!<conditional>’ ’\n’ ’>>’
    (name ’: ’ ’\n’ ’>>’ condition ’<<’)+ ’<<’

condition ::= (’is: ’ value ’\n’) | (’exists: ’
    boolean ’\n’) | ((’greater: ’ numeric ’\n’)
    (’less: ’ numeric ’\n’)?) |
    (’less: ’ numeric ’\n’)

extended ::= ’- ’ ’!<extended>’ ’\n’ ’>>’
    ((’beginning: ’ duration ’\n’)
    (’end: ’ duration ’\n’)? |
    (’end: ’ duration ’\n’)) ’<<’

context ::= ’>>’ (name ’: ’ ’\n’ context-def+)+ ’<<’

context-def ::= (’- ’ ’multiplied: ’ numeric ’\n’ ’>>
    ’
    (’added: ’ numeric ’\n’)? | (’- ’ ’added:
    ’ numeric ’\n’ | ’- ’ ’is: ’ value ’\n’) ’>>’)
    ’during:’ ’\n’ timeframe ’<<’

aggregation ::= ’!<’ name ’>’ properties

adjustments ::= (’- ’ ’!<’ name ’>’ properties)+

properties ::= ’{}’ | ’\n’ ’>>’ (name ’: ’ value
    ’\n’)+ ’<<’
```
Listing C.2: EBNF of the LCtL sections.

Appendix D

Grammar of Behavior-driven Load Testing Language

Below, we provide the formal grammar of the BDLT language in extended Backus–Naur form (EBNF), corresponding to the system diagrams in Section 9.2.2 and our previous publication (H. Schulz et al., 2019c).

```
bdlt ::= 'GIVEN' given ('AND' given)* 'WHEN' when
  ('AND' when)* 'THEN' then ('AND' then)*.

given ::= (daterange | nextevent | alterusers |
  assignment | services)

when ::= (vary | event)

then ::= ( run | collect | ensure | break )
```

Listing D.1: EBNF of the BDLT root clauses (H. Schulz et al., 2019c).

```
daterange ::= date 'to' date

nextevent ::= 'the'? 'next' eventId ('after' date)?

alterusers ::= 'the number of users' (adjust
     'set to' number adjust?)

assignment ::= ('the')? id 'is' (number | string |
   enumeration)

services ::= ('the')? ('service' id |
   'services' (id ('and' | ','))+)
```

Listing D.2: EBNF of the *given* clauses of the BDLT language (H. Schulz et al., 2019c). The `services` clause is not contained in the previous publication.

```
vary ::= 'varying' ('the number of users' |
   ('the')? id) ('between' number 'and' number
   ('in steps of' numeric)? | 'among' enumeration )

event ::= ('a' | 'an')? id ('happened' | 'happens')?
   ('on' date | 'from' daterange)
```

Listing D.3: EBNF of the *when* clauses of the BDLT language (H. Schulz et al., 2019c).

```
run ::= 'run ' ('each' | 'the') ' experiment for '
    duration

collect ::= 'collect' id

ensure ::= 'ensure' check

break ::= 'break if' check

check ::= ('the')? (aggregator)? id 'is'
    comparison number
```

Listing D.4: EBNF of the *then* clauses of the BDLT language (H. Schulz et al., 2019c).

```
adjust ::= ('increased' | 'decreased') 'by'
    numeric '%'

number ::= numeric | ('the')? aggregator

aggregator ::= 'maximum' | 'minimum' | 'final' |
    'average' | 'summarized' | numeric 'th percentile'

enumeration ::= '(' string ( ',' string )+ ')'

comparison ::= 'less than' | 'greater than' |
    'equal to' | 'less or equal to' |
    'greater or equal to'
```

Listing D.5: EBNF of the utility clauses of the BDLT language (H. Schulz et al., 2019c).

Appendix E

Overview of Supplementary Material

Table E.1 summarizes the artifacts and material we publish online as a supplement to this dissertation. Relating to the specified dissertation chapters and containing the listed types of artifacts, we refer to the following material:

- Several repositories contain the source code, documentation, and Docker container images of our implementation (Chapter 10). We differentiate between the implementation of the introduced approach (Section 10.1) and additional implementations (Section 10.2).
- We demonstrate the usage of parts of our approach, specifically the Input Data and Properties Annotation (IDPA) (Chapter 6). We provide an HTML document that describes individual steps to be replayed.
- To replicate or extend our evaluation, we have previously published replication packages supplementary to our conference and journal papers, evaluating the automated load test parameterization (Chapter 12), service-tailoring (Chapter 13), context-tailoring (Chapter 14), Behavior-driven Load Testing (BDLT) (Chapter 15), and Domain-based microservice scalability assessment (Section 15.2) approaches.

Table E.1.: Overview of Supplementary Material

		Artifact type									
Reference	**Chapter**	Schemata & grammars	Source code	Container images	Usage instructions	Interactive HTML documentation	Experiment setup & configuration	Experiment instruction/automation	Raw experiment results	Case study protocol/artifacts	Analysis scripts & results
	Approach implementation										
H. Schulz et al., 2020a	10.1		●		●						
H. Schulz and Dang, 2020	10.1		●		●						
H. Schulz, 2020a	10.1		●		●						
H. Schulz, 2020b,c	10.2		●		●						
H. Schulz, 2020d	10			●							
	Approach demonstration										
H. Schulz, 2019	6				●	●					
	Evaluation replication										
H. Schulz et al., 2019f	12	●					●	●	●		●
H. Schulz et al., 2019b	13						●	●	●		●
H. Schulz et al., 2020b	14	●				●	●	●	●	●	●
H. Schulz et al., 2019d	15	●								●	
Avritzer et al., 2020b	15.2		●		●		●	●	●		●