Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Modularization of Representative Load Tests for Microservice Applications

Tobias Angerstein

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Henning Schulz, M.Sc., Dr. Dušan Okanović
Commenced:	June 21, 2018
Completed:	December 21, 2018
CR-Classification:	D.4.8

Abstract

Nowadays, user satisfaction and business value of software applications are mainly influenced by the performance and scalability of software applications. Thus, it is a necessity to test the performance of the application during the application development cycle. In order to have meaningful results, the whole application has to be tested with a representative load. Using representative load tests, the application's performance is tested by a simulated request load, which represents a realistic group of users. In times of DevOps and microservice architectures, each microservice is developed, tested and deployed by a different development team. However, existing approaches, which provide representative load testing, only generate workload for the whole system. Thus, all services have to be deployed, have to be configured by hand, and the execution takes very long compared to typical delivery pipeline executions. In addition, this contradicts the idea of microservice architecture development, where each team develops, tests and deploys its service independently.

Addressing the downsides of system-wide load testing regarding microservices, we propose a concept of a representative load test workload modularization. Instead of targeting the whole system, only dedicated services are targeted by modularizing the workload into service-specific parts. Providing modularized load tests for certain services saves many resources and we expect, that it eases the test integration into a continuous delivery pipeline.

This thesis aims to achieve the following goals:

- Automated representative per-service and integration performance testing
- Simplify the use of load testing for microservice development teams

Since representative load testing requires a pipeline starting from extracting monitoring data, processing the data, generating a workload model and transforming the workload model into an executable load test, there are many different stages, were the modularization of a load test can take place. In this thesis, we elaborate on possible modularization approaches and evaluate them regarding the resource consumption, duration, representativeness, and the practicability.

We implement two of the elaborated approaches, in order to evaluate and compare the results of the modularization approaches. A selection of microservices, which are part of a sample application is load tested with the implemented approaches and compared with the non-modularized load test executions. The results of the experiment show, that the trace modularization approach, which modularizes the request traces, provides the most promising results. However, the results are not significant enough in order to draw a valid conclusion.

Kurzfassung

Heutzutage werden die Benutzerzufriedenheit und der Geschäftswert von Softwareanwendungen hauptsächlich durch die Leistung und Skalierbarkeit von Softwareanwendungen beeinflusst. Daher ist es notwendig, die Performance der Anwendung während des Entwicklungszyklus zu testen. Um aussagekräftige Ergebnisse zu erzielen, muss die gesamte Anwendung mit repräsentativer Belastung getestet werden. Mit repräsentativen Lasttests wird die Performance der Anwendung durch eine simulierte Anforderungslast getestet, die eine realistische Gruppe von Benutzern darstellt. In Zeiten von DevOps und Microservice-Architekturen wird jeder Microservice von einem anderen Entwicklungsteam entwickelt, getestet und eingesetzt. Bestehende Ansätze, welche repräsentative Lasttests ermöglichen, erzeugen jedoch nur eine Nutzerlast für das gesamte System. Daher müssen alle Services gestartet und manuell konfiguriert werden und die Ausführung dauert im Vergleich zu typischen Ausführungszeiten von Delivery Pipelines sehr lange. Darüber hinaus widerspricht dies der Idee der Entwicklung von Microservice-Architekturen, bei der jedes Team seinen Service eigenständig entwickelt, testet und einsetzt.

Um die Nachteile des systemweiten Lasttests für Microservices zu beheben, entwickeln wir ein Konzept, welches die Modularisierung eines repräsentativen Lasttests ermöglicht. Anstatt das gesamte System zu testen, werden nur dedizierte Services durch die Modularisierung des Workloads in servicespezifische Teile getestet. Die Bereitstellung modularisierter Lasttests für bestimmte Dienste spart viel Ressourcen und wir erwarten, dass es die Integration von Lasttests in eine Continuous Delivery Pipeline erleichtert.

Diese Arbeit zielt darauf ab, die folgenden Ziele zu erreichen:

- Automatisierte repräsentative Lasttests auf Serviceebene und Integrationstestebene
- Vereinfachung der Verwendung von Lasttests für Microservice Entwicklungsteams

Die Erzeugung eines repräsentativen Lasttests besteht aus mehreren Arbeitsschritten: Zunächst werden relevante Informationen aus den Monitoringdaten extrahiert, die Daten anschließend verarbeitet, ein Modell des Nutzerverhaltens generiert und anschließend ein ausführbarer Test zur Verfügung gestellt. Zwischen jedem Arbeitsschritt entstehen Zwischenergebnisse, auf die eine Modularisierung angewendet werden kann. In dieser Arbeit erarbeiten wir mögliche Modularisierungsansätze und bewerten sie hinsichtlich Ressourcenverbrauch, Dauer, Repräsentativität und Praktikabilität.

Wir implementieren zwei der erarbeiteten Ansätze, um die Ergebnisse der Modularisierungsansätze zu bewerten und zu vergleichen. Eine Auswahl von Microservices, die Teil einer Beispielanwendung sind, werden mit den implementierten Ansätzen getestet und mit den nicht-modularisierten Lasttestausführungen verglichen. Die Ergebnisse des

Experiments zeigen, dass der Ansatz der Trace-Modularisierung, welcher die Monitoringdaten modularisiert, die vielversprechendsten Ergebnisse liefert. Die Ergebnisse sind jedoch nicht signifikant genug, um eine valide Schlussfolgerung zu ziehen.

Contents

List of Figures

List of Listings

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Because the user satisfaction and business value of modern software application is mainly influenced by the performance of the application, performance testing is an essential part of the todays software development cycle. In order to test the performance of an application, load tests are a suitable method. Load tests generate simulated workload against the application under test in order to test the performance of the application [JH15]. However, the meaningfulness of a load test strongly depends on the used workload. Hence, there are existing approaches such as WESSBAS [VHS+18] which use monitoring data of a life system in order to generate a representative workload. The approaches derive simulated users which represent the overall user behavior by analyzing user sessions of the monitoring data. A created representative load test tries to reflect an equal user behavior as the load in production. The user behavior is thereby mainly characterized by the order and the frequency of the executed requests.

However, in times of DevOps and microservice architectures, using the existing approaches of representative load testing can be very challenging. Microservices are small service units, which are usually designed around business capabilities. The deployment is fully automated. As microservices are developed, tested and automatically deployed by different teams, a representative load test for the whole application is impracticable and contradicts to the core concept of microservice architecture development. In order to validate the functional correctness of a service, usually a component test is used [HF10]. Since problems which are caused by the interaction between different services cannot be covered by testing a single service, integration tests which test the interaction between services are necessary. Considering the performance of a service landscape, there are also existing approaches which address automated performance testing of microservices (see Section 2.2.1). However, state-of-the-art representative

load testing cannot directly target specific services, because the session information is only contained in requests which are submitted to the application by clients.

In this thesis, we extend the concept of representative load testing by enabling automated representative load testing on a service level and compare different modularization approaches. With regard to independent microservice development teams, each development team can test their service without deploying the whole application. In addition, our approach is automated, in order to simplify an implementation into a continuous delivery pipeline. In a nutshell, our approach modularizes the representative workload into service-specific modules which can be used to test a single service, but remain representative.

This thesis is conducted within the research project ContinuITy on "Automated Performance Testing in Continuous Software Engineering" in cooperation with the Novatec GmbH.

1.2 Objectives

In this section, we introduce the objectives and the corresponding research questions with regard to the proposed problem.

1.2.1 Automated Representative Per-service and Integration Load Testing

To the best of our knowledge, existing approaches supporting automated representative load testing are neither capable of testing the performance on a service level nor on a subset of services, because session information is required in order to generate behavior models and the session information usually is not provided in back-end calls. Hence, all services have to be deployed, in order to test a specific service. In order to fill this lack, our goal is to *support automated representative per-service and integration load testing*. Concerning testing a single service with representative load, the requests of the load test have to be mapped to the requests that directly target the service, without loosing the representativeness. Instead of all services, only the service itself and all dependent services which have an influence on the performance of the services of interest, are deployed. By replacing the dependent services with performance stubs, the concept can further improved. However this is not in the scope of the thesis. This thesis aims to design and compare different modularization approaches which enable representative per-service and integration performance testing. This is addressed by the following research question:

Research Question 1: How can existing approaches to representative load testing be extended for usage with microservice applications?

Whereas the performance component test only focuses on a single service, the performance integration test may also focus on multiple services. With regard to the importance of the representativeness of per-service and integration performance testing, this thesis addresses the following research question:

Research Question 2: How representative are the modularization approaches compared to a representative system-level load test?

Since the architecture and the dependencies between the services varing between different microservices and applications, the influence of the architecture has to be analyzed. This is addressed by the following research question:

Research Question 3: How is the representativeness influenced by the architecture and dependencies of an application?

1.2.2 Simplifying the Use of Representative Load Testing for Microservice Development Teams

Besides offering the possibility of representative component and integration load testing, the modularization additionally should simplify the use of load tests in the context of microservices. There are two aspects which are going to be considered: First, the modularization of a representative load test supports the autonomy of each team. The teams can use their own representative load test and do not have to coordinate the tests with other teams. Based on this aspect, the thesis answers the following research question:

Research Question 4: How much does the modularization simplify the use of automated representative load testing in the context of microservice architecture development?

Second, we expect that the use of representative load testing for microservice development teams will be simplified, because service-specific modularized load tests can be more easily integrated into an automated build and deployment process. Compared to a system-level representative load test, the approach saves resources, because we expect, that less services have to be deployed. However, this depends on the architecture of the application. In a worst case scenario, all services have to be deployed, because the targeted service depends on all remaining services. This aspect is considered by the following research question:

Research Question 5: How does the modularization approach affect the resource consumption of a load test?

Furthermore, we expect, that the duration of a representative load test will decrease compared to existing system-level representative load test approaches. Although the think times between the request remain constant, the different request sequence patterns might be less complex and once all different request sequence patterns are executed, a load test can be stopped. In addition, since the modularized load test needs less resources, multiple load tests can be parallelized and the overall duration of all load tests can be decreased. In order to verify this goal, this thesis focuses the following research question:

Research Question 6: How does the approaches affect the needed duration of a load test?

1.3 Methods and Results

In order to answer the research questions, we develop different approaches which meet predefined requirements. Since ContinuITy already consists of an architecture which is able to automatically generate and execute representative load tests on a system-level, the considered approaches should fit into the existing architecture. All identified approaches are going to be compared in the evaluation.

Referring to the second research question the representativeness of the modularized load test is evaluated by comparing the resource consumption such as the CPU utilization, memory usage, and heap usage of each deployed microservice to a system-level load test. In addition, the request rate of each microservice endpoint has to be compared.

As the simplicity of the approaches cannot be quantified and depends on the context in which the approach is used, we answer the third research question by conducting a survey. The goal of the survey is to get feedback by prospective users such as Continuous integration architects or microservice developers, weather the approach is practical and usable. Although a case study, where experts are using the approach in practice would provide more representative results, a survey can already provide some indicators on how the practicability of the approach is assessed by experts. With regard to the significance of the survey, it is important to choose the participants carefully.

In order to identify the influence of the approaches on the resource consumption, we measure meaningful comparable metrics such as the memory usage, CPU utilization, heap usage, and thread count. We compare the run of the system-level representative load test and a service-specific representative load test.

Apart from identifying the influence on the resource consumption we examine the influence on the duration of a load test by using the concept of AlGhmadi et al. [ASSH16]. The authors propose an automated approach which recommends when to stop performance tests by measuring data, which is generated by the performance test and repetitive. We implement this approach and compare the recommended duration of a system-level representative load test and a service-specific modularized load test. As there are multiple possible load test modularization approaches, each approach will be executed and compared to the system-level load test.

Thesis Structure

Chapter 2 – State of the Art and Open Challenges: This chapter presents the background of the thesis and elaborates the open challenges.

Chapter 3 – Load Test Modularization presents the modularization approaches, which were elaborated during the thesis.

Chapter 4 – Modifications to ContinuITy introduces additional contributions to ContinuITy.

Chapter 5 – Implementation gives an overview about the implementation details of the modularization approaches.

Chapter 6 – Evaluation describes the evaluation of the elaborated approaches including the experiment results and a result discussion.

Chapter 2

State of the Art and Open Challenges

Since there are already many concepts and technologies in the context of performance testing, the following chapter introduces the important terms and differentiates already existing approaches from the proposed approaches of this thesis.

2.1 Background

The approaches of this thesis are going to build on existing technologies and methods. In the following, we introduce the technologies and methods which are already existing and are going to be used.

2.1.1 Test Automation Pyramid

Software testing is essential in order to verify the requirements. A common model describing the different levels of automated testing is the test automation pyramid [Mar11]. Figure 2.1 shows different testing levels for functional tests. Each level defines a different testing granularity. The granularity increases from the top to the bottom. Testing an application usually starts with unit testing, which usually tests an application on a method level. Automated component tests are written against individual components of a system. If an application consists of multiple components, the communication of the components is tested using automated integration tests. In order to test the whole application, automated system tests can be used. Since not all tests can be automated, there is usually a small amount of manual tests. The higher the testing level, the less tests are executed, because the execution and maintenance costs are increasing.

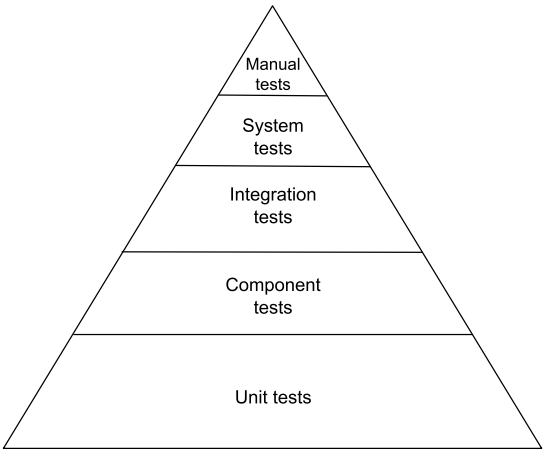


Figure 2.1: Test automation pyramid

2.1.2 Continuous Integration

Continuous Integration is a software development practice and propagates the idea of integrating every small change into production[HF10]. The software is proven to work with every new integration step. In the contrast to conventional development cycles, where the software was only integrated after long time periods including time-consuming acceptance tests, continuous integration represents a new paradigm. Software can be delivered much faster and bugs can be found and fixed earlier. It is considered as an essential practice for professional development teams. Meanwhile there is a huge number of different tools which support Continuous integration. In order to implement Continuous Integration it is recommended to have a version control system, an automated build process and the full commitment of the development team. The version control system is essential, in order to track the different code changes. Second, it is important to be able to automate the build process. The build scripts ideally should be also under version control. As Continuous Integration is a practice, it can only work, if the development team commits to the idea of Continuous Integration.

2.1.3 Component/ Service Testing

Using component testing the application is tested on a component/ service level. Despite unit tests test certain isolated functionalities, unit tests cannot detect bugs which occur as a result of interaction between different units [HF10]. Component tests close this lack by testing larger groups of functionality. The groups of functionality are mostly build along the architecture of the application. Since the tests a more complex and involve a more complex setup, the tests are slower compared to unit tests.

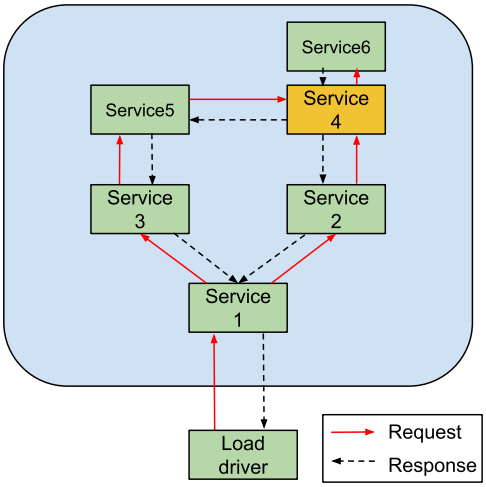


Figure 2.2: Hierarchical structure of an example microservice application

2.1.4 Integration Testing

After the components of a software system are successfully tested, they have to be integrated and the interaction between the components has to be tested [Mye06]. There are existing two different types of integration methods:

- Big-Bang-Integration: All components are integrated and tested at the same time. However, this approach usually is not appropriate, because integrating all components at the same time is very error prone; possible errors are harder to isolate.
- Continuous integration of components: Instead of integrating all components at the same time, the components are integrated step by step. Since there are existing dependencies between the components, the missing components have to be simulated by using *drivers* and *stubs*. Whereas drivers are generating requests, stubs are responding to requests.

Top-down Integration

Since most of the software systems have a hierarchical structure, the top-down integration approach follows the structure. Starting from the root component, the components are integrated using a breadth-first search. All depending components which are not deployed yet, are substituted with a stub which simulates the behavior. This approach does not need any drivers. Focusing the example structure in Figure 2.2, the top-down integration approach initially integrates service 1, service 2 and service 3 have to be

substituted by stubs. Next, Service 2 is deployed and Service 3 and 4 are simulated by stubs. The integration lasts until each service is deployed.

Bottom-up Integration

As an analogy to top-down integration, bottom-up integration also uses the hierarchical structure of the software system. Instead of using breadth-first search, the depth-first search is used. Not deployed components are going to be substituted by drivers. This approach does not need any stubs. Focusing the example structure in Figure 2.2, the bottom-up integration approach initially integrates Service 5, Service 4 has to be substituted by drivers. Next, Service 4 is deployed and Service 2 and 5 are simulated by stubs.

Integration Microservice Validation

Savchenko et al. elaborated the new open challenges of software validation in the context of microservice architectures [SRT15]. Microservice integration testing is an important topic, because microservices architectures are usually increasing the complexity of the infrastructure. Integration validation of microservice architectures includes testing the whole communication of the microservices. Whereas the *functional integration validation* mainly focuses on the correctness of sending requests and the sequence of interactions *load integration validation* includes loading of communication channels to discover the maximum load [SRT15]. However the paper does not focus on load integration testing using representative load.

2.1.5 Load Tests

Load tests are used to analyze the behavior of applications under load using stochastic form-oriented analysis models [DGH+06] [JH15]. Thereby, a load test helps to detect load-related problems. The load-related problems may be either functional or non-functional problems. The model defines the relationship between different requests which are executed against a real system under test. Load tests can be used to conduct performance regression testing. For instance a load test is declared as passed, if the non-functional properties of the tested release is at least as good as the previous release. A load test can be either conducted with normal load which represents the load in an operational environment, with a very high and unrealistic load (stress load testing), in

order two find performance problems. Since ContinuITy focuses on generated representative load testing generated from monitoring data, stress load testing will not be used in the context of this thesis.

2.1.6 Load Test Drivers

Load test drivers are executing the requests which are defined in a load tests. The results of the load test are usually provided as report [CPG13]. There are many different load test drivers such as Apache JMeter, LoadRunner and Gatling [Gat18] available. In the context of the thesis, we are focusing on Apache JMeter. Apache JMeter is an open-source load driver, written in Java and designed to load-test applications and measure performance metrics. Besides HTTP, JMeter supports several common protocols. The load tests can be configured using a UI.

2.1.7 Workload

A workload characterizes the behavior of the users, which are interacting with a software system. A workload is characterized by the workload intensity and the workload mix [JH15]. Workload intensity describes the rate of incoming requests. Workload mix describes the proportion of different request types There are mainly two different types of workloads to be distinguished:

Closed Workload

A closed workload is characterized by the fixed *number of users* [SWH06]. All users have to finish their task before starting again a new interaction with the system. Between the tasks, the user usually waits a predefined amount of time. This time is called *think time*. A closed workload is used, if user behavior has to be simulated.

Open Workload

In contrast to a *closed workload*, there is no fixed number of users. Instead, there is a stream of arriving users. New requests start depending on a predefined arrival rate. Using the open workload model, the exact number of users is ambiguous. An open workload is characterized by its *arrival rate* [SWH06]. An open workload is used, if a certain request rate has to be simulated.

2.1.8 Workload Model

Most commonly, a workload model is a statistical summary of measured workload [Fei02], provided by an APM tool. The summary is applied to all relevant attributes, which are captured. The derived distributions of the different attributes can be used to create a synthetic model. The workload model can be used to create a load test for a certain load driver. Thereby it is important to find a balance between the representativeness and the manageability of the workload model. Too much details may lead to an over-fitting model, which cannot be appropriately generalized [Fei02].

2.1.9 Markov Chain

A Markov chain is a directed, weighted graph. Each transition has a certain probability. Hence, Markov state can be used to formalize the probabilistic relationship between different states of a system. Hence, Markov chains are suitable in order to model the user behavior of a web system [LT03].

In the context of WESSBAS, a transition between two Markov states is enriched think time of the user. This information is usually provided as normal distribution. Since the manipulation of a Markov chain enforces different transitions to be merged, the corresponding normal distributions have to be combined. Thereby, the following mathematical principles can be used:

If two normally distributed random variables X and Y are independent, the sum of the two random variables is another normal distribution:

$$X \sim \mathcal{N}(\mu_x, \sigma_x^2) + Y \sim \mathcal{N}(\mu_y, \sigma_y^2) = X + Y \sim \mathcal{N}(\mu_x + \mu_y, \sigma_x^2 + \sigma_y^2)$$

Considering the deletion of a certain Markov state, at least two different transitions have to be summed up. By using the proposed invariance, the two different normal distributions can be simply summed up.

In addition, the arithmetic mean of two normally distributed random variables X and Y is another normal distribution:

$$\frac{X+Y}{2} \sim \mathcal{N}(\frac{\mu_x + \mu_y}{2}, \frac{\sigma_x^2 + \sigma_y^2}{2^2})$$

According to join two transitions having the same start state and the same target state, this mathematical principle can be used to compute a joined normal distribution.

The probability of each transition can be further used to weight the different normal distributions :

$$\frac{\alpha * X + (1-\alpha) * Y}{2} \sim \mathcal{N}(\frac{\alpha * \mu_x + (1-\alpha) * \mu_y}{2}, \frac{\alpha^2 * \sigma_x^2 + (1-\alpha^2) * \sigma_y^2}{2^2})$$

2.1.10 Workload Model Generator

By using monitoring data, a workload model which models the load of an application and its services is generated. The workload model is usually described using stochastic models, such as Markov chains. There are several generators available, such as SWAT [KRM06], SURGE [BC98] and WESSBAS [VHS+18]. Since in context of ContinuITy, WESSBAS is used, we are focusing on WESSBAS in detail: WESSBAS is a workload model generator, which extracts workload models from recorded session-based monitoring data. [VHS+18] The session identifier is used in order to group the requests to a certain user behavior group by using a clustering algorithm. The workload model is defined as WESSBAS-DSL. Using the session logs, WESSBAS identifies customer usage patterns and generates behavior models as part of WESSBAS-DSL. In addition, WESSBAS-DSL consists of application-specific information, such as protocol information. A WESSBAS-DSL instance can be transformed into executable load tests such as a JMeter test plan. WESSBAS-DSL can also be transformed into workload specifications of the Palladio Component Model [BKR09].

2.1.11 Workload Model Annotation

The annotation of a workload model is a concept, which was elaborated in the context of ContinuITy. Before executing the workload models with a load engine, it is necessary to parameterize and enrich the workload model with contextual information such as specific parameters of a request. The data of the parameters can be extracted from former responses using regex expression. These adoptions will be stored as annotations and can be applied to any other load test of a certain application. Hence, the manual adaption has to be done only once and can be reapplied, if a load test changes. If the API specification changes, the annotation can be changed automatically [SAH18].

2.1.12 Application Performance Monitoring Tools

An APM tool monitors the performance metrics and the availability of monitored applications. The tools can either be used in development in order to detect performance

regression or in production to monitor the performance under realistic usage. Typically, besides measures, APM tools provide traces, which allow to understand the execution logic of the application. Most APM tools also cover invocations between different applications [HHMO17].

2.1.13 Microservices

Microservice architectures describe a certain architectural style [LF14]. The core idea is to build an application out of a service suite. Each microservice has its own process and is communicating through lightweight protocols such as HTTP. Microservices can be deployed automatically, can use different data storages and programming languages and can be independently developed, deployed and scaled. Usually, microservices are designed around business capabilities. For instance, a shopping cart would be represented as a separate microservice [LF14]. This implies new challenges for load testing microservices. In the context of this thesis, the term *service* will be used synonymously with the term *microservice*.

Functional Unit Validation of Microservices

Since microservices are designed to be developed and deployed independently and communicate over open, standardized protocols, the functional validation of a microservice can be done to each microservice separately, ignoring the dependencies to other microservices [SRT15]. Thus, the functional requirements of a microservice can be tested as testing monolithic applications. Because microservices are usually designed to provide some interaction, microservices usually provide external interfaces. In order to guarantee a coincidence with the specification, especially the interfaces have to be tested. However this does not work for testing non-functional requirements such as the performance, because the performance of a certain service can be influenced by other communicating services.

2.1.14 ContinuITy

All the results of this thesis will directly contribute to ContinuITy and the approaches will be evaluated in the context of the existing ContinuITy ecosystem. Hence, it is necessary to focus on the current architecture of ContinuITy. Figure 2.3 shows the architecture of ContinuITy. The representation is reduced to the components, which are relevant for the context of the thesis and consists of five main components:

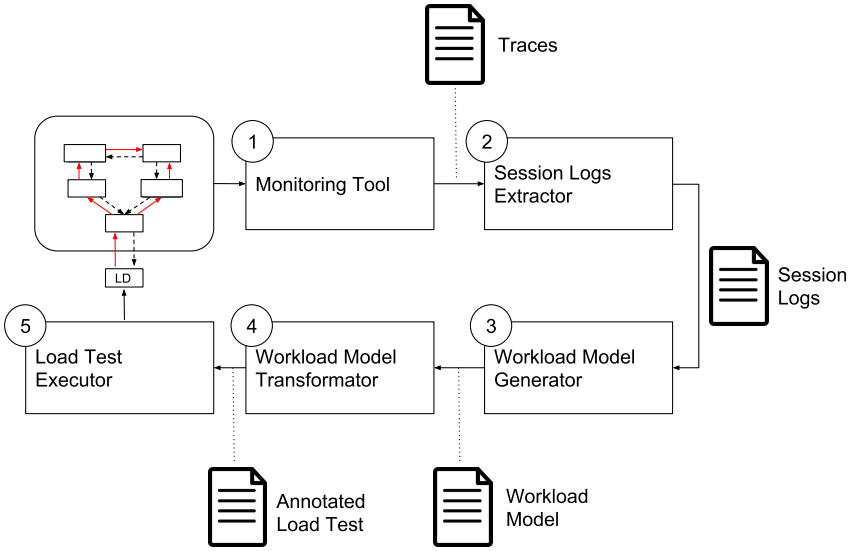


Figure 2.3: Architecture of ContinuITy (abstracted)

- Monitoring Tool: The monitoring tool (1) monitors the application with all its services and provides monitoring data in form of execution *traces* (see in Section 2.1.12). Based on traces, the invocations between services can be analyzed. The quality of the generated workload model is strongly influenced by the quality of the monitoring data.
- Session Logs Generator: Using the *traces* provided by (1), the session logs extractor (2) extracts all relevant requests and generates *session logs*. Besides meta data, such as headers, HTTP parameters and response codes, each session log consists of start timestamp, end timestamp, session cookie, and the request URL. This data can later be used to generate a workload model.
- Workload Model Generator: The workload model generator (3) uses the *session logs* to generate a *workload model*. The *workload model* models the properties of the *session logs* using stochastic models such as Markov chains (see in Section 2.1.8).
- Workload Model Transformation: Because most requests need contextual information in their parameters, the workload model has to be transformed using an annotation (4). An annotation describes the dependencies between the parameters of a request and a response of already executed requests. In addition an annotation also provides input data. For instance, in order to execute request B, it is necessary to have the response of request A. The annotation is application specific and has to be created manually once, but can be reapplied automatically to any new workload model of a certain application (see in Section 2.1.11). The component provides an *annotated load test*.

- Load Test Execution: The *annotated load test* can now be executed by an appropriate load driver (5).

2.2 Related Work

Since our approach of modularizing representative load tests in the context of microservice architectures is related many different research fields and there are already many approaches which solve similar problems, we present the research fields and the corresponding approaches. Besides automated performance testing of microservices, representative load testing is very related to our approach. As we are using real monitoring data from a production environment, relatively new approaches in the field of testing in production are related. Besides performance testing, there are also nonfunctional testing methods for microservices such as reliability and resilience testing. According to the testing pyramid, an application can be tested with different granularities. Whereas conventional representative load tests are testing the whole application, there are also approaches testing the performance of an application on a method level. Another alternative to a system-level load test is to use performance models in order to predict the performance metrics of an application. We also present an approach, which combines performance models with real components.

2.2.1 Automated Performance Testing of Microservices

Camargo et al. [CSMS16] presents an approach, which automates the testing of certain microservices of a microservice architecture. Each microservice exposes a test specification, which can be accessed via HTTP. In addition, a framework is provided which automatically generates the additional HTTP endpoint for each service by using Java annotations. However, the specification of the test has to be provided by developers, hence, the tests do not base on real load and the representativeness of the load test cannot be guaranteed.

Ferme et. al presents [FP18] a declarative domain specific language and a corresponding model-driven framework enabling the declarative description of load tests and the automated execution in the context of continuous software delivery lifecycles. In addition the framework is able to deploy the SUT automatically by using container-based deployment environments such as Docker.The authors do not focus representative testing using generated load tests. However, an integration into ContinuITy is ongoing.

2.2.2 Representative Load Testing

As our approach is modularizing representative load tests, our work is strongly related to representative load testing in general.

Barford et al. [BC98] presents an approach for generating representative workload and simulating a set of real users accessing an application. The authors introduce a Scalable URL Reference Generator (SURGE), which considers probability distributions about the server resources, such as the relative file popularity, the server file size distribution and the idle periods of additional users. The distributions are used in order to model a so called *user equivalents*. Since the approach mainly focuses on web applications with mainly static resources, the approach cannot model users of modern dynamic web applications appropriately.

Menascé et al. [BC98] proposes a methology of identifying and generating workload models in the context of e-commerce. They introduce the Customer Behavior Model Graph (CBMG), which models the behavior of different groups of users. By using a clustering algorithm, the different user groups can be identified. The actions of different user groups are modeled as probabilistic transitions between the different states including the averaged think times of the approach.

Ruffo et al. [RSSP04] uses the presented Customer Behavior Model Graph (CBMG), in order to generate user sessions and representative load on the SUT . The authors propose WALTy, a Web Application Load-based Testing tool, which automatically generates CBMGs using web application log files. An algorithm is presented, which automatically converts CMBGs into user sessions, which are used to generate load for the SUT.

Draheim et al. presents an approach, which aims to load test web applications by simulating realistic user behavior with stochastic form-oriented models [DGH+06]. The models represent the possible actions of users and the statistical probability. An action can have one or more outgoing transitions representing possible responses. Using stochastic formcharts, the navigational choice and input of the users is modeled. Although the models can be automatically executed as load test using MaramaMTE, the stochastic formcharts cannot be automatically generated.

In order to test applications with representative load, WESSBAS [VHS+18] is an approach, that uses session based monitoring data to identify customer usage patterns and generate behavior models. The behavior models are modeled using Markov chains. All user actions are connected with probabilistic transitions. The model can be transformed into a load test, which can be executed to test the application with representative load. However, WESSBAS is only able to generate load tests for the whole application. Load tests for specific services cannot be generated by default, because WESSBAS needs the session ID of the requests which directly target the specific services and this session

ID is usually only provided in the requests which are targeting the entry point of the application.

An alternative to the proposed modularization approach would be to generate a workload model only based on the requests which targeted the set of services. However, in order to generate an appropriate workload model, it is necessary to group the requests in sessions using the session ID of each request [VHS+18]. Depending on the services, which are under test, the requests directly targeting the services often do not provide the session ID. The session ID is often only used in a front end service. Hence, the monitoring data of the targeted service might have to be enriched by the session ID of the parent request.

Instead of using Markov chains, Abbors et al. [AATP12] elaborate MBPeT, which uses Probabilistic Timed Automata (PTA), in order to model the user behavior. A PTA describes a set of locations, which are connected through transitions. A transition can have four different labels, a clock zone, which defines the time to wait until to enter a new location, the probability value, which defines the probability of the transition, the reset, which represents the possibility to reset the global clock and the action of the transition, for instance, the executed request. Since the approach also uses monitoring data in order to generate the model and executing the transformed model again, the approach is very similar to the Wessbas approach and the approach can only be used to test the whole application.

Krishnamurthy et al. [KRM06] propose a technique for representative synthetic workload generation in order to evaluate the performance of enterprise applications. The approach can automatically generate a synthetic workload by using the request logs of the application. The authors presented the framework SWAT, which is able to generate load against the system. SWAT is either able to generate user based sessions, which behave as users or generating requests according to a specified request arrival rate. In addition, SWAT addresses the problem of inter-request dependencies, which might occur if using Markov chains. SWAT is using a workload model, which includes attributes, that can have an influence on the performance of the SUT.

Shams et al. [SKF06] based their work on SWAT. The authors use Extended Finite State Machines (EFSMs) in order to describe the workload model. The authors modified SWAT in a way, that it is able to accept an application model as input including the inter-request dependencies and data dependencies. In addition to FSMs, EFSMs are modeling the inter-request dependencies by introducing state variables, such as *signed_in*. The state variables can be evaluated in predicates of each state and edited by actions. Besides inter-request dependencies, data dependencies are considered, such as session-dependent variables.

Besides modeling user behavior using Probabilistic Timed Automata and Markov chains, Zhou et al. [ZZL14] propose the Context-based Sequential Action Model (CBSAM) and the Workload Parameter Specification Language (WPSL), in order to describe the relationship of workload components and to specify workload parameters, such as the number of virtual users. CBSAM consists of requests, actions and sessions. An action is a set of sequential requests, which form a realistic event. A session is a set of sequential actions which are typically executed in a specific order and duration. The different sessions can be connected using probabilistic transitions. In addition, the model considers contextual information such as the session status and the session ID. In the contrast to already discussed modeling approaches, the CBSAM is much more complex.

2.2.3 Testing in Production

Besides using real data from production, there are also approaches, which directly test the production environment.

Canary release [HF10] is a technique used in the context of continuous delivery to provide a smooth crossover from the current deployed version to the newest version. It is based on the blue-green deployment. Using blue- green deployment both versions are deployed in separated production environments. The incoming traffic can be switched from the previous version to the latest version without any down-time. In addition to the blue-green deployment, the canary release approach aims to send incoming traffic to a router which forwards a proportion of the requests to the new release. Using this approach, the new release can be tested with real load. However this approach also has its drawbacks. New releases can only be tested with the load, which is currently available. Except through the routing rules, the load cannot be manipulated. It is not possible to test the application with a predefined sort of load. Especially regression testing cannot be implemented with canary release, because similar test conditions cannot be guaranteed.

A similar way of testing is A/B testing [HF10]. However, instead of detecting problems and regressions, A/B testing tests a hypothesis. User are randomly split between two different software versions. Once a user is assigned to one certain version, the version stays the same for the user. By instrumenting the interaction with the application, developers can analyze, which version might be more promising.

Related to Canary Release, Veeraraghavan et al. [VMC+16] proposed Kraken, a framework, which supports load testing using live traffic. Kraken shifts live traffic to a dedicated server region or cluster in order to identify resource utilization bottlenecks, such as capacity and load balancing. Thereby, Kraken helped to increase the hardware

utilization by over 20 percent. However, using Kraken, the load cannot be defined and is dependent on the live load.

2.2.4 Reliability and Resilience Testing of Microservices

Besides testing the performance of microservice architectures, reliability and resilience testing is a trending topic in the context of microservices.

Similar to Kraken, Basiri et al. [BBR+16] from Netflix presented an approach, which tests the application in production. Instead of redistributing the load of the system, they are using a service called *Chaos Monkey*. Chaos Monkey is able to randomly choose virtual machines and terminate them. In addition Chaos Monkey can also terminate whole server regions or manipulating requests between services. Using this approach, Netflix can improve the reliability of their services. Such an approach can be easily adapted in order to support reliability testing on a microservice level. Instead of choosing the services randomly, Chaos Monkey could terminate determined virtual machines, where a specific microservice is deployed.

Beside the performance of the application in real use, the ability of recovering itself has to be tested using resilience tests. Resilience testing aims to simulate several failures, which might occur in reality and analyzes the effects of the manipulation. Heorhiadi et al. have introduced a resilience testing framework for microservices called Gremlin. Gremlin simulates possible failures by manipulating the network communication between the microservices [HRJ+16]. However, it needs all microservices to be deployed, because the failures are caused between the microservices. In order to test a specific microservice and test performance impacts, which do not directly lead to a complete failure, a modularized load test is necessary.

2.2.5 Performance Unit Tests

Using the concept of functional unit tests, performance unit tests are testing the performance of specific methods [HLM+15]. Performance unit tests are also consisting of a setup, execution, validation and cleanup phase. In addition to functional unit tests, the setup phase also provides the workload of the test case, which will be executed by a workload model generator. The workload can consist of arbitrary data which is characterized by the different size of the data structures. In the contrast to load tests, only methods and not the whole services are tested. Possible influences between different methods cannot be tested and the tests do display less realistic results.

2.2.6 Software Performance Testing in Virtual Time

In order to accelerate performance testing, Field et al. [FCW18] developed an approach to test components in virtual time. To do so, they enriched mock objects with performance models to provide a realistic behavior of the mock objects. It is also necessary to virtualize the time of the real code in order to have a time consistency between the mock objects and the real code [FCW18]. Since this approach provides performance mocks, it enables top-down performance integration testing. However, since we focus on modularization of load driver generated load, the thesis concentrates on bottom-up performance integration testing.

2.2.7 Model-based Performance Prediction

Instead of testing the performance of already deployed services, there are approaches, which are predicting the performance of an application during a software development process. This approach would constitute an alternative to the proposed approach of the thesis. Using performance prediction saves resources and is much faster, because no real components have to deployed. By using the already existing artifacts, which describe the architecture and the abstract behavior of the resulting software, models are generated. The models are based on performance modeling techniques such as Petri-nets and Queuing networks [BDIS04]. The performance behavior can be predicted in a simulation. The execution duration of this approach would be much less compared to a load test for the whole application. Thus, it can be easily integrated into a continuous deployment pipeline.

An example for such a model is the Palladio component model (PCM), which constitutes a meta model for modeling the architecture of an application with respect to QoS attributes. Thereby, a PCM instance consists of component specifications, assembly models, allocation models and usage models. Using the PCM instance, different artifacts such as Queuing Network Models or Performance prototypes can be derived [BKR09]. Besides using the specification of the application, a PCM instance can be also based on monitoring data using WESSBAS [VHS+18].

The models and prototypes can also be used to predict the performance metrics of certain services. However, the quality and representativeness of the prediction significantly depends on the input artifacts, which describe the architecture of the application and the results are not as meaningful as testing the application under real conditions.

2.3 Open Challenges Addressed by this Thesis

In order to ensure the performance of an application, representative performance tests under realistic conditions are non-negotiable. However, existing approaches supporting automated representative load testing only can test the application on a system-level and all services have to be deployed. This collides with the concept of continuous development, where fast feedback loops and pipelines are necessary and each development team develops and tests their service separately. Performance Unit tests test the target application on a method level. Thus, they are not able to evaluate the service-wide performance including inter-request effects. The performance metrics of a certain service can be encapsulated, if model-based performance prediction is used, but the representativeness of the prediction significantly depends on the input data of the model. An alternative would be a service-specific workload model, which only uses requests directly targeting the service. However, in order to provide a user-behavior-based workload model, the session ID is necessary. To the best of our knowledge, there is no approach, which comprises an equivalent load test to functional component and integration tests. Modularized load tests would close the lack, because only a determined set of services is going to be tested.

In the context of this thesis, this problem is addressed by elaborating different concepts of modularizing load tests, such that only targeted services are tested.

The following four challenges are addressed by the thesis:

- *Automated representative performance-aware per-service and integration testing of microservices*: To the best of our knowledge, existing representative load testing approaches are not capable of testing the application on a service-level. It is a challenge to modularize the workload of the whole application into service-specific parts without loosing the representativeness.
- *Minimize the resource consumption of a load test*: Currently, load testing is very resource consuming. We expect, that the modularization will save resources without loosing the representativeness.
- *Minimize the duration of a load test*: Besides the high resource consumption, in relation to typical continuous delivery stage tasks a representative load test can take very long. It is a challenge to minimize the resource consumption. We expect, that the modularization of the load test leads to a lower duration.
- *Representative load testing in the context of service-specific development teams*: Using current representative load testing approaches in the context of microservices violates a fundamental key concept of microservices. Once a team wants to test their own service, they have additional coordination efforts with other development

teams. It is a challenge, to enable autonomous representative load testing for each development team.

Chapter 3

Load Test Modularization

This chapter introduces the modularization approaches, which are elaborated in the context of the thesis. Section 3.1 places the load test modularization approach in the overall context of load testing of microservices. On the basis of ContinuITy, Section 3.2 provides an overview of the different modularization approaches. Section 3.3 presents the modularization of monitored traces: Each trace is a tree a requests and the first request, which targets dedicated service becomes the new root request of the trace.

Section 3.4 presents the replacement of the extracted session log requests: Since the session logs contain the requests per session ID, the approach replaces each request with a nested request of the corresponding trace, which directly targets the dedicated services.

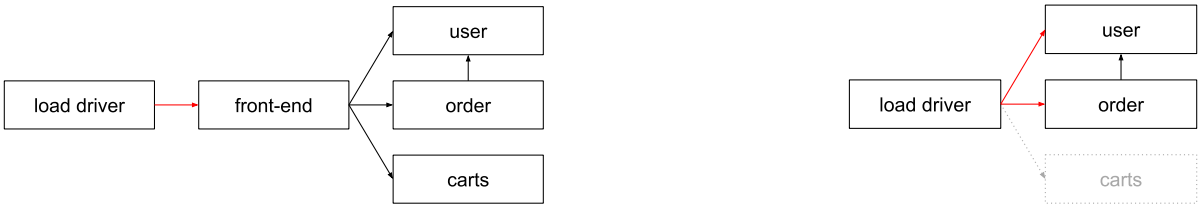
Section 3.5 introduces the modularization of the generated workload model: The generated workload model is represented as Markov chain and each Markov state represents a certain endpoint of the application. The modularization approach now replaces each Markov state which is not representing an endpoint of the target system with a Sub-Markov chain. The Sub-Markov chain is generated by using the modularized traces of the trace modularization approach.

Since all approaches are influencing different steps of the load test pipeline, the advantages and disadvantages are compared in Section 3.6.

3.1 Overview of the approach

In times of microservice architectures and subdivided times, it is a basic principle to isolate the development process of different microservices as much as possible. Hence, each team aims to build its own continuous delivery pipeline, since the pipeline

also depends on the used programming language of the development team [LF14]. Writing unit tests and integration tests is now in the responsibility of the development teams. For functional testing, the delegation of responsibilities eases the process of writing and executing tests. However, when it comes to load testing, the delegation of responsibilities to each development team can become difficult. Especially, if using a existing representative load testing approaches (see Section 2.2.2), which is using monitoring data of the whole system in order to generate a representative workload model (see Section 2.1.14), a single development team cannot execute an isolated representative load test for their microservice. To the best of our knowledge, there is currently existing no representative workload generation approach, which is capable of testing single components of a software architecture. Our approach aims to solve this lack by introducing the modularization of representative workload models. With our approach, each development team is now capable of load testing their single service with a representative load. This means that the desired service and its dependent services are targeted with a realistic user behavior, without deploying all services. In addition, we presume that a modularized load test needs less resources and is faster, because the number of deployed services can be decreased and the time until a load test has covered all possible scenarios is presumed to be shorter than running a non-modularized load test. Thus, it can be better integrated into a continuous delivery pipeline.



(a) Conventional load test example

(b) Modularized load test example

Figure 3.1: Load test modularization example

Figure 3.1 illustrates the difference between a load test generation using existing approaches and a modularized load test. The application consists of four different services. *front-end* is delegating the requests to *user*, *order* and *carts*. In Figure 3.1a, a load driver generates load targeting the front-end of the application. The front-end delegates the requests to different services, depending on the request, which was generated by the load driver. Assuming the development team of microservice *order* wants to integrate a representative load test into its continuous delivery pipeline, they currently have to deploy every service, because current representative load testing approaches are only able to generate a load test targeting the entrypoints of the application such as the endpoints of the service *frontend*. This restriction is caused by the fact that the

representative workload model is based on the behavior of the different session IDs and the IDs are only available in the requests which directly target the entrypoint. By using our modularization approach, which is illustrated in Figure 3.1b, only the services *order* and *user* have to be deployed. With our proposed approaches, the the load test is shrinked to the requests which directly target the dedicated service without loosing the characteristics of the user behavior.

The *load driver* simulates requests, which are directly targeting *order*. The service *user* also has to be deployed, because the *order* service depends on *user*. In addition, the load driver also needs to simulate requests to *user*, because under real conditions, the behavior of this service is influenced by the requests of the front-end. By using this approach, the *order* development team saves resources and can isolate their representative load test as much as possible.

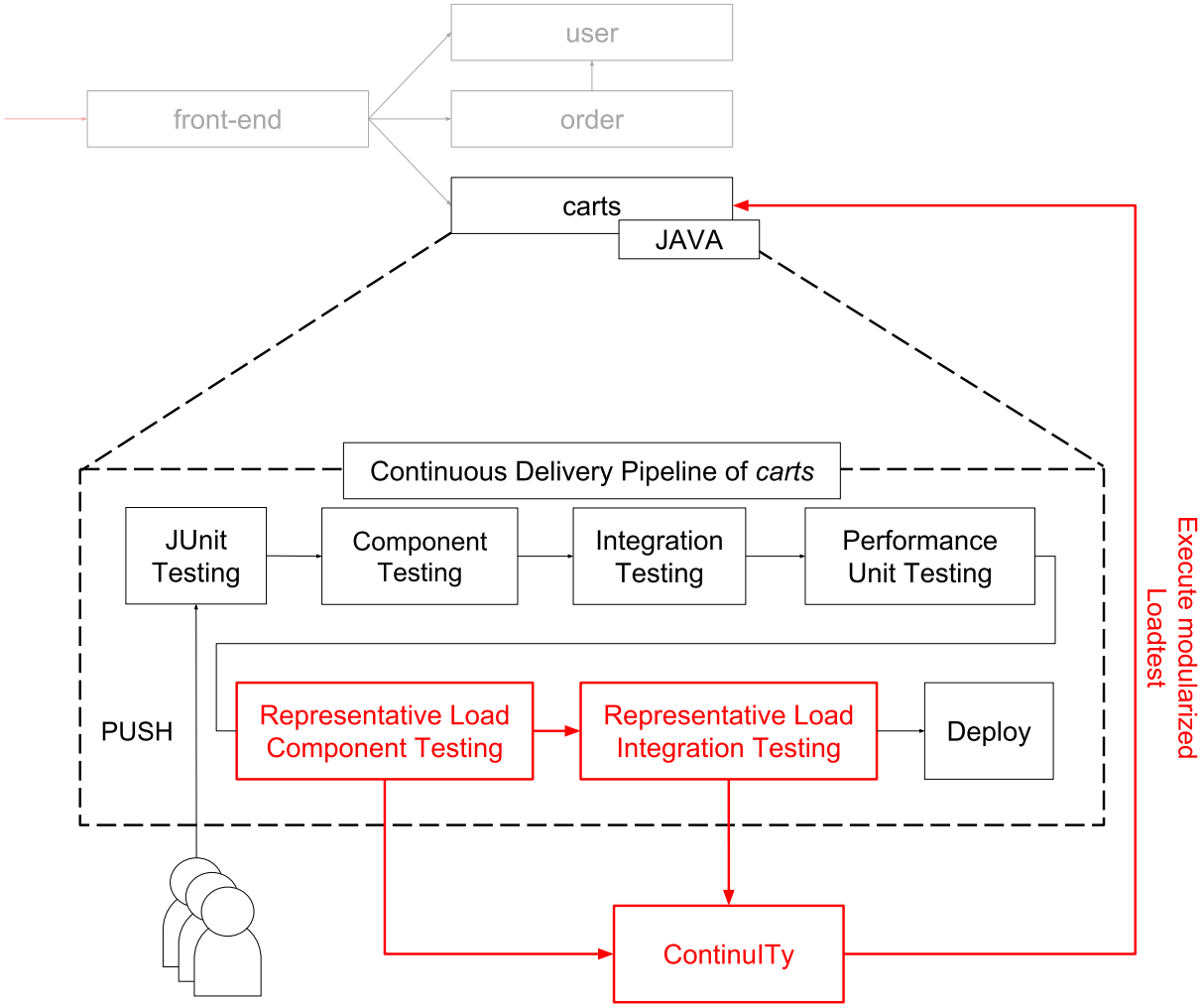


Figure 3.2: Integration of our approach into a continuous delivery pipeline

Figure 3.2 illustrates the integration of ContinuITy into a continuous delivery pipeline. The example focuses on the development pipeline of the microservice *carts*. Usually, a pipeline includes different test stages, such as JUnit testing, functional component and functional integration testing. Besides functional testing, non-functional requirements such as the performance have to be tested. However, a typical build pipeline should be as fast as possible, in order to enable fast feedback loops (see Section 2.1.2). Especially the execution of load tests can take much time in comparison to the execution time of functional tests. By now, representative load testing in a continuous delivery pipeline delays the execution of the pipeline or is not used due to this reason. By using a modularized representative load, the execution can be done with less resources and the execution of different representative load test scenarios can be parallelized. Concerning the example, our approach modularizes the representative load test for service *carts*. In this scenario, even only the service *carts* has to be deployed, because the service is not dependent on other service. Still, the load test provides as representative insights as targeting the *front-end*. By using performance stubs, the resource saving would be improved, because only the service under test has to be deployed.

3.2 Overview of Modularization Approaches

In the following, we introduce three different modularization approaches, which use different artifacts of the representative load test extraction and generation. Regardless of which representative load test approach is used, the approaches mainly are using a similar pipeline. Since ContinuITy also follows the introduced pipeline and includes some additional steps, the different approaches of modularization can be elaborated by analyzing the intermediate artifacts, which are illustrated in Figure 3.3. First, the system under test (SUT) is monitored and execution traces are collected (1). Second, the collected traces are used to extract session logs, which represent the requests per session (2). Next, the session logs are used to generate a workload model, which is represented as Markov chains (3). The used Workload Model is going to be used to transform the generated workload model into an executable load test (4). The load test is then applied on the SUT (5) (See Section 2.1.14).

Since the pipeline uses different intermediate artifacts, which are exchanged between the services of ContinuITy, it is reasonable to analyze the possibility of modularizing the different artifacts. One possible approach is modularizing the traces, which are going to be used to extract the session logs. Since a trace represents the request path through the system, a subset of all collected traces are containing requests which directly target the service or multiple services, which are going to be load-tested. The basic idea of this approach is to filter the traces, which directly target the SUT and remove all previous

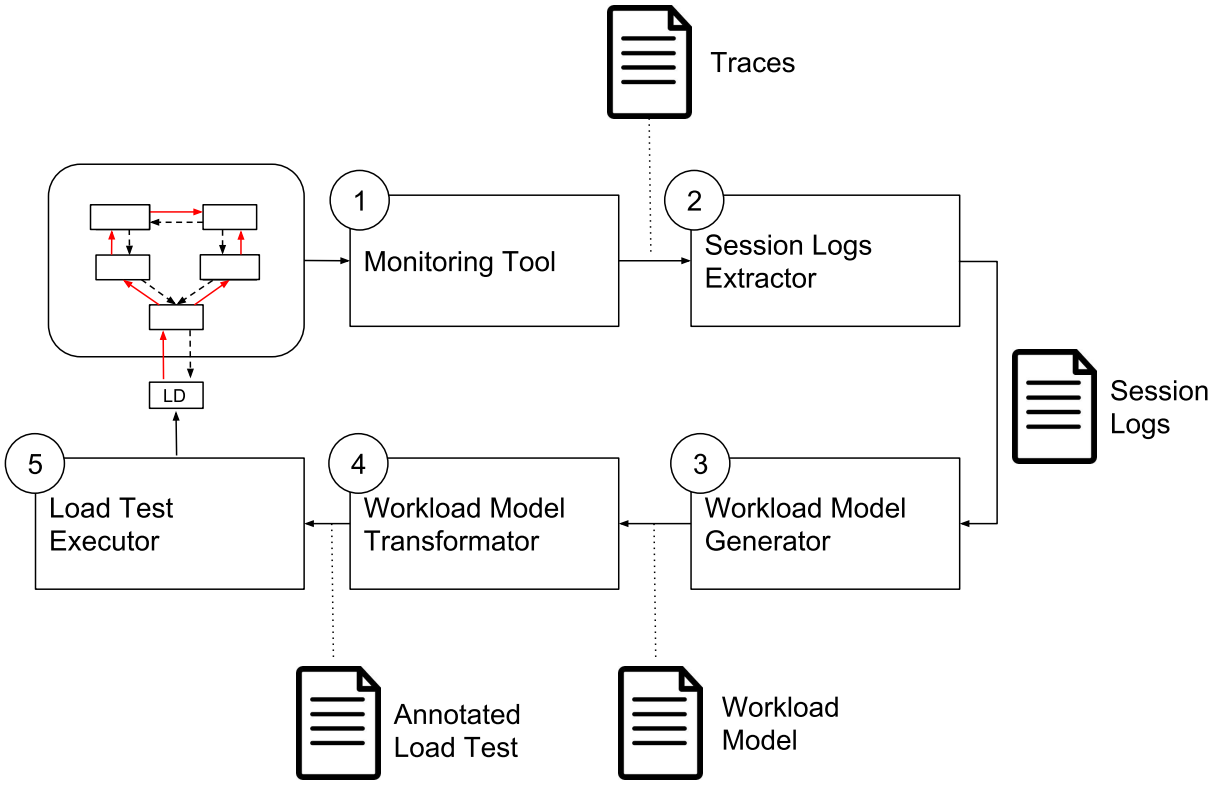


Figure 3.3: Steps for extracting and executing a representative load test

requests in this trace, such that the first request of a trace is directly target the dedicated services.

Another approach is to replace the contained requests of the session logs, which do not directly target the services under test. Analogously to the trace modularization, for each request nested requests are searched, which directly target the dedicated services. If all nested requests do not target one of the dedicated services, the request is deleted from the session log. As a result, the session logs containing modularized requests can be provided to the workload model generator.

In addition, it is also possible to modularize the workload model. In the context of this thesis, we base on Wessbas as workload model generator and Wessbas represents the user behavior by using Markov chains (see Section 2.1.9). Each state, which represents a request targeting not one of the desired services has to be replaced by one or more states, which represent the modularized requests. For each replacing state, all traces, which can be mapped to the corresponding request are used to generate modularized session logs by using the trace modularization approach. The modularized session logs are used to generate a Sub-Markov chain for the state to replace. Once each sub chain is merged with the original chain, the behavior model is modularized.

Although the annotated load test is another artifact of ContinuITy, the additional information provided by the annotation does not influence the replacement procedure. Hence, this approach can be excluded from further analysis and evaluation.

Since all approaches are based on the different artifacts and are used as input for the modularization approaches, we express the different artifacts and inputs as mathematical expressions. The expression can be used in further discussions and descriptions of the approach.

S defines all services of an application, whereas k is the amount of services:

$$S = \{s_1, \dots , s_k\}$$

S_{test} defines a set of services, which are going to be tested:

$$S_{test} \subset S$$

E_s defines all endpoints of a service s :

$$E_s = \{e_{s,1}, \dots, e_{s,o}\}$$

Each endpoint e maps to a certain target service s

$$s_{target} : e \to s$$

We defined E to be the union of all endpoints of each service:

$$E = \bigcup_{s \in S} E_s$$

We define R as the infinite set of all possible requests:

$$R = \{r_1, \dots r_l, \dots\}$$

A request r is defined by a timestamp t, a target endpoint $e_{s,l}$ and a duration Δ. The request parameters are abstracted, since the parameters do not have any influence on the modularization approach.

$$r_l = (t, e_{s,l}, \Delta), \quad e_{s,l} \in E, \Delta \to \mathbb{R}$$

A trace can be represented as a tree defined by the visited endpoints $E_{visited}$, the requests R, which target the endpoints, and the root request, which is the first request in the tree:

$$t_n = (E_{visited}, R, r_{root}), E_{visited} \subseteq E$$

A session log is defined by a start timestamp, an end timestamp and the corresponding requests, which were submitted as part of the session:

$$l_z = (t_{start}, t_{end}, R_{session}), \quad R_{session} \subset R$$

We define L as set of multiple session logs l.

$$L = \{l_1, \ldots, l_z\}$$

A Markov chain is represented as directed graph. The graph is defined by endpoints as vertices V and the edges, which are represented with the probability p and the normal distribution $\mathcal{N}(\mu, \sigma^2)$ of the think time. We define α as the initial state and ω as the exit state of the Markov chain.

$$\begin{aligned} M_i &= (V, p, \theta) \\ V &= E \cup \{\alpha, \omega\} \\ p &: V \times V \to [0, 1] \\ \theta &: V \times V \to \mathcal{N}(\mu, \sigma^2) \end{aligned}$$

A behavior mix defines a set of tuples, which contain a Markov chain M and the corresponding probability q of the Markov chain:

$$\begin{aligned} B &= (M = \{M_1, \ldots, M_w\}, q) \\ q &: M \to [0, 1] \end{aligned}$$

3.3 Modularization of Traces

One modularization approach modifies the traces. The traces are an intermediate artifact, which are used to extract the session logs. A trace is a tree of requests, which describes the path through the application. It is noticeable that the session logs extractor uses

the first HTTP request, which occurs in the trace as request, which is added to a certain session log. If the traces are modularized, all subsequent steps and artifacts remain the same, because the next processing steps are directly using the modularized requests. The core idea is to search for requests, which directly target the dedicated services in each trace. Once a request is found, the request will become the root request of the trace and all parent requests are removed. If none of the requests is fitting, the whole trace will be removed.

3.3.1 Input

The trace modularization approach gets monitored traces as input.

$$\begin{aligned} T &= \{t_1, \ldots, t_n\} \\ t_n &= (E_{visited}, R_{trace}, r_{root}), E_{visited} \in E \end{aligned}$$

Each trace consists of at least one request:

$$R_{trace} \neq \emptyset$$

3.3.2 Output

The trace modularization approach provides modified traces as result. If none of the input traces consists of a request, which directly targets one of the dedicated service, the resulting trace set can be empty. Each root request of the result set targets one of the dedicated services.

$$\begin{aligned} T' &= \{t'_1, \ldots, t'_m\}, \\ \forall (E_{visited}, R_{trace}, r_{root}) \in T' : r_{root} &= (t, e_{s,l}, \Delta) \Rightarrow s_{target}(e_{s,l}) \in S_{test} \end{aligned}$$

3.3.3 Algorithm

Algorithm 3.1 describes an algorithm for to modularizing the traces of the monitored system. First, the algorithm iterates over all input traces. For each input trace, the algorithm Algorithm 3.2 is called. The algorithm does a breadth-first search on the tree with the goal to find all subtraces/ requests, which are targeting one of the provided hostnames. The search stops, if a fitting subtrace is found. Since the session extraction

Algorithmus 3.1 Trace modularization

```
function MODULARIZETRACES(T, hostnames)
    T' = ∅
    for all t ∈ T do
        T' = T' ∪ GETMATCHINGSUBTRACES(t, hostnames)
    end for
    return T'
end function
```

Algorithmus 3.2 Get matching subtraces

```
function GETMATCHINGSUBTRACES(t, hostnames)
    traceChildren = t
    for all traceChild ∈ traceChildren do
        if GETHOSTAME(traceChild) ∈ hostnames then
            sessionId = GETSESSIONID(trace)
            SETSESSIONID(sessionId, traceChild)
            subtraces = subtraces ∪ traceChild
        else
            traceChildren = traceChildren ∪ GETCHILDREN(traceChild)
        end if
    end for
    return subtraces
end function
```

is dependent on the session information, the session Id of the original request is added to the subtrace. The algorithm can return zero, one ore multiple subtraces. Since the time complexity of a Breath-first search is the sum of the edges and vertices, the number of edges in a trace is always the number of vertices minus one:

$$\begin{aligned} General: &\quad \mathcal{O}(|V| + |E|) \\ Tree: &\quad \mathcal{O}(2 \cdot |V| - 1) \end{aligned}$$

The overall complexity of the approach is:

$$\mathcal{O}(|T| \cdot (2 \cdot |R_{trace}| - 1))$$

3.3.4 Example

Figure 3.4 illustrates the modularization of the traces. *Trace 1*, *Trace 2*, and *Trace 3* are defining the input traces. Each trace consists of multiple requests. The bold

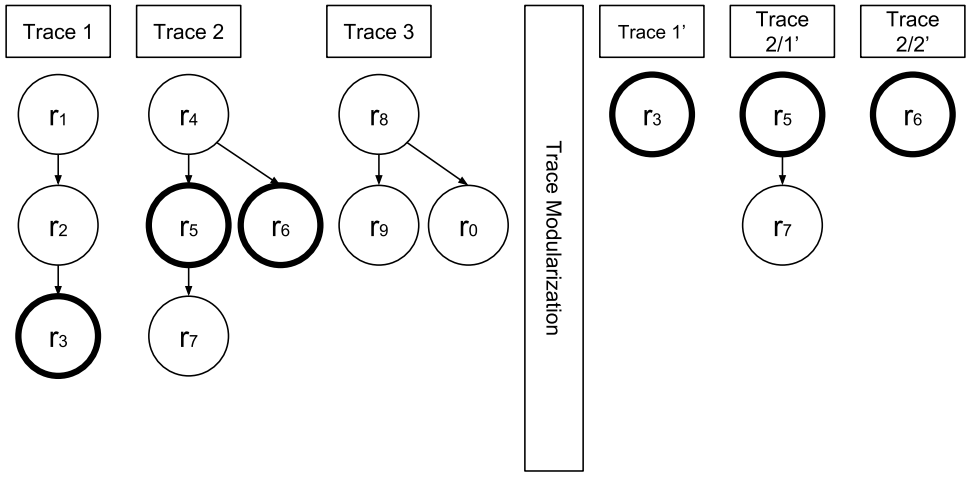


Figure 3.4: Example trace modularization

marked circles are defining requests, which are targeting the dedicated services. After the modularization, *r3* becomes the root request of *Trace 1'*. Since *r5* and *r6* are both targeting one of the dedicated services, both will be formed to a new *Trace 2/1'* and *Trace 2/2'*. In *Trace 3*, none of the requests are targeting one of the dedicated services. Hence the trace is completely removed after the modularization.

3.4 Replacement of Session Logs Requests

Another approach is to replace the requests of the generated session logs, with nested requests, which directly target the dedicated services. Analogously to the trace modularization approach, each request, which does not directly target the dedicated service, is going to be replaced by one or more services or completely removed. Once again, the raw traces are used to determine the replacing requests. Based on the characteristics such as the timestamp and the HTTP request properties, the request is mapped back to a certain request in a trace, since the trace context of the request is lost in the session log. However, the mapping between the session logs request and the corresponding request in the raw traces is ambiguous, because it cannot be excluded, that two requests have the completely same characteristics and the time resolution is to small in order to distinguish the timestamps. Once the session logs are modularized, the subsequent steps do not have to be adapted, because the requests, which have to be focused, are already encoded into the session logs.

3.4.1 Input

The session logs request replacement approach gets session logs L and the corresponding traces T as input:

$$\begin{aligned} L &= \{l_1, \ldots, l_z\} \\ l_z &= (t_{start}, t_{end}, R_{session}), R_{session} \subset R \\ T &= \{t_1, \ldots, t_n\} \\ t_n &= (E_{visited}, R_{trace}, r_{root}), E_{visited} \in E \end{aligned}$$

3.4.2 Output

The session logs request replacement approach provides modularized session logs as a result. If none nested requests can be found for each session log, the resulting set can be empty. The start time and the endtime of a session may differ, if requests are replaced or removed.

$$\begin{aligned} L' &= \{l'_1, \ldots, l'_y\} \\ L' &\neq \emptyset \\ l'_y &= (t'_{start}, t'_{end}, R'_{session}), R'_{session} \subset R, \\ \forall (t, e_{s,l}, \Delta) \in R'_{session} &: s_{target}(e_{s,l}) \in S_{test} \\ \forall (t'_{start}, t'_{end}, R'_{session}) &\in L', t'_{start} \geq t_{start}, t'_{end} \leq t_{end} \end{aligned}$$

3.4.3 Algorithm

Algorithm 3.3 describes the used algorithm in order to implement the replacement of the session log requests. First, the algorithm iterates over all existing requests of a given session log and checks, whether the current request is targeting one of the provided hostnames. If a request targets a different hostname, the request is removed.

After removing the request from the session log, the algorithm Algorithm 3.4 searches for the corresponding trace t of the request. This can be done based on the timestamp of the replacing request and the properties of the request, such as the HTTP method and the used parameters. However, it cannot be ruled out, that there might be more than

Algorithmus 3.3 Replacement of the session logs requests

```
function REPLACESESSIONLOGREQUESTS(l, hostnames, T)
    for all r ∈ l do
        if GETHOSTNAME(r) ∉ hostnames then
            REMOVEREQUEST(r, R_session)
            R' = GETREPLACINGREQUESTS(r, T, hostNames)
            R_session = R_session ∪ R'
            UPDATETIMESTAMPS(l)
        end if
    end for
    return l
end function
```

one matching traces, which occured at the same time. Although this is very unlikely, this is a weakness of the approach.

Since the extracted session logs are always containing the first request of a trace, only the root request r_{root} of a trace has to be compared with the replacing request. If the corresponding root request was found, the method GETMATCHINGSUBTRACES is called. Finally, each resulting $t_{subtrace}$ is transformed into a request. All requests are then added to the current session log. Since the first request and the last request of a certain session log might have changed, the start timestamp t_{start} and the end timestamp t_{end} are changed by executing the method UPDATETIMESTAMPS(l). Analogously to the trace modularization, the method GETMATCHINGSUBTRACES is a breadth-first search, which stops, if fitting $T_{subtraces}$ are found. Before calling GETMATCHINGSUBTRACES, the corresponding root request has to be found. This leads to the following time complexities:

$$\begin{aligned} \text{Find corresponding root request in traces:} \quad & \mathcal{O}(|T|) \\ \text{Find fitting nested request in trace:} \quad & \mathcal{O}(2 * |R_{trace}| - 1) \end{aligned}$$

The overall time complexity is the sum of both searches multiplied with the amount of session logs and corresponding session logs requests:

$$\mathcal{O}(|L| * |R_{session}| * ((|T|) + (2 * |R_{trace}| - 1)))$$

This approach can be optimized, if the session log requests are storing a reference to the corresponding trace. Although this hurts the independence of the artifacts it would

Algorithmus 3.4 Get replacing requests

```
function GETREPLACINGREQUESTS(r, T, hostNames)
    R' = ∅
    for all t ∈ T do
        r_root = GETROOTREQUEST(t)
        if REQUESTSAREMATCHING(r, r_root) then
            T_subtraces = GETMATCHINGSUBTRACES(t, hostnames)
            for all t_subtrace ∈ T_subtraces do
                R' = R' ∪ CONVERTTOREQUEST(t_subtrace)
            end for
            return R'
        end if
    end for
end function
```

increase the performance of the approach. I would not be necessary anymore to iterate over all traces. This would lead to the following improved time complexity:

$$\mathcal{O}(|L| * |R_{session}| * (2 * |R_{trace}| - 1))$$

3.4.4 Example

Figure 3.5 illustrates an exemplary request replacement procedure. Each log holds a session ID, a set of requests, and metadata, such as the start timestamp and end timestamp. In this scenario, only *service 1* is going to be load-tested. The bold requests are already targeting *service 1*. However, the request *A* of *Log1* is targeting *service 2*. The algorithm now searches for the corresponding root request by using the request characteristics. By using a Breath-first search through the trace, the approach finds the requests *B, C*, and *D*, which are targeting *service 1*. In this case, request *A* has to be replaced by *B, C*, and *D*. As a result, the modularized session logs only contain requests, which are directly targeting the desired services.

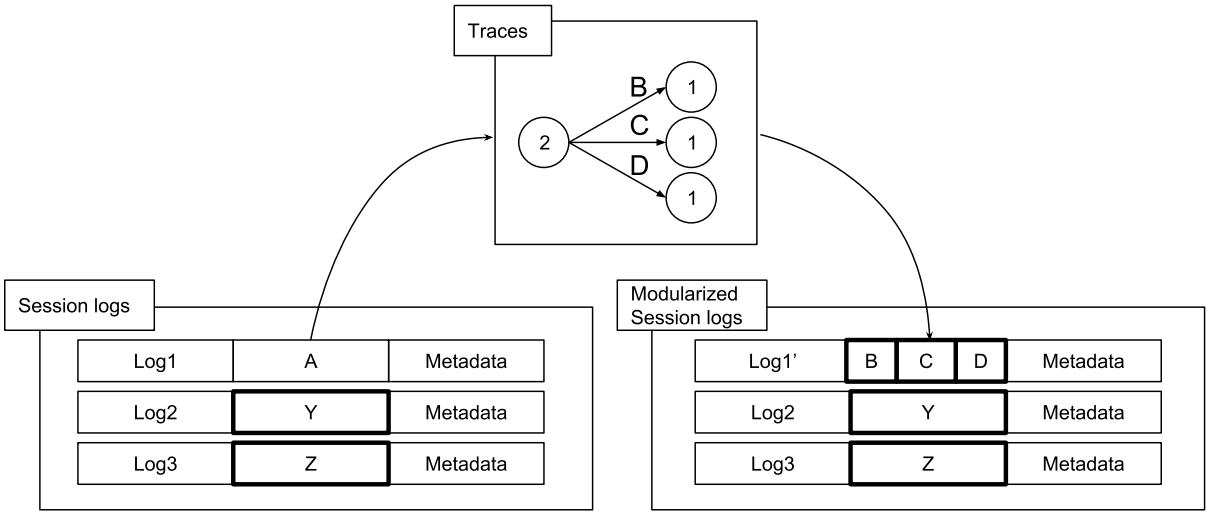


Figure 3.5: Generate modularized session logs by replacing entries

3.5 Modularization of Workload Models

Instead of manipulating the trace records or the session logs, it is also possible to manipulate the generated workload model. Other than providing a set of requests, the WESSBAS model provides a detailed execution logic based on stochastic models. The workload model, represented in the WESSBAS DSL is generated in two separate steps. First, each session is transformed into a behavior model. All behavior models are clustered based on the their similarity. For each cluster, a representative behavior model is calculated and a certain propability is calculated. All generated behavior models are called *behavior mix*. Second, the behavior mix is converted and enriched to the WESSBAS DSL workload model.

A behavior model is defined by a Markov chain. Each Markov state represents a request to the application. Besides representing the probability, the transitions define the normal distribution of the think time between the different states.

Figure 3.6 shows an example for a WESSBAS behavior model. Each transition is defined by three values: the first value represents the *probability*, the second value represents the *mean* think time, and the third value represents the *deviation* of the think time. In order to simplify the examples, the standard deviation is always set to 0.

The process of modularization is done between the behavior model extraction and the workload model generation. Therefore, the behavior mix, the session logs, the traces, and the services under test have to be provided. For each behavior model, the approach identifies all Markov states which need to be modularized. If the Markov state

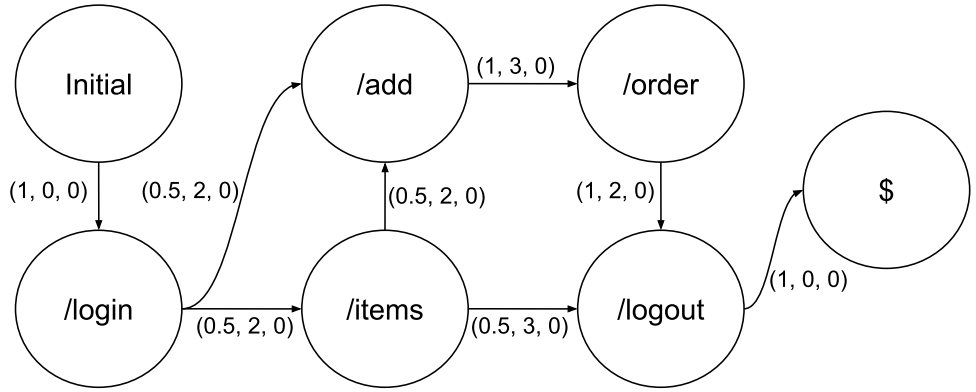


Figure 3.6: Wessbas behavior model example

represents an endpoint, which belongs to the wrong service, the Markov state has to be replaced or even removed. For each identified Markov state, all corresponding traces containing the same request are collected. Since the requests have to be replaced with the dependent requests targeting the desired services, the trace modularization approach (see Section 3.3) can be reused in order to get modularized session logs. Again, the behavior model extraction of WESSBAS is used in order to convert the modularized session logs into a behavior model for each identified Markov state. If a certain endpoint of a Markov state does not invoke any fitting request, no behavior model is provided and the Markov state has to be skipped. Else, the generated Sub-Markov chain for a certain Markov state has to be merged into the existing Markov chain.

For skipping Markov states and replacing a Markov state with an Sub-Markov chain, the transitions between the Markov chains have to be adapted. If a Markov state is skipped, the incoming transitions have to be merged with the outgoing transitions of a Markov chain. Since the duration of the removed state is not represented anymore in the Markov chain, the normal distribution of the duration is added to the think time of all merged transitions.

Concerning the merge of a Sub-Markov chain into an existing Markov chain, the incoming transitions of the replacing state have to be merged with the outgoing transitions of the initial state of the Sub-Markov chain. In addition, all incoming transitions of the exit state of the Sub-Markov chain have to be merged with the outgoing transitions of the replacing state.

Analogously to the skip operation, the shorter execution time of the Sub-Markov states has to be compensated by adding the preprocessing time normal distribution to the incoming transitions of a modularized Markov state and the postprocessing time normal distribution to the outgoing transitions.

- Preprocessing Time: The preprocessing time is the time range from the start of the original request until the start of the modularized request:

$$t_{pre} = modRequest.startTimeStamp - rootRequest.startTimeStamp$$

- Postprocessing Time: The postprocessing time is the time range from the end time of the modularized request to the end time of the original request:

$$t_{post} = rootRequest.exitTimeStamp - modRequest.exitTimeStamp$$

3.5.1 Input

The workload model modularization approach gets a Markov chain and traces as input:

$$\begin{aligned} M_i &= (V, p, \theta) \\ V &= E \cup \{\alpha, \omega\} \\ p &: V \times V \to [0, 1] \\ \theta &: V \times V \to \mathcal{N}(\mu, \sigma^2) \\ T &= \{t_1, \ldots, t_n\} \\ t_n &= (E_{visited}, R_{trace}, r_{root}), E_{visited} \in E \end{aligned}$$

3.5.2 Output

The algorithm provides a modularized Markov chain. If the Markov states cannot be replaced, the set of Markov states V can be empty. Each Markov state only presents an endpoint, which belongs to one of the dedicated services.

$$\begin{aligned} M_i' &= (V', p', \theta') \\ V' &= E \cup \{\alpha, \omega\} \\ p' &: V' \times V' \to [0, 1] \\ \theta' &: V' \times V' \to \mathcal{N}(\mu, \sigma^2) \\ \forall v_x &\in V : s_{target,x} \in S_{test} \end{aligned}$$

In addition, if a Markov state v_x is removed, each probability transition coming from v_i, which targeted the removed state, has to be multiplied with the outgoing probability transition, targeting v_j.

$$\begin{aligned} \forall v_x \forall v_i \forall v_j \quad &: \quad v_x \in V \wedge v_x \notin V' \\ \wedge v_i \quad &\in \quad V, V' \wedge p(v_i, v_x) \neq 0 \\ \wedge v_j \in V, V' \wedge p(v_x, v_j) \neq 0 \quad &\Rightarrow \quad p'(v_i, v_j) = p(v_i, v_x) \cdot p(v_x, v_j) \end{aligned}$$

Analogously to the propability, the think time transitions have to be merged. Since the think time is represented as Gaussian distribution, the two Gaussians have to be summarized. Since the original request is not executed and the think time between two modularized requests would decrease, the duration of the removed request Δ has to be added to the think time distribution.

$$\begin{aligned} \forall v_x \forall v_i \forall v_j \quad &: \quad v_x \in V \wedge v_x \notin V' \\ \wedge v_i \quad &\in \quad V, V' \wedge p(v_i, v_x) \neq 0 \\ \wedge v_j \in V, V' \wedge p(v_x, v_j) \neq 0 \quad &\Rightarrow \quad \theta'(v_i, v_j) = \mathcal{N}(\mu_{i,x} + \mu_{x,j} + \overline{\Delta},\, \sigma_{i,x}^2 + \sigma_{x,j}^2 + \sigma_{\Delta}^2) \end{aligned}$$

If a Markov state is replaced, the incoming probability transitions of the replacing Markov State v_z have to multiplied with each outgoing transition of the initial state $\alpha_{sub,z}$ of the modularized Sub-Markov chain $M_{sub,z}$.

$$\begin{aligned} \forall v_z \forall v_{sub,z} \forall v_s \quad &: \quad v_z \in V \wedge v_z \notin V' \\ \wedge v_{sub,z} \quad &\in \quad V_{sub,z} \wedge p(\alpha_{sub,z}, v_{sub,z}) \neq 0 \\ \wedge v_s \in V, V' \wedge p(v_s, v_z) \neq 0 \quad &\Rightarrow \quad p'(v_s, v_{sub,z}) = p(v_s, v_z) \cdot p(\alpha_{sub,z}, v_{sub,z}) \end{aligned}$$

Analogously, all incoming propability transitions of the exit state $\omega_{sub,z}$ have to be merged with all outgoing transitions of v_z:

$$\begin{aligned} \forall v_z \forall v_{sub,z} \forall v_t \quad &: \quad v_z \in V \wedge v_z \notin V' \\ \wedge v_{sub,z} \quad &\in \quad V_{sub,z} \wedge p(v_{sub,z}, \omega_{sub,z}) \neq 0 \\ \wedge v_t \in V, V' \wedge p(v_z, v_t) \neq 0 \quad &\Rightarrow \quad p'(v_{sub,z}, v_t) = p(v_{sub,z}, \omega_{sub,z}) \cdot p(v_z, v_t) \end{aligned}$$

The same conditions are stated for the think time transitions: All incoming think time transitions of the replacing Markov State v_z have to summed with each outgoing transition of the initial state $\alpha_{sub,z}$ of the modularized Sub-Markov chain $M_{sub,z}$. Since the duration of the request is shorter than the original request, the pre processing time distribution Δ_{pre} has to be added to the think time distribution.

$$\begin{aligned}
\forall v_z \forall v_{sub,z} \forall v_s &: v_z \in V \wedge v_z \notin V' \\
\wedge v_{sub,z} &\in V_{sub,z} \wedge p(\alpha_{sub,z}, v_{sub,z}) \neq 0 \\
\wedge v_s &\in V, V' \wedge p(v_s, v_z) \neq 0 \\
\Rightarrow \theta'(v_s, v_{sub,z}) &= \mathcal{N}(\mu_{s,z} + \mu_{\alpha_{sub,z}, v_{sub,z}} + \overline{\Delta_{pre}},\ \sigma^2_{s,z} + \sigma^2_{\alpha_{sub,z}, v_{sub,z}} + \sigma^2_{\Delta_{pre}})
\end{aligned}$$

Analogously, all incoming think time transitions of the exit state $\omega_{sub,z}$ have to be merged with all outgoing transitions of v_z. Since the duration of the request is shorter than the original request, the post processing time distribution Δ_{post} has to be added to the think time distribution.

$$\begin{aligned}
\forall v_z \forall v_{sub,z} \forall v_t &: v_z \in V \wedge v_z \notin V' \\
\wedge v_{sub,z} &\in V_{sub,z} \wedge p(v_{sub,z}, \omega_{sub,z}) \neq 0 \\
\wedge v_t &\in V, V' \wedge p(v_z, v_t) \neq 0 \\
\rightarrow \theta'(v_{sub,z}, v_t) &= \mathcal{N}(\mu_{v_{sub,z}, \omega_{sub,z}} + \mu_{z,t} + \overline{\Delta_{post}},\ \sigma^2_{v_{sub,z}, \omega_{sub,z}} + \sigma^2_{z,t} + \sigma^2_{\Delta_{post}})
\end{aligned}$$

3.5.3 Algorithm

Algorithm 3.5 describes the used algorithm, in order to implement the modularization of the workload model. The algorithm uses the behavior mix B, session logs L, $hostnames$ and traces t as input. All Markov states representing a request which does not target one of the dedicated hostnames is added to $markovStatesToReplace$. Since a certain behavior model only covers the behavior of certain sessions, the traces have to be filtered based on the session ID. For each behavior model, the method REPLACESTATES is called. The manipulated behavior mix will then be transformed to a WESSBAS DSL model.

For each Markov state to replace, the algorithm Algorithm 3.6 determines the corresponding traces. In order to generate a modularized behavior model, session logs are necessary. At this point, the algorithm reuses the approach of modularizing session logs and generates modularized traces for a certain Markov state. If session logs can be generated, a modularized Markov chain for the Markov state to replace is created

Algorithmus 3.5 Modularization of the workload model

```
function MODULARIZE(B, L, hostnames, T)
    for all M ∈ B do
        statesToReplace = ∅
        for all v ∈ V do
            if GETHOSTNAMEOFREQUEST(v) ⊈ hostnames then
                statesToReplace ∪ v
            end if
        end for
        T_filtered = FILTERTRACESBYSESSIONID(B, T)
        REPLACESTATES(statesToReplace, M, T_filtered, hostnames)
    end for
    return GENERATEWESSBASDSL(B, L)
end function
```

by using the WESSBAS behavior model extractor. The method MERGEMODELS updates the transitions by using the conditions defined in Section 3.5.2. Else, the corresponding traces of the Markov state do not have any indirect calls to the dedicated services and the Markov state is going to be skipped by calling the method SKIPMARKOVSTATE.

The time complexity of the workload model modularization approach is mainly influenced by the following parameters. For each behavior model and each Markov State, the traces are filtered, the modularized session logs are generated, a new Markov Chain is generated, and the transitions are updated. Since Wessbas uses the cluster algorithm X-Means, the time complexity of generating the Sub-Markov chain can be approximated with the following complexity, where d is the number of dimensions and k is the number of clusters in the cluster:

$$\mathcal{O}(n^{dk+1})$$

Transferring this to the WESSBAS context, the number of clusters is the number of different behavior models, the number of cluster element is the number of session logs and the dimension is the number of all existing Sub- Markov states $|M_{sub}|$.

$$\mathcal{O}(|L|^{|V_{sub}|\cdot|B|+1})$$

This leads to the following time complexity:

$$\mathcal{O}(|B|\cdot|V|\cdot(|T|\cdot p_B + \underbrace{(n^{|M_{sub}|\cdot|B|+1})}_{\text{wessbas clustering}} + \overbrace{|V\cdot V|}^{\text{transitions}} + \underbrace{(|T|\cdot(2\cdot|R_{trace}|-1))}_{\text{trace modularization}}))$$

Algorithmus 3.6 Replace Markov states

```
function REPLACESTATES(statesToReplace, M, T, hostnames)
    for all v ∈ statesToReplace do
        matchingTraces = GETMATCHINGTRACES(v, T, hostnames)
        L_modularized = GETMODULARIZEDSESSIONLOGS(matchingTraces, hostnames)
        if L_modularized ≠ ∅ then
            M_sub = EXTRACTBEHAVIORMODEL(L_modularized)
            MERGEMODELS(M, v, M_sub)
        else
            SKIPBEHAVIORSTATE(v, M)
        end if
    end for
end function
```

3.5.4 Example

In the following, we introduce the skipping procedure and the merge procedure by using an example.

Skipping a Markov State

Figure 3.7 shows how the Markov state */items* of Figure 3.6 is skipped: In order to have a valid Markov chain after deleting the Markov state, the incoming transitions of the state */items* have to be merged with the outgoing transitions. The probabilities are multiplied, the means and the variance are summed up. Since in this particular case, the standard deviation is used, the aggregated deviation is computed by $\sigma_{12} = \sqrt{\sigma_1^2 + \sigma_2^2}$. Because the request of the original Markov state be will not executed, the execution time of the Markov chain is shorter. In order to compensate the decrease of the execution time, the algorithm adds the normal distribution of the response time of the request */items* to the transition. In this case, the response time is 1, which is represented with the summand *+1*.

However, after the merge of the incoming and outgoing transitions, */login* has two separate transitions to */add*. In order to join transitions with the same target state, the probabilities of both transitions have to be summed up and a weighted mean of both distributions is computed (see Section 2.1.9). The results of the skip operation is shown in Figure 3.8.

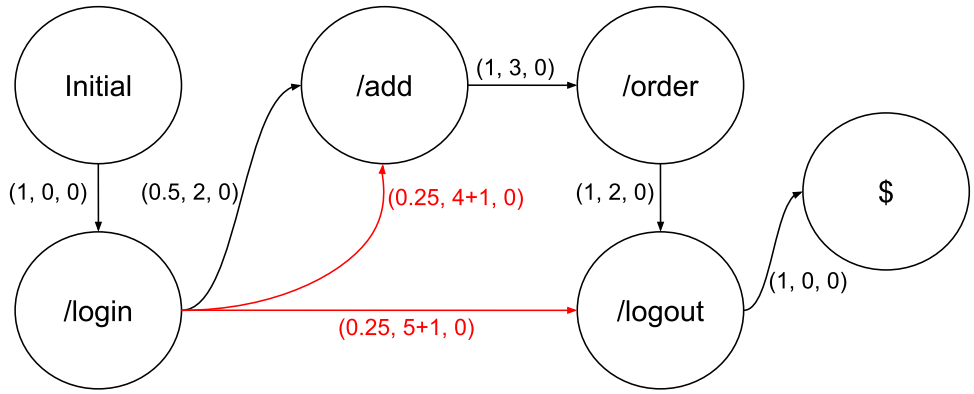


Figure 3.7: Skip Markov state example - step 1

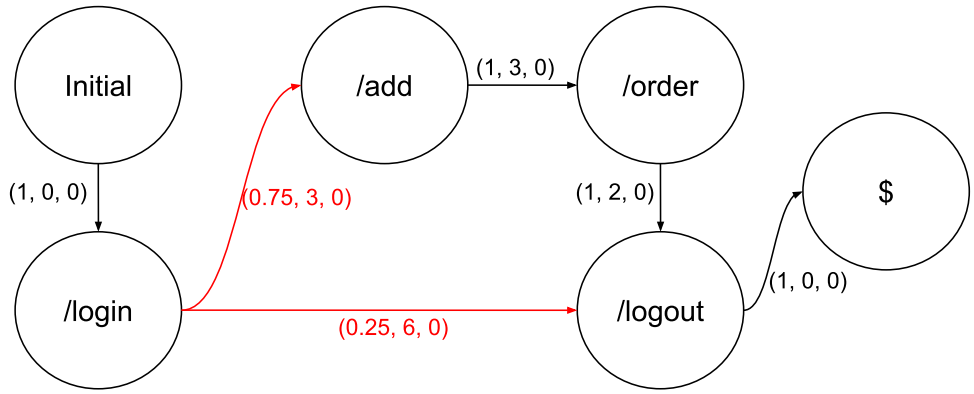


Figure 3.8: Skip Markov state example - step 2

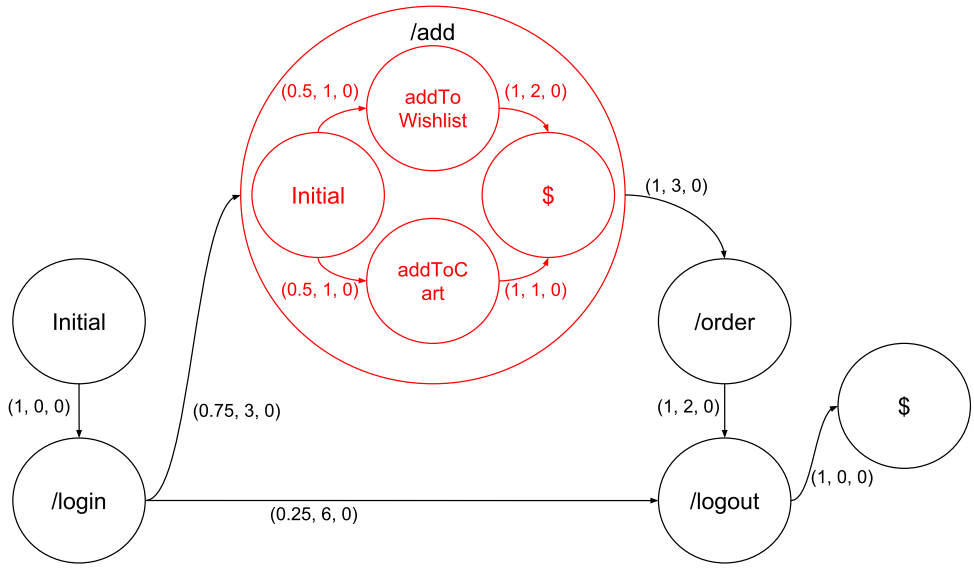


Figure 3.9: Replace state with modularized chain - step 1

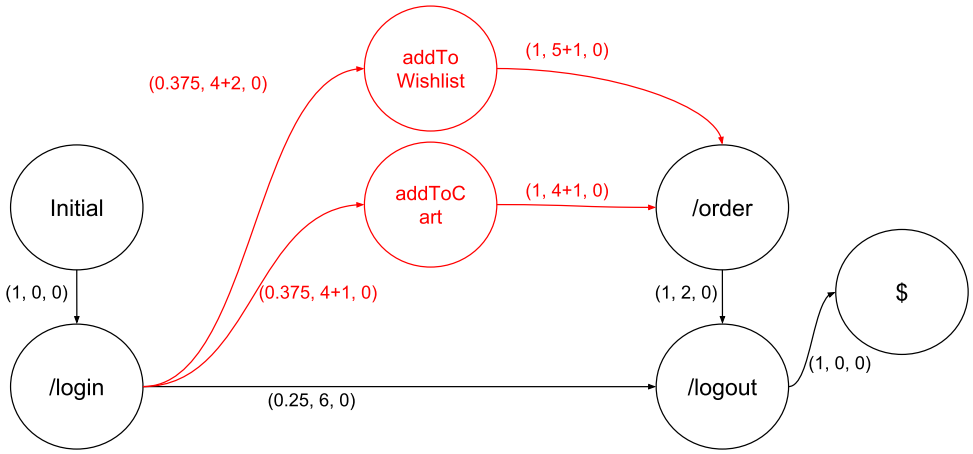


Figure 3.10: Replace Markov state with modularized Markov chain - step 2

Replace a Markov state with a modularized Markov chain

However, if a modularized behavior model is available for a certain Markov state, the modularized behavior model, represented as another chain, has to be merged into the Markov chain of the non-modularized behavior model. Figure 3.9 shows the modularized behavior model for Markov state */add*. In order to merge the modularized Markov chain into the non-modularized Markov chain, the incoming transitions of */login* have to be merged with the outgoing transitions of the modularized intial state. In this case, the Markov state */login* has transitions to *addToWishlist* and *addToCart*. In addition, all transitions targeting the modularized exit state have to be merged with the outgoing transitions of */login*. In this example, *addToWishlist* and *addToCart* have a transition to */order*.

For each modularized Markov state, a normal distribution of the pre processing time and the post processing time is calculated and added to the corresponding transitions. Figure 3.10 illustrates the preprocessing and postprocessing time by the summands.

However, this approach depends on the used workload model generator and might not work for every workload model generator.

3.6 Approach Discussion

In the following, the introduced approaches are compared and discussed. In Section 3.6.1, it is shown, that the trace modularization and the session logs modularization provide semantically equivalent results. In Section 3.6.2 the limitations, of the trace modularization and the session log request replacement is discussed.

3.6.1 Semantical Equivalence of Trace Modularization and Session Logs Request Replacement

In the following, we show that both approaches lead to a semantically equivalent load test, because the session logs and the created workload model will be the same.

Since the conversion of the traces to the session logs ($T \rightarrow L$) does only extract the relevant root requests of each trace, the used modularized requests are the same and the used breadth-first search is the same, the resulting session logs are semantically equivalent.

$$\forall r_{root} \in T : r_{root} \in R_{session}$$

It is irrelevant, whether the traces are first modularized and then extracted or the other way round.

In addition, the time complexity of both approaches is equivalent, because the the number of sessions multiplied with the corresponding request is the number of traces ($|R_{session}| \cdot |L| = |T|$):

$$\underbrace{\mathcal{O}(|T| \cdot (2 \cdot |R_{trace}| - 1))}_{\text{trace modularization}} \equiv \underbrace{\mathcal{O}(|L| \cdot |R_{session}| * (2 \cdot |R_{trace}| - 1))}_{\text{optimized session logs request replacement}}$$

To sum up, the concluding session logs are semantically equivalent.

3.6.2 Limitations of Trace Modularization and Session Logs Request Replacement

If the traces are modularized or the session log requests are replaced, the context of the requests, such as the parent call is removed. Hence, the context of a request cannot be represented by the Markov chain, which only contains the direct requests. However, the context or the state of previous services might have an influence. The following example aims to illustrate such a case. Figure 3.11 shows the communication of two services, which are used by two different users. The black arrows represent requests and the dashed arrows represent responses. Parameters, which are sent with a request are represented as letters in the brackets next to the requests. Request *B* and request *C* are dependent on the response of the previous requests *A* and *B*. The order of the requests must not be changed by the load test. In this scenario, the session logs would

contain two entries, because two different users with two different session IDs invoke requests. Request *X* is placed in the first session log of *user 1* and in the session log of *user 2*. Although both users are executing the same requests, the request *A* is only executed once. *Service 1* can reuse the response *a* in the request of *user 2*. Concerning a conventional generated load test, the Markov chains do not allow any illegal transitions, because the order of the first and second request *X* does not matter.

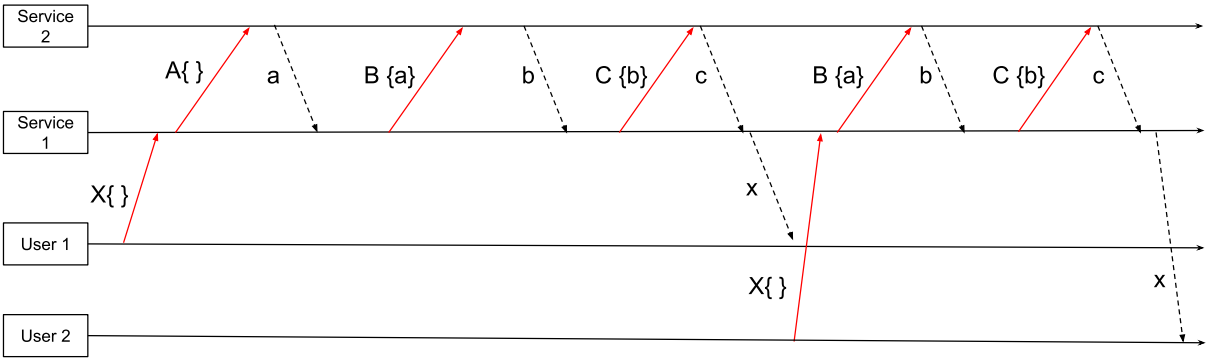


Figure 3.11: Request scenario with two different users

However, if only *Service 2* is deployed, *A, B*, and *C* would be represented as separate requests of the first session log and *B* and *C* would be separate requests of the second session log. Hence, the workload generator will generate a Markov chain, which allows to initially invoke request *B*. This may cause an error, because the response of request *A* is needed in request *B*.

Chapter 4

Modifications to ContinuITy

Although the ContinuITy platform already is capable of automatically generating, annotating and executing representative load tests, the implemented modularization approaches required some conceptional changes in parts of the pipeline. In the following, the changes are described and explained on a conceptional level. In Section 4.1 the *JsonInput* is used. Section 4.2 describes the introduced metrics exporter of the *continuity.jmeter* service.

4.1 Adaptions of the Input Data and Properties Annotation Model

Regardless of the used modularization approach, the generated modularized load test needs to be annotated, in order to have an error-free execution. However by now, ContinuITy only manages an application model and the corresponding annotation model of the whole application. Since the modularized load test generation needs to know the endpoints of the services under test, the developer also has to provide an application model and an annotation model for each service. After a modularized load test is generated, the annotation models of the corresponding services are applied.

The annotation model basically provides the possibility of defining different inputs. The inputs can be then associated with an input parameter of an endpoint, which is defined in the application model. There are existing different types of inputs:

- CSV input: The input of a certain parameter is extracted from a CSV file.
- Direct list input: The input of a certain parameter is randomly chosen from a provided list of values.

Listing 4.1 Example json input annotation

```
- !<json>
 &Input_paymentPaymentAuthorizationUsingPOST_body_BODY type: object
 items:
 - !<json>
  name: customer
  type: object
  items:
  - !<json>
   name: addresses
   type: array
   items:
   - !<json>
    type: string
    input: *Input_addressIds_BODY_PART
   - !<json>
    type: string
    input: *Input_addressIds_BODY_PART
 - !<json>
  name: amount
  type: number
  input: *Input_Amount_BODY_PART
```

- Extracted input: The input of a certain parameter is extracted from a response of a defined request. This can be done by defining a Regex expression or a Regex path.
- Counter input: The input of a certain parameter is defined by a counter which will be continuously incremented, until a predefined maximum is reached.

In common microservice architectures, lightweight protocols such as HTTP are used for communication. In order to exchange objects, JSON is a common standard. However, it is very difficult to represent a JSON object by the existing input types. Hence, in order to be able to annotate requests, which consist of a JSON body, we introduce an additional JSON input type. A JSON input consists of the following attributes:

- type: A JSON input can be a STRING, NUMBER, OBJECT or ARRAY. This field is required.
- name: A JSON input can have a name. This field is optional.
- input: A JSON input can also hold a direct value in form of an arbitrary value. For instance, the input field can refer to a Direct list input. This field is optional.
- items: A JSON input can additionally hold a list of nested JSON inputs. If a JSON input is type of OBJECT or ARRAY, this field is required.

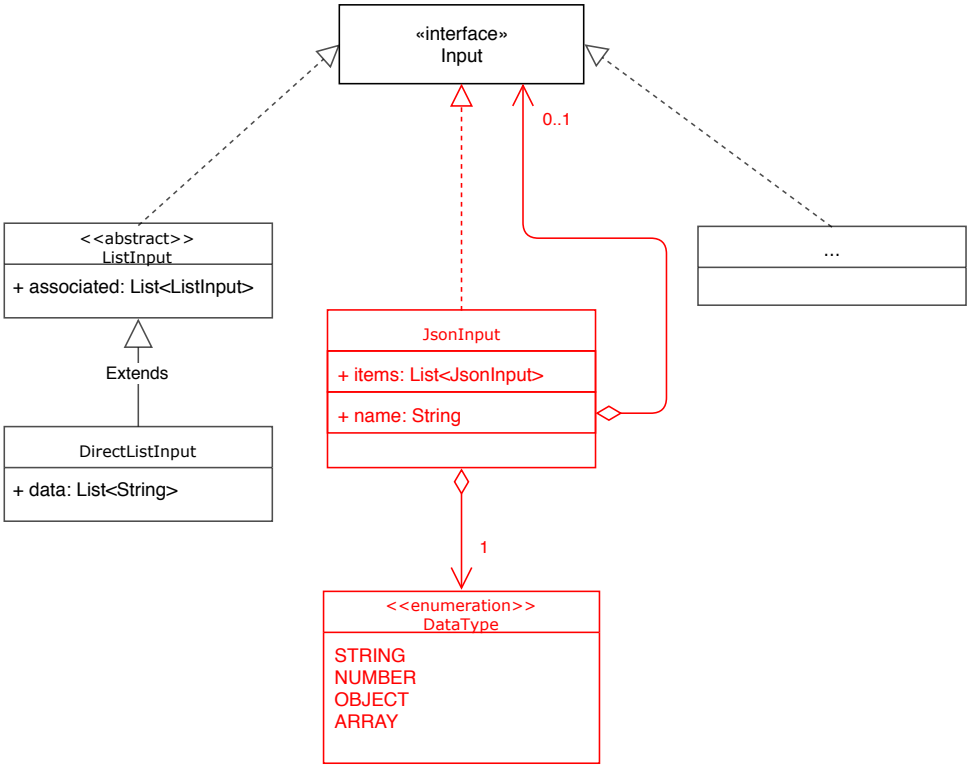


Figure 4.1: Class diagram of the annotation model extension

The structure of the JSON input is represented in Figure 4.1.

The JSON representation is based on the JSON data type representation of the Swagger OpenAPI Specification [Sma18b]. Since the Swagger OpenAPI specification is only a meta model of concrete data, we adapted the model, in order to handle real JSON object and JSON array instances. For instance whereas in the Swagger OpenAPI Specification the field *items* specifies the type of the objects, which are consisted in an array, the JSON input annotation holds concrete items in the field. Listing 4.1 shows an example of a JSON input annotation of the type *object*. The object contains the two fields *customer* and *amount*. Amount is of the type *number* and references another input definition. The field *customer* is of the type *array* and consists of the two different items of the type *string*. Both are referencing another input definition.

4.2 Metrics Export of continuity.jmeter

Considering the process of load testing it is also essential to have an insight into the performance of the load testing framework itself. In order to support monitoring,

a Prometheus [Piv18a] metrics exporter is integrated into the JMeter microservice. Arbitrary metrics such as memory usage and the CPU utilization are published at an additional endpoint and can be scraped by the Prometheus server.

Chapter 5

Implementation

In order to evaluate the developed approaches (see Chapter 3), we implemented them into the current code base of ContinuITy. Only the trace modularization (see Section 3.3) and workload model modularization (Section 3.5) are implemented, because the other approaches produce artifacts, which are semantically equivalent. Section 5.1 introduces the implementation of ContinuITy and explains the given technical constrains of the project. Section 5.2 presents the modifications, that are made in the context of the modularization, regardless of a specific modularization approach. Because each modularization approach requires changes in specific microservices, Section 5.3 and Section 5.4 give an insight on how the different approaches are implemented.

5.1 Overview

Since the implementation of the modularization approaches is done in the context of the ContinuITy research project, the implementation of the approaches is technically constrained. ContinuITy is implemented in Java and published on GitHub[1]. ContinuITy itself is subdivided into different microservices. Each microservice is responsible for a certain processing step of the ContinuITy pipeline. At the state before the start of this thesis, ContinuITy consists of the following services:

- *continuity.session.logs*: This service converts traces provided in the OPEN.xtrace format into session logs.

[1] https://github.com/ContinuITy-Project/ContinuITy

- *continuity.idpa.application*: This service manages the application model of the systems under test. The application model defines the different entrypoints of the application.
- *continuity.idpa.annotation*: This service manages the annotation model of the systems under test.
- *continuity.wessbas*: This service extracts the user behavior model, provides the extracted results formatted in the WESSBAS DSL and transforms the workload model into a JMeter test.
- *continuity.jmeter*: This services executes the transformed load test of the wessbas service and provides the results of the load test.
- *continuity.benchflow*: Analogously to the JMeter microservice, this service generates benchflow load tests.
- *continuity.request.logs*: This service generates a JMeter load test, which is generated based on request rates of the entry points. The generated JMeter load test uses an open workload and the intensity is defined by the amount of requests per second for each endpoint under test.
- *continuity.forecast*: This service is able to forecast the workload intensity for a context-specific event in the future.
- *continuity.orchestrator*: This service orchestrates the whole pipeline and triggers all other microservices, since the microservices should be as much independent as possible. The orchestrator accepts an order, which defines the configuration of the pipeline such as the used modularization approach and the used monitoring data type.

All services are using the Spring Framework [Piv18c] and the HTTP endpoints are documented using Swagger [Sma18a]. Figure 5.1 shows the architecture of ContinuITy. All service are coordinated by the *continuity.orchestrator* service. *continuity.orchestrator* accepts orders, which are transformed to tasks and sent to the responsible services via the message queue. Each service provides a task report, after a certain task has failed or successfully finished. The artifacts of the services are published via HTTP and are called by other services. The colored HTTP connection between *continuity.session.logs* and *continuity.wessbas* is needed by the workload model modularization approach.

5.1.1 Communication

All microservices are either communicating over HTTP or over a message broker. ContinuITy uses RabbitMQ [Piv18b] as message broker. An order can be either sent via HTTP

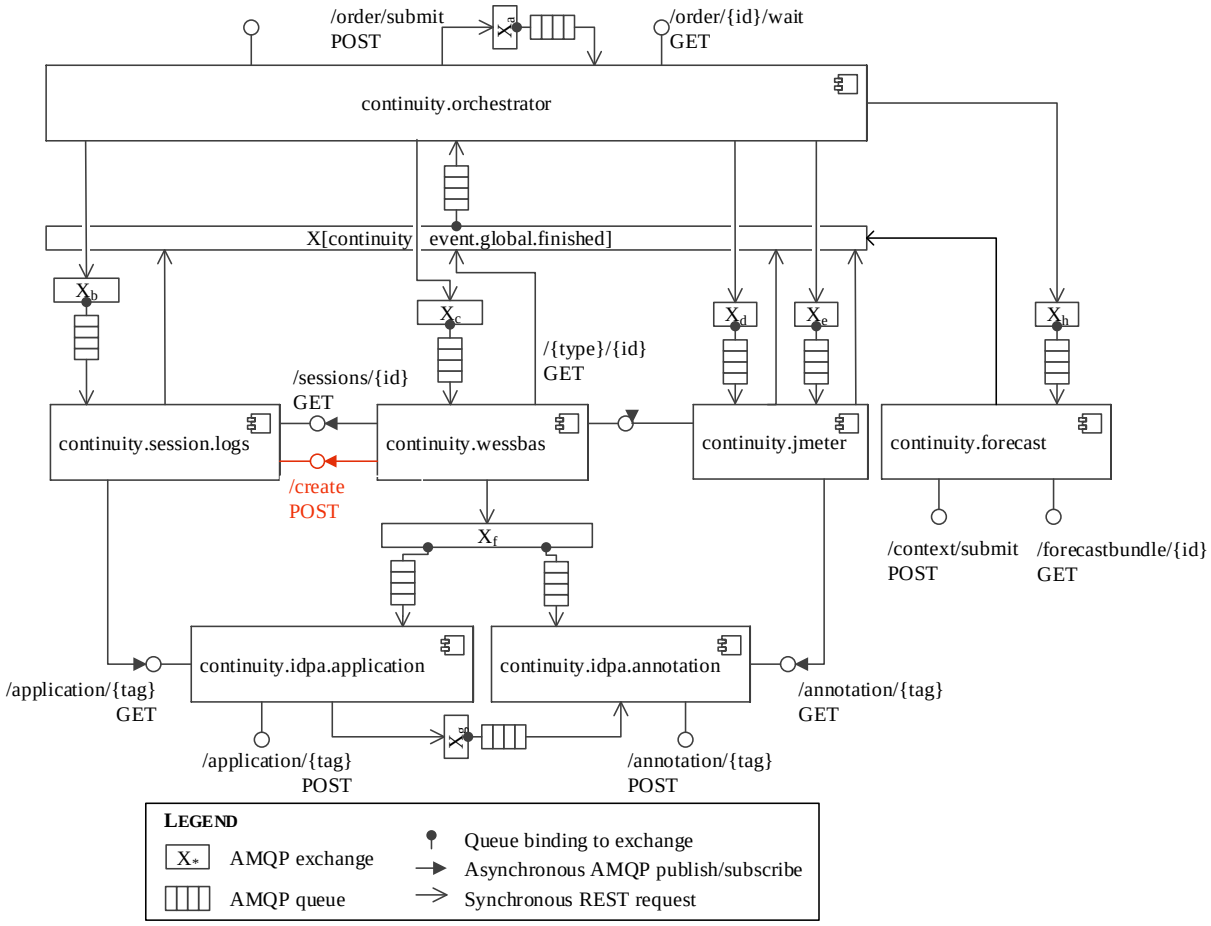


Figure 5.1: Architecture overview (based on [com18])

to the *continuity.orchestrator* or ContinuITy also provides a command line interface in order to manage orders and IDPA artifacts.

5.1.2 Order Creation

An order defines the process of the whole ContinuITy pipeline. The following attributes can be configured:

- Order Goal: The goal defines the purpose of a run of the pipeline. The user can choose between the following goals:
  - *create-session-logs*: If this goal is defined, the pipeline stops after generating the session logs.

  - *create-behavior-mix*: If this goal is defined, the pipeline stops after generating the different behavior models. The workload model such as the WESSBAS DSL is not generated.
  - *create-forecast*: This goal defines that the workload intensity is forecasted by the *continuity.forecast* microservice.
  - *create-workload-model*: If this goal is defined, the pipeline stops after generating a complete workload model. Currently, a WESSBAS DSL model is generated.
  - *create-load-test*: If this goal is defined, the workload model is transformed into an executable load test.
  - *execute-load-test*: If this goal is defined, the generated load test is executed. Depending on the defined load test, either the *continuity.jmeter* or the *continuity.benchflow* service is used to execute the load test.

- *mode*: Since some of the goals depend on the execution of other goals, ContinuITy currently defines three different execution modes, which declare the dependency and order of different goals. Currently, three different cycles are defined:
  - *past-sessions*: This mode defines the following dependencies:

    create-session-logs → create-workload-model → create-load-test → execute-load-test
  - *past-requests*: This mode defines the following dependencies:

    create-workload-model → create-workload-model → create-load-test → execute-load-test
  - *forecasted-workload*: This mode defines the following dependencies:

    create-session-logs→ create-behavior-mix → create-forecast → create-workload-model → create-load-test → execute-load-test
- *testing-context*: The run of the pipeline can be labeled with a number of tags. The links of already executed orders with the same testing-context can be reused.
- *options*: In *options*, arbitrary configurations such as the type of the load test and the duration of the load test can be defined.
- *source*: In source, the different input artifacts can be defined. For instance, the link to the trace data provider have to be defined here. In addition, it is possible to reference artifacts from all stages of the pipeline. For instance, if the user wants to reuse the workload model of a previous run, the link to the workload model can be simply defined in *source*.

Listing 5.1 Example order configuration file

```
---
goal: execute-load-test
mode: past-sessions
tag: sock-shop
options:
  duration: 60
  rampup: 1
  workload-model-type: wessbas
  load-test-type: jmeter
  num-users: 1
source:
  measurement-data:
    link: http://trace-provider/getTraces
    timestamp: 2018-12-21T00-00-00-000Z
    type: open-xtrace
modularization:
  services:
    sock-shop-orders: orders
    sock-shop-carts: carts
  modularization-approach: trace
```

- *forecast-input*: This part of the order defines configuration of the *continuity.forecast* microservice application.
- *modularization*: Since the proposed approach of the thesis also needs to be configured, it is essential to have a separate configuration section. In *modularization*, the desired services under test can be defined. In addition, it has to be defined, which modularization approach is used by using the keywords *trace* or *workload model*. If no *modularization* section is provided, no modularization is applied.

  Listing 5.1 shows an example order configuration file. If this order is executed, all goals of the mode *past-sessions* are executed. The executed load test is configured to be executed for 60 seconds and JMeter is used as load test driver. As source the pipeline uses traces, which are provided by the defined URL. The order enables the trace modularization approach and only the microservices *orders* and *carts* are going to be load-tested.

5.2 General Modifications

Regardless of the two different modularization approaches, which are implemented, certain changes on ContinuITy have to be made. Besides changes in *continuity.session.logs*

and *continuity.wessbas*, which implement the modularization approaches, *continuity.orchestrator* and *continuity.jmeter* need to be adapted.

5.2.1 Modification of continuity.orchestrator

The modularization is designed to be an optional functionality. Thus, the modularization flag and the modularization configuration has to be propagated to the responsible microservices. This is done by the *continuity.orchestrator* microservice. The *continuity.session.logs* and the *continuity.wessbas* service are applying the modularization, once the flag is propagated.

5.2.2 Modification of continuity.jmeter

Before *continuity.jmeter* starts executing the load test, the load test is annotated. Without applying the modularization, only the annotation model of the whole application needs to be applied. If the modularization is applied, possibly multiple services are going to be targeted. Thus, the previous implementation is extended in order to be capable of applying the annotation model for each microservice under test.

5.3 Trace Modularization

Because the *continuity.session.logs* service pulls the monitoring data from a monitoring data provider and the approach has to be independent of the monitoring data provider, the modularization has to be done in the *continuity.session.logs* service.

Figure 5.2 shows the modularization process. All colored elements are added in the context of the modularization approach. If an order is submitted, the *continuity.session.logs* service calls the trace data provider and receives all traces, which need to be modularized. Secondly, *continuity.session.logs* triggers the trace modularization, which is implemented as introduced in Section 3.3. After the traces are modularized, the original pipeline is executed and the session logs are extracted. The modularized session logs can then be retrieved by *continuity.wessbas* and used to generate a workload model.

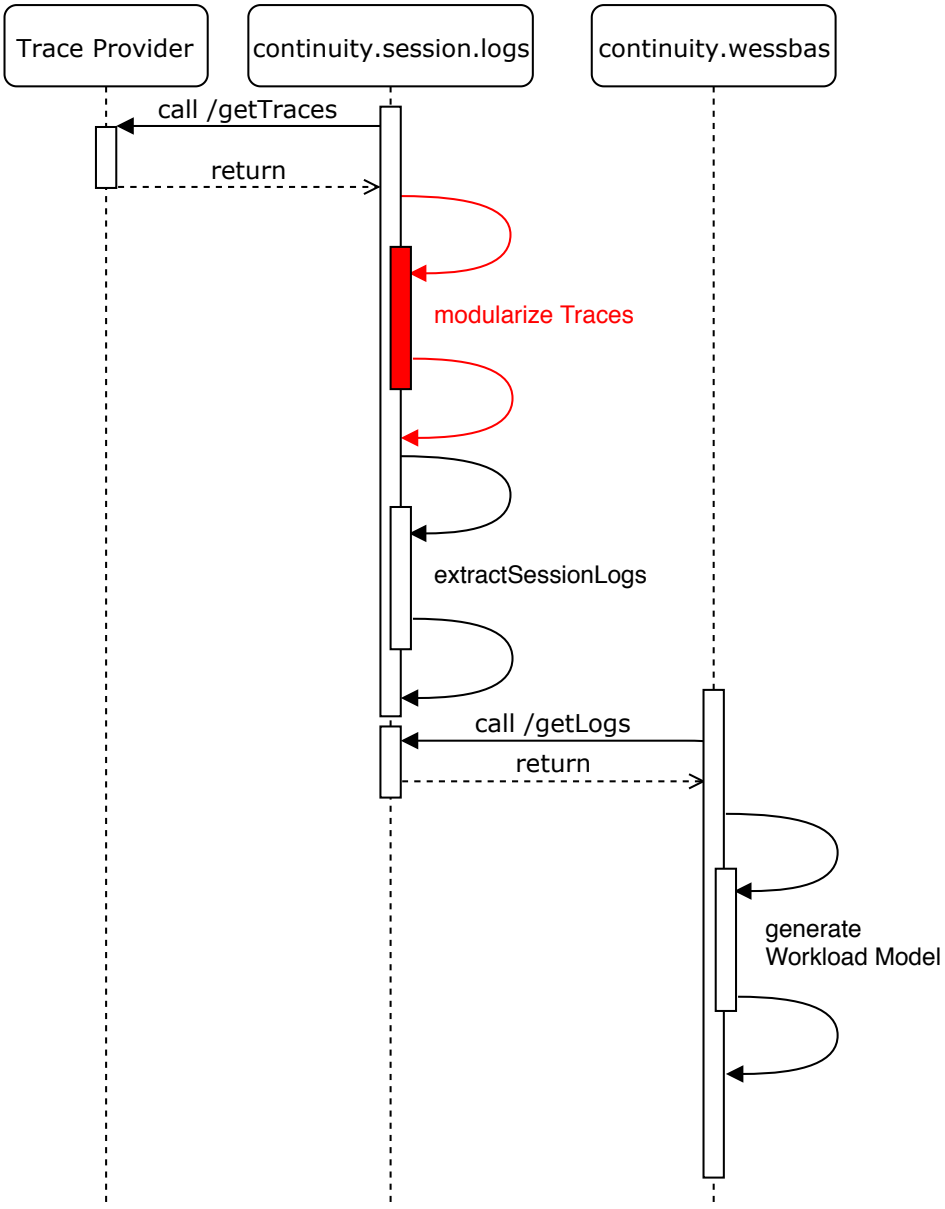


Figure 5.2: Sequence diagram of the trace modularization approach

5.4 Workload model modularization

Considering the workload model modularization approach, both *continuity.session.logs* and *continuity.wessbas* need to be modified.

Figure 5.3 shows the workload model modularization process. All colored elements are added in the context of the modularization approach. Analogously to the implementation of the trace modularization, *continuity.session.logs* retrieves traces from a trace data provider. Instead of modularizing the traces, *continuity.session.logs* directly extracts session logs from the traces and the non-modularized session logs are retrieved by the *continuity.wessbas* service. The session logs first are transformed into a mix of behavior models. Each behavior model represents different user groups. In order to modularize certain requests, *continuity.wessbas* loads the corresponding traces from the trace provider. For each behavior model, each Markov state is analyzed. First, the corresponding traces of the current Markov state are extracted. This can be done based on the request properties of the Markov state. Second, the traces are sent to *continuity.session.logs*. Therefore, a separate HTTP endpoint was introduced, which is capable of receiving serialized trace data. *continuity.session.logs* modularizes the traces and extracts session logs. In addition, the pre- and postprocessing time of the entry states and exit states are calculated (see Section 3.5.4) The modularized session logs for the specific Markov state are sent back to *continuity.wessbas*. The modularized session logs are then transformed to a modularized behavior model. If the session logs of the Markov state cannot be modularized, the current Markov state has to be skipped or if a modularized behavior model is existing, the behavior models have to be merged. After each Markov state is skipped or replaced, a WESSBAS DSL is generated.

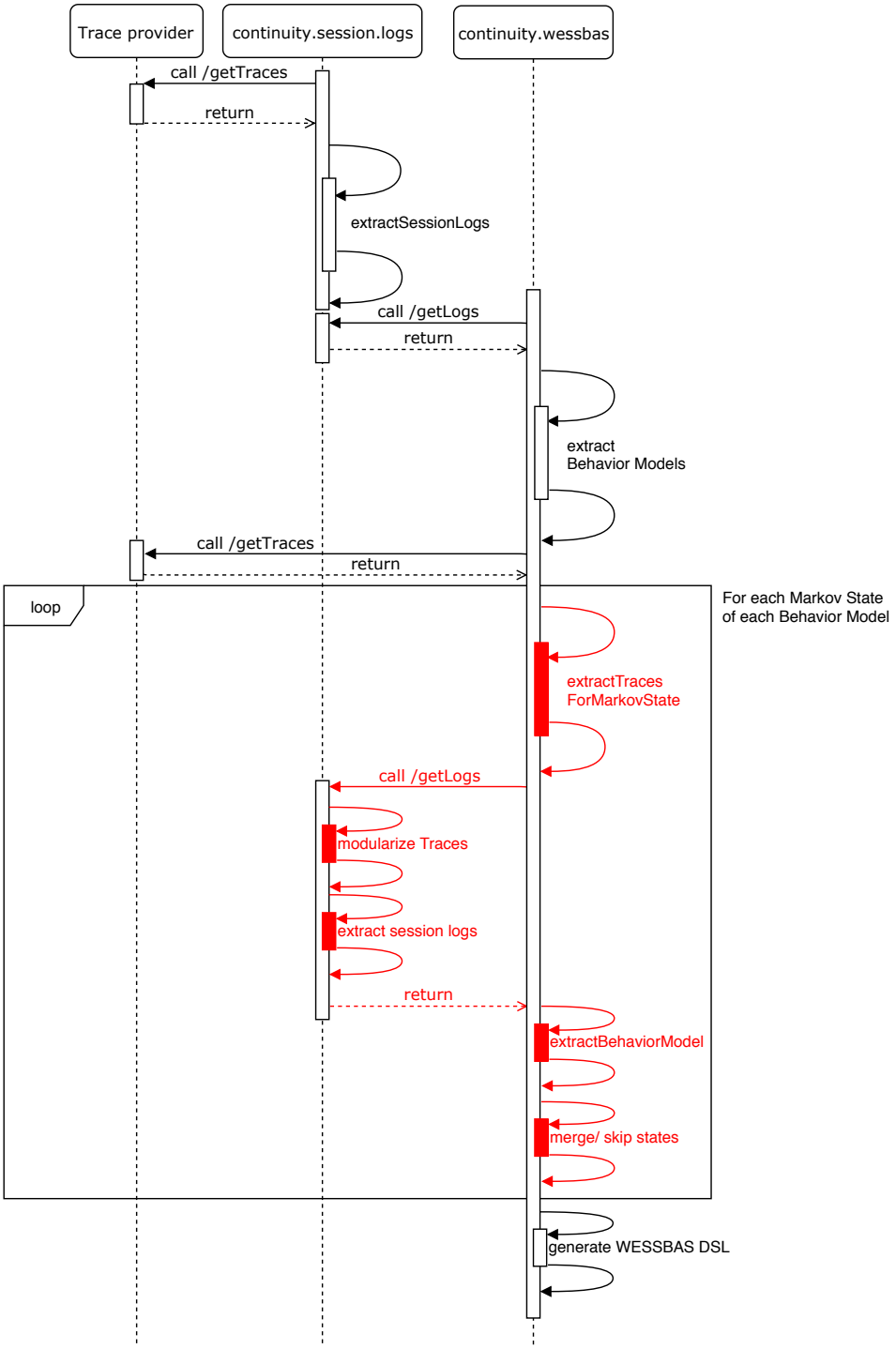


Figure 5.3: Sequence diagram of the workload model modularization approach

Chapter 6

Evaluation

In this chapter, the presented modularization approaches are evaluated with respect to the research questions. Section 6.1 presents the research questions, which are tried to be answered with the experiment results. Section 6.2 presents the design of the experiment and introduces the used experiment setup. In Section 6.3, the measured metrics of the conducted experiment are presented and described. The results are further discussed and interpreted in Section 6.4. In Section 6.5, the threats to validity are discussed.

6.1 Research Questions

As already presented in the objectives (see Section 1.2), most of the research questions are going to be answered by conducting an experiment. In the context of the evaluation of the approaches, the following research questions are addressed:

- **RQ2**: How representative are the modularization approaches compared to a representative system-level load test?
- **RQ3**: How is the representativeness influenced by the architecture and dependencies of an application?
- **RQ4**: How much does the modularization simplify the use of automated representative load testing in the context of microservice architecture development?
- **RQ5**: How does the modularization approach affect the resource consumption of a load test?
- **RQ6**: How does the approaches affect the needed duration of a load test?

Research question 1 is not in the scope of the evaluation, since the modularization approaches are presented in Chapter 3.

6.2 Methodology

In the following, the design of the experiment is introduced. Section 6.2.1 presents the procedure of the experiment. In Section 6.2.2, the used metrics are presented and described. The technical details of the experiment execution are described in Section 6.2.3

6.2.1 Experiment design

In order to evaluate the approach, we are using a microservice demo application, called Sock Shop [Wea18], since it is recommended to be an appropriate benchmark application [AMPJ17]. Sock Shop is a typical web store offering a catalogue of different socks, which can be ordered. The application consists of 7 different microservices written in NodeJS, Java and GO:

- *frontend*: This service provides the endpoints for the webclient and forwards the requests to the responsible microservices.
- *catalogue*: This service is responsible for offering the different items, which can be bought.
- *user*: This service manages the users and the corresponding addresses and credit cards
- *cart*: This service manages the shopping cart of the application.
- *orders*: This service is responsible for accepting orders.
- *shipping*: This service executes the shipment of the order.
- *payment*: This service authorizes the payment of the application.

In the context of ContinuITy, a experiment framework was developed, which enables to automize the execution of experiments with ContinuITy (see https://github.com/ContinuITy-Project/Experimentation-Utils). Concerning the research questions, the approaches need to be compared with a conventional representative load test. Since all approaches need existing monitoring data of a production system, we simulate the load with a predefined load test. This load test will be run once before the non-modularized load test is going to be started.

During the experiment, certain metrics are captured. In order to get metrics of all services, we are using Prometheus [Piv18a]. As monitoring tool, which is capable of providing traces, we are using inspectIT [Gmb18].

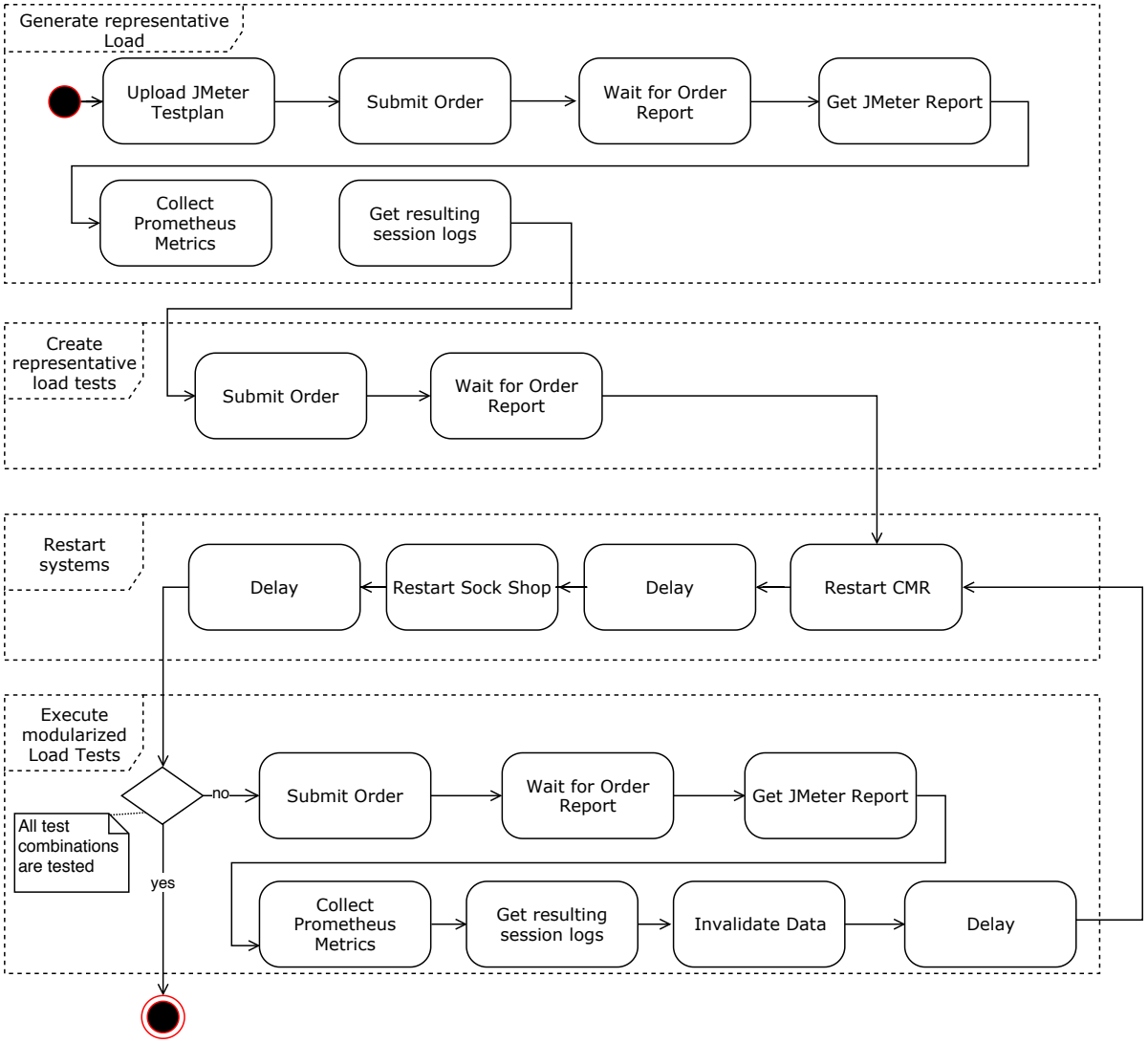


Figure 6.1: Experiment design

Figure 6.1 shows the design of the experiment. In the first phase the representative load is generated by a predefined load test. After uploading the load test to *continuity.jmeter*, an order executing the uploaded load test is executed. After the load test has finished, the JMeter report is stored, the Prometheus metrics are captured and the resulting session logs are stored.

In the second phase, 6 different load tests are generated. For the services *orders*, *shipping* two modularized load tests using the two different modularization approaches are created. In addition, a non-modularized load test is generated and a modularized test for *payment* by using the trace modularization approach. In order to minimize the risk

of any failures, the load test execution is separated from the load test generation. If a load test is once generated, it can be executed at any point of time.

In the third phase, the the monitoring tool (CMR) and the Sock Shop application is restarted, in order to guarantee independent test conditions for each test run.

In the fourth phase, load tests are going to be executed. For each run, the JMeter report and metrics provided by Prometheus are provided. Each Sock Shop microservice is tested separately with the trace modularization and the workload model modularization. The *frontend* is not tested separately, because this test scenario is equivalent to a non-modularized load test.

6.2.2 Metrics

The analysis of the runs is mainly based on the Prometheus metrics. In the context of this experiment, the following metrics are going to be focused:

- *Request Rate*: In order to compare the workload behavior and workload intensity of the different modularization approaches, the number of requests per time and per endpoints is analyzed.
- *Memory consumption*: Another characteristics for the workload is the resource consumption. Thus the memory usage has to be analyzed and compared. In addition, this metric is essential for answering *RQ5*.
- *CPU usage*: Analogously to the memory usage, the CPU usage is an import characteristic of used workload.

Based on the measured metrics, RQ2, RQ3, RQ5, and RQ6 can be answered.

6.2.3 Experimental Setup

In order to avoid any mutual interference between the measured metrics of the different service, it is essential to assign each microservice to one physical CPU and provide a separate virtual memory. The experiment is executed on two different servers.

Figure 6.2 shows the setup of the experiment. ContinuITy and Sock Shop are deployed on different physical machines, and the respective microservices of the Sock Shop application and the corresponding databases of the services are pinned to a certain CPU and a certain virtual memory. All services provide an additional endpoint, were Prometheus metrics can be fetched. In order to analyze the resource usage of the *continuity.jmeter* service, the service also offers an additional Prometheus endpoint and

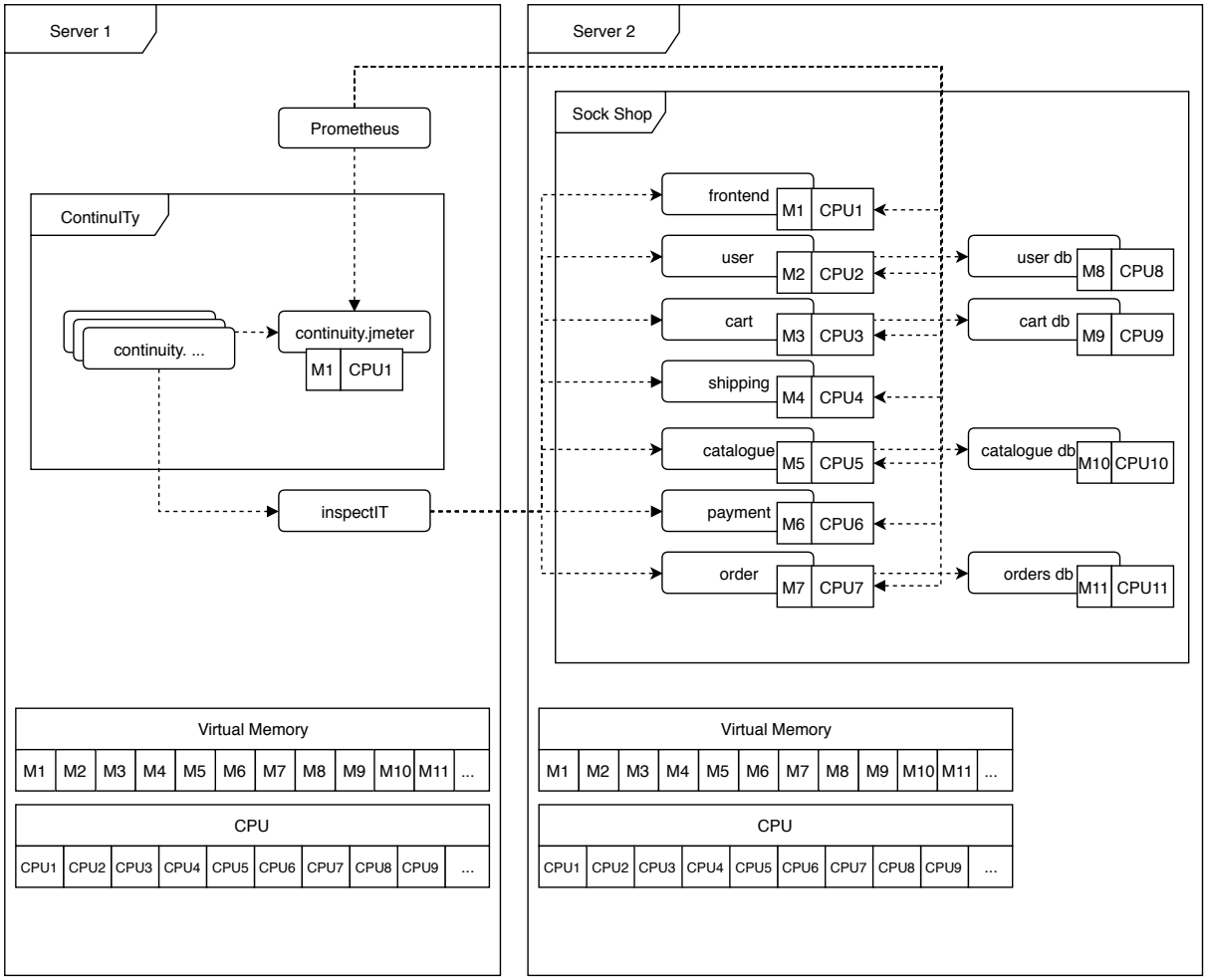


Figure 6.2: Experimental setup

is pinned to a separate memory and CPU range. In addition, a monitoring tool which is capable of providing traces is needed. We use a modified version of inspectIT [Gmb18], which is compatible of Zipkin traces [org18]. Whereas the Java services *cart*, *orders* and *shipping* are monitored with the inspectIT agent, the services *frontend*, *catalogue*, *user* and *payment* are monitored using a Zipkin tracer.

Both machines provide eight CPU cores. Server 1 provides 32 GB of memory, Server 2 provides 8 GB of memory. Since the deployment of ContinuITy and the monitoring service consumed much more resources, we decided to deploy it on a machine with more run. Each of the seven services under test got reserved 50% of a CPU core, each database 25 %. Hence the resources of the services cannot interact. Each load test was conducted with 100 users and took 15 minutes.

6.3 Experiment Results

In the following, the results of the conducted experiment (see Section 6.2.1) are provided. Thereby, the modularized load test runs of the services *shipping*, *payment*, and *orders* are compared with the non-modularized load test and the reference load test by using the presented metrics *request rate*, *memory consumption*, and *CPU usage*. All generated load tests were executed twice. Hence, in the following, we are going to distinguish between *run 1* and *run 2*.

6.3.1 Experiment Results of Modularized Load Test Execution of Microservice Shipping

As microservice *shipping* is only responsible for enabling a shipment, the microservice has only one endpoint. Figure 6.3 shows the amount of requests, which targeted the endpoint */shipping* of the microservice shipping in the first run. The solid line shows the number of requests of the reference load test. The dotted line shows the number of requests of the modularized load test using the trace modularization and the line with a mixed pattern represents the execution of the modularized load test using the workload model modularization. The non-modularized execution is not represented, since the non-modularized load test did not invoke the endpoint of service shipping in both runs. At a first glance, all depicted load tests are creating requests targeting the endpoint of shipping, because all lines are constantly rising. It is remarkable, that the request behavior of the trace modularization is very similar to the request behavior of the reference load test. However, the number of requests of the workload model modularization increases much faster than the original reference load.

Figure 6.4 shows the number of requests of endpoint *shipping* in the second run. The general trend of the load test executions does not differ to Figure 6.3. However the similarity of the reference load test and the trace modularization decreased.

Besides the request behavior of the load test, the resources of the SUT are good indicators, in order to determine the similarity of the load tests. Figure 6.5 shows the CPU time in seconds of service *shipping* while executing a load test for service *shipping*. On a first view, it is worth to mention, that for all test executions except for the trace modularization load test, the CPU time is rising almost linear. The workload model modularization approach almost fits to the line of the representative load test.

However, in Figure 6.6, the trace modularization approach is very similar to the reference load. Analogously as in the first run, the workload model modularization is also very similar to the load test. Although the execution of the non-modularized load test is

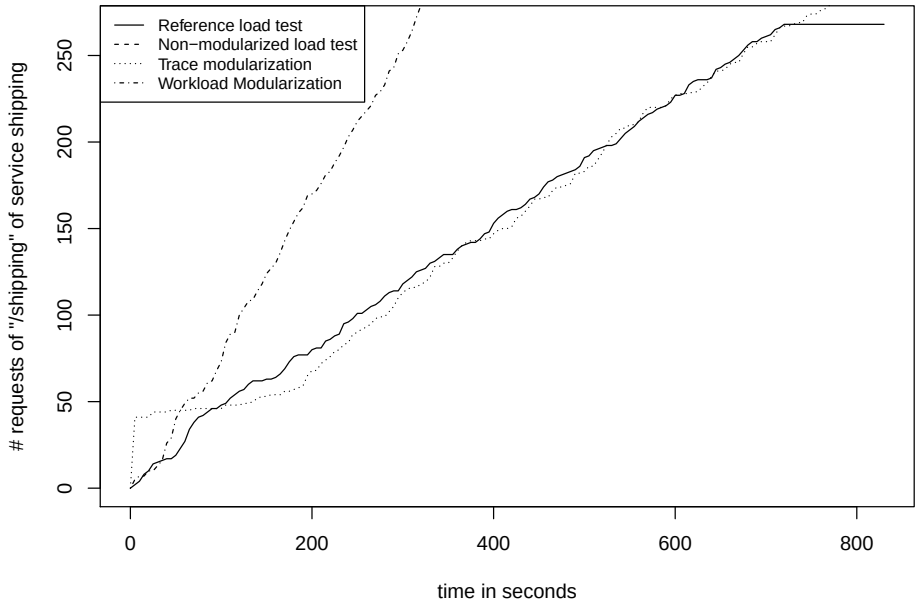


Figure 6.3: Modularized test of shipping: request count of shipping: /shipping, run 1

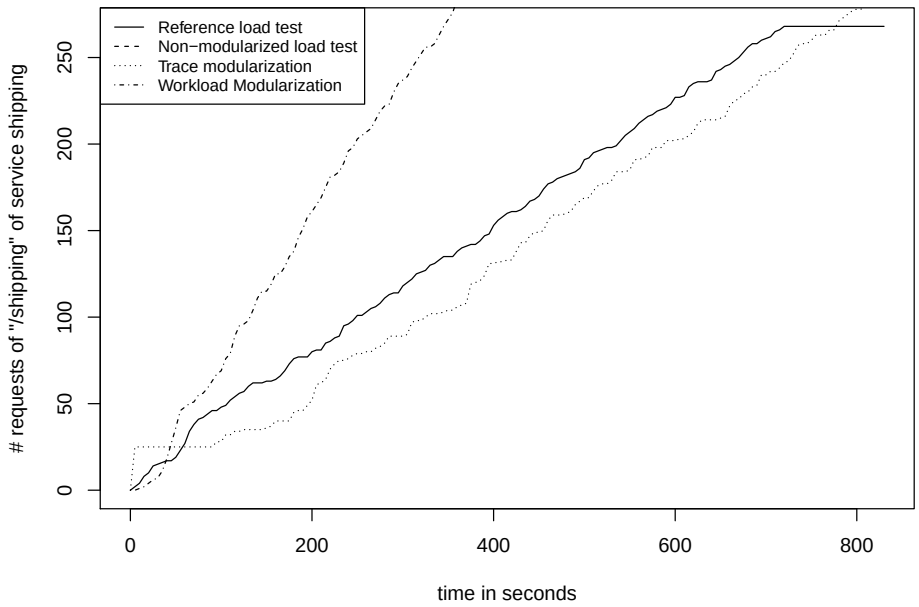


Figure 6.4: Modularized test of shipping: request count of shipping: /shipping, run 2

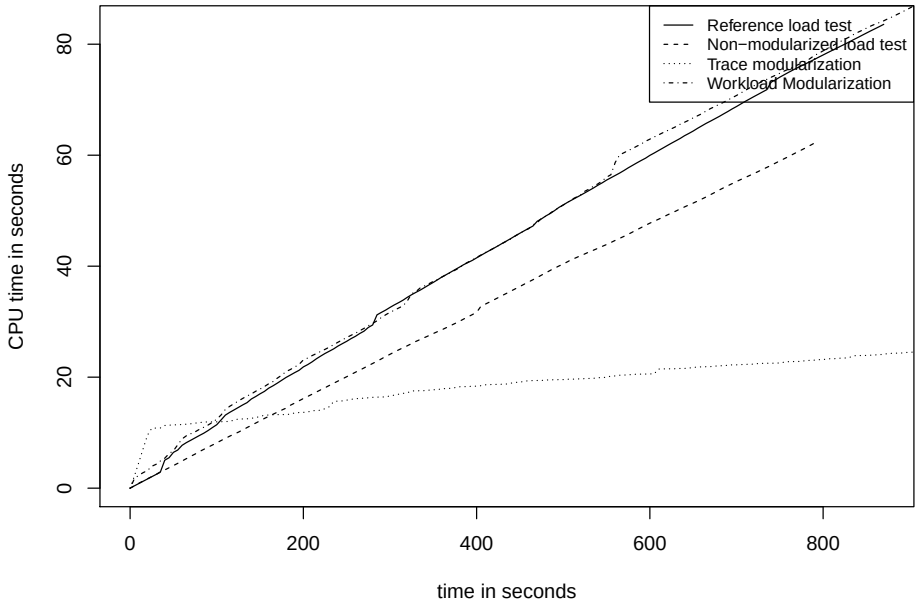


Figure 6.5: Modularized test of shipping: CPU seconds of service shipping, run 1

not as similar as the modularized load test executions, the non-modularized load test execution has a similar slope. Hence the CPU utilization is very similar. This can be also identified in Figure 6.7. The different box plots are visualizing the distribution of the CPU utilization of service shipping. The CPU utilization of all tests executions do not differ very much. Whereas the non-modularized execution has the lowest amount of outliers, the execution of the trace modularization has the highest amount of outliers. In addition, there is not much variation of the CPU utilization. Except of the outliers, for all test executions the CPU utilization of the shipping is constantly less than 15 percent. Although the non-modularized load test did not invoke the service *shipping*, the non-modularized CPU utilization is not significantly lower. Hence, this indicates, that CPU utilization of service *shipping* is not sensible for load.

Besides the CPU utilization, we measured the memory usage of the services. Figure 6.8 shows the memory usage in bytes of service shipping during the execution of the test scenarios. Concerning the reference load test execution, the memory usages vary not much. However, the execution of the non-modularized load test and the modularized load tests lead to a significantly higher memory usage of the service *shipping*. In addition the variation of the memory usage during the load test execution is much higher.

The described trend is even more significant in the second run. Figure 6.9 illustrates the memory usage distribution of run 2 during the different test scenarios. The median of

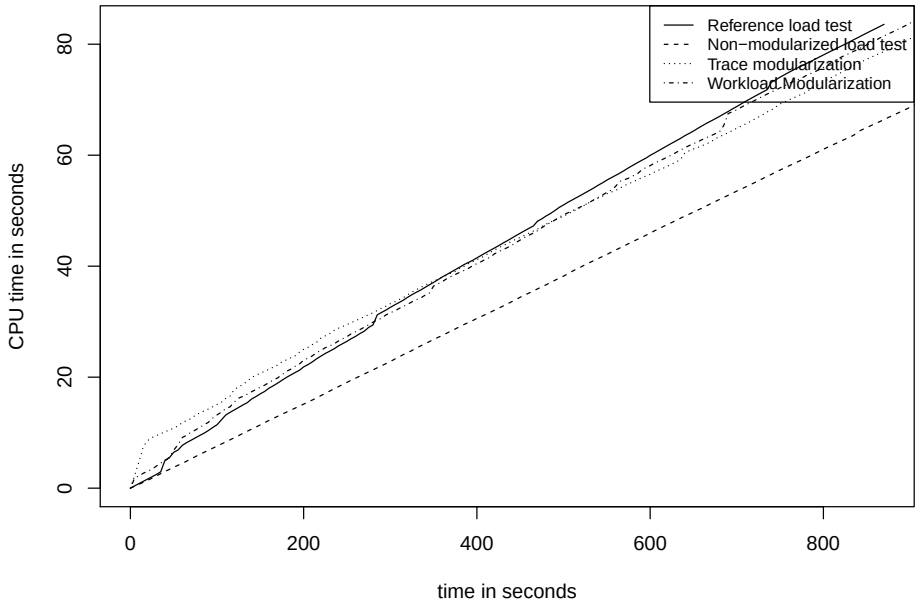


Figure 6.6: Modularized test of shipping: CPU seconds of service shipping, run 2

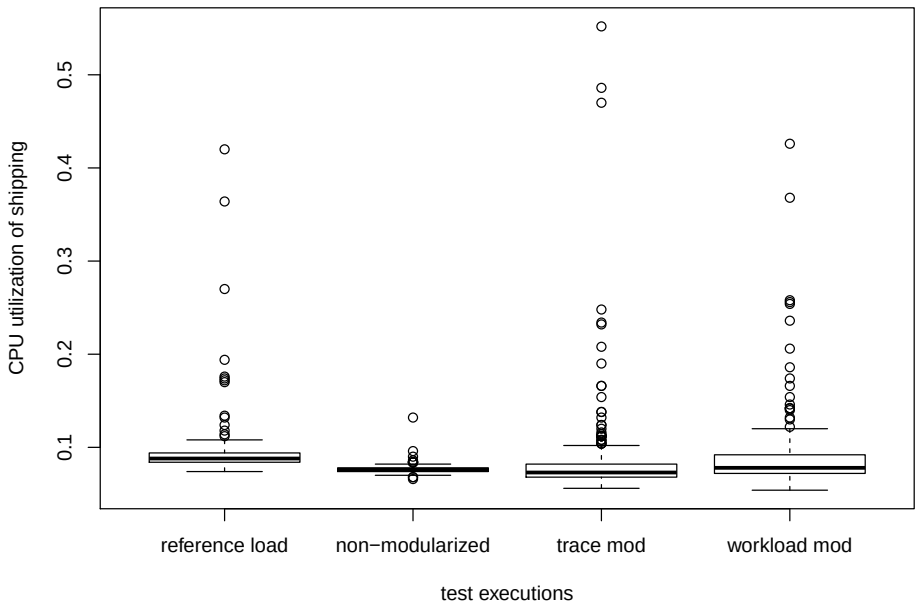


Figure 6.7: Modularized test of shipping: CPU utilization of service shipping, run 2

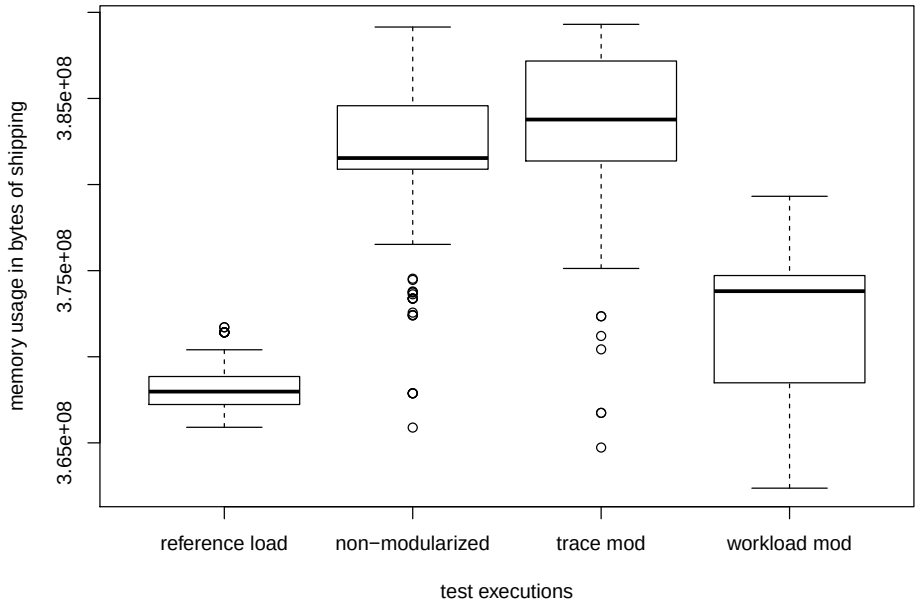


Figure 6.8: Modularized test of shipping: memory usage of service shipping, run 1

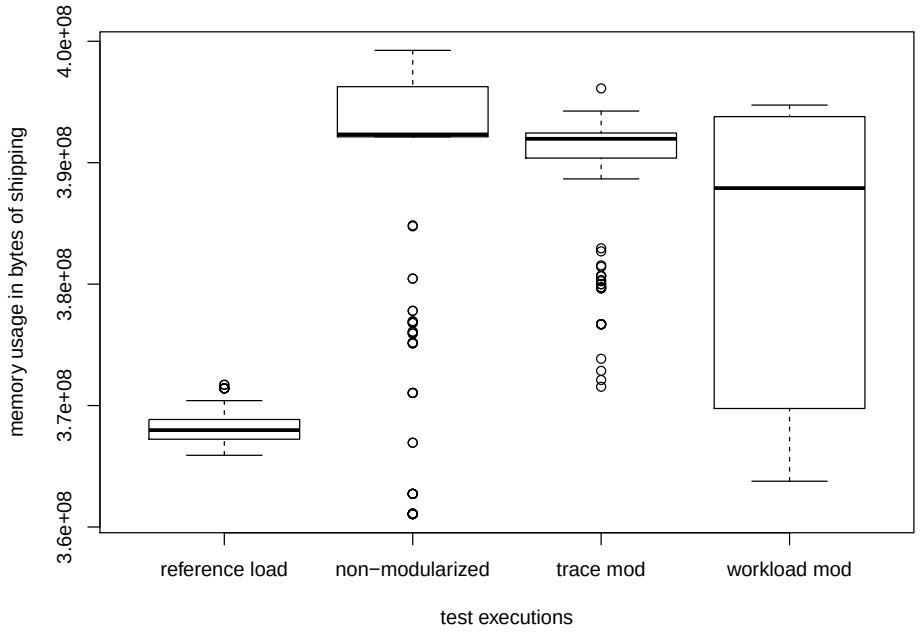


Figure 6.9: Modularized test of shipping: memory usage of service shipping, run 2

the memory usage of the generated load tests is much higher, than the reference load test. It is worth to mention, that the median and the variation of the memory usage during the workload model modularization load test execution increased compared to run 1. Other than in run 1, the quantiles of the trace modularization are significant smaller.

6.3.2 Experiment Results of Modularized Load Test Execution of Microservice Payment

Analogously to microservice *shipping*, microservice *payment* only provides one endpoint, which is responsible for authorizing the payment procedure. Figure 6.10 shows the request count of the endpoint */paymentauth* of run 1 during the execution of the modularized load test targeting microservice *payment*. Again, the non-modularized test execution is missing, because the non-modularized load test did not invoke the service *payment*. In addition, the request count of the workload model modularization is missing, because the results of the current test scenario were not available.

It is apparent, that right after the start of the trace modularization load test execution, the number of requests immediately rises up to approximately 80 requests, whereas the reference load test continuously creates requests. By comparing the slope of both lines, it is notable, that the slope of the trace modularization is lower than the reference load. Hence follows, that the request rate of the trace modularization is less than the reference load test.

Focusing the second run in Figure 6.11, the trend of of run 1 is reflected in the second run. However, the instant rise of the trace modularization only increases to a request count of approximately 40.

Figure 6.12 fits well into the context, since all test executions do not differ significantly. Figure 6.12 shows the variance of the CPU duration, represented as box plots. It is worth mentioning, that the quantiles of the trace modularization ideally match with the quantiles of the reference load test execution. However, although the workload model modularization and the non-modularization was not executed, the box plots do not differ significantly.

The trend is also detectable in the second run. As Figure 6.13 shows, the box plots differ very much. Although there are differences between the runs, the differences are not significant.

Besides the CPU utilization, the memory usage needs to be considered. Figure 6.14 and Figure 6.15 shows the memory usage of service *payment*. Both runs do not differ significantly: The difference between the medians is less than 0,5 MB. For both runs,

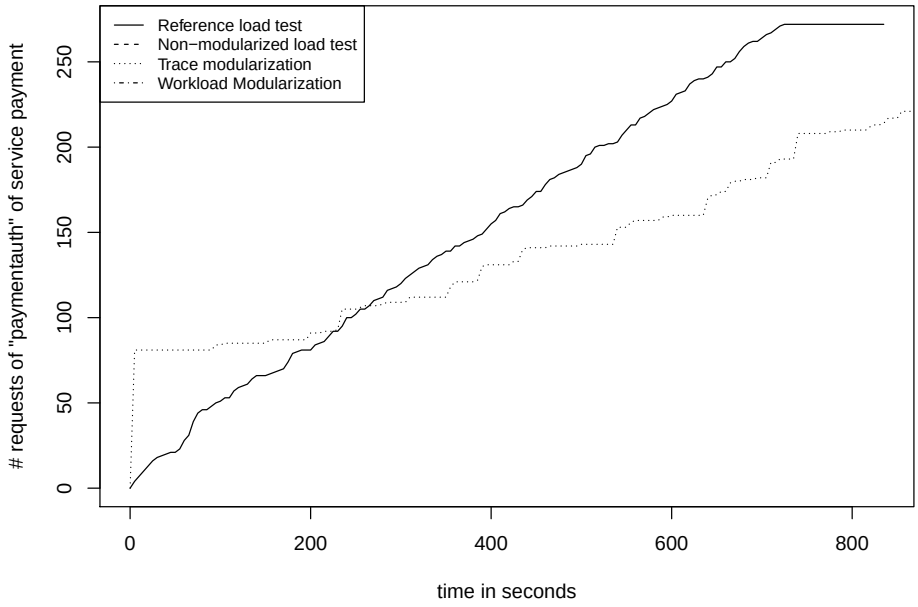


Figure 6.10: Load testing of payment: request count of payment: /paymentauth, run 1

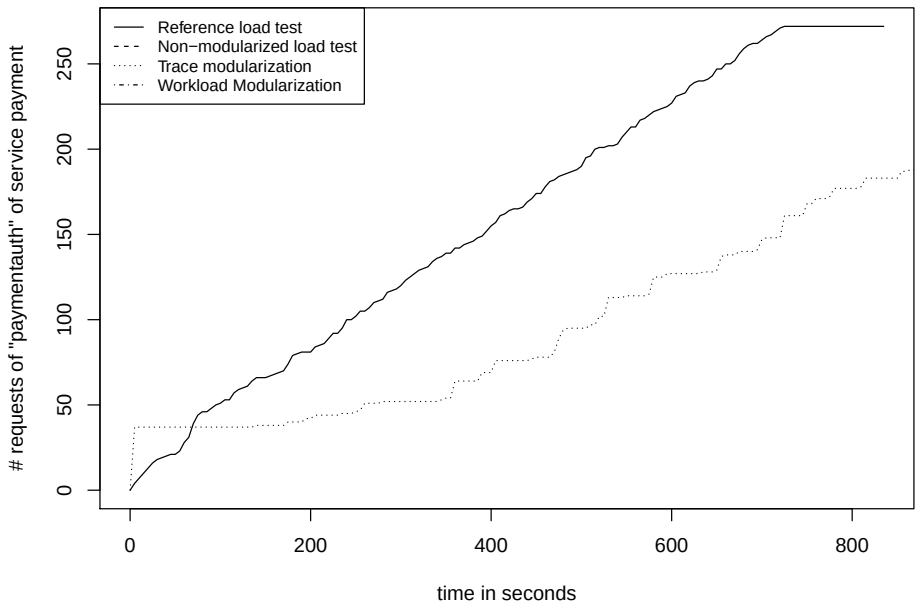


Figure 6.11: Load testing of payment: request count of payment: /paymentauth, run 2

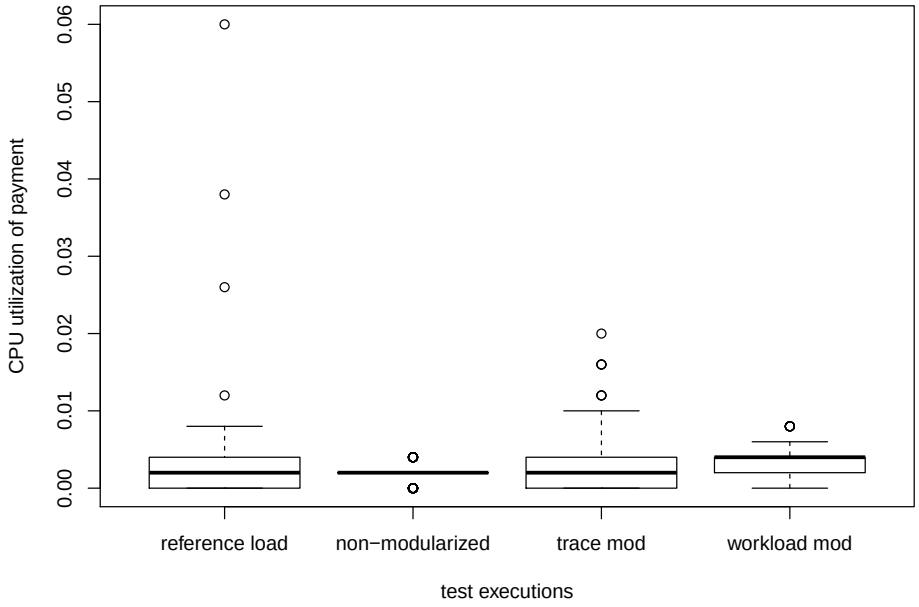


Figure 6.12: Load testing of payment: CPU utilization of payment, run 1

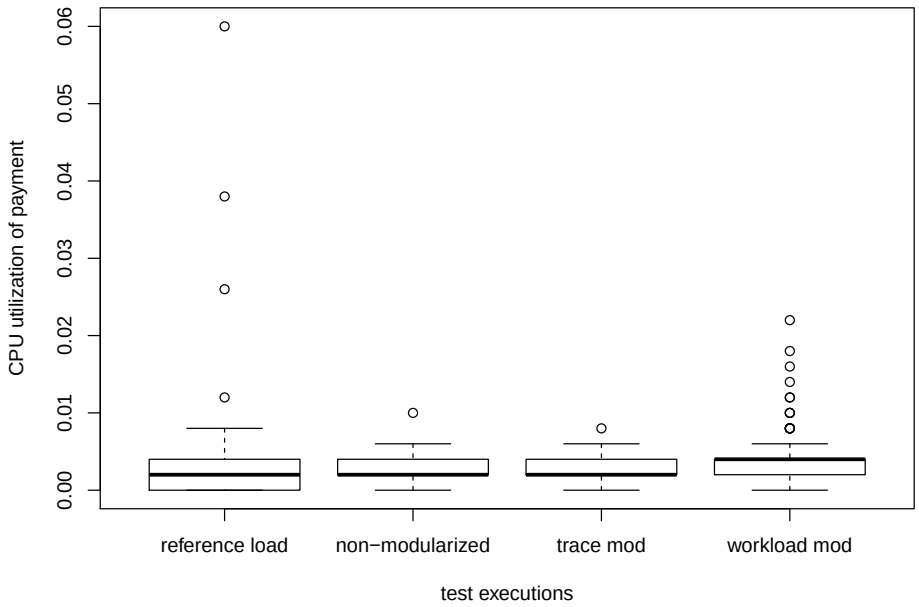


Figure 6.13: Load testing of payment: CPU utilization of payment, run 2

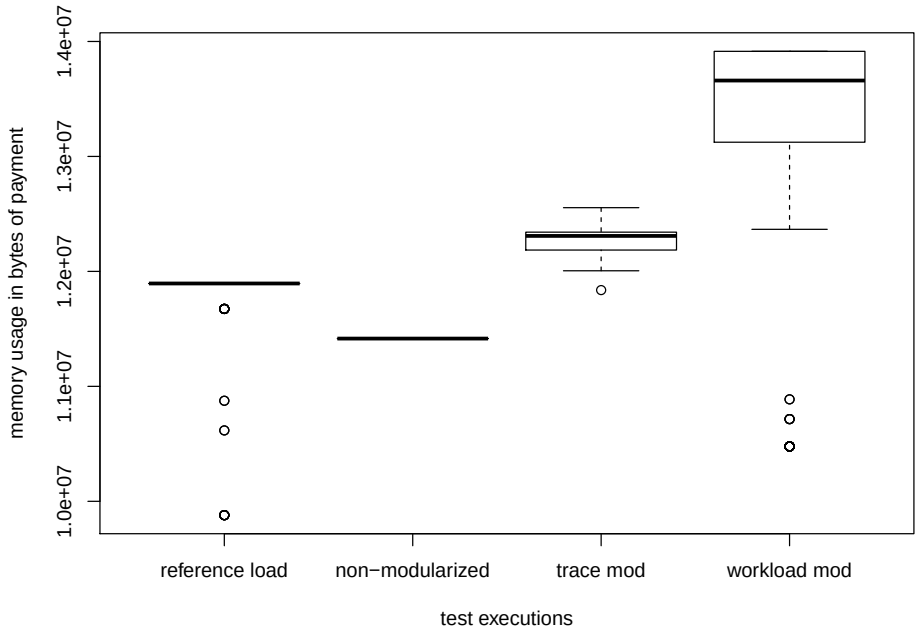


Figure 6.14: Load testing of payment: Memory usage of payment, run 1

the execution of the trace modularization approach leads to results, which do not differ significantly. In addition, it is worth to mention, that the memory usage during all runs almost stays on the same level.

6.3.3 Experiment Results of Modularized Load Test Execution of Microservice Orders

In the context of this experiment, we focus on two endpoints of microservice *orders*.

- */orders*: This endpoint is responsible for the creation of the order. It invokes calls to the services *carts*, *user*, *shipping* and *payment*.
- */{repository}/{id}/{property}*: This is a generic GET endpoint, which provides different information about already submitted orders.

Since all invoked services of microservice *orders* can have an influence on the performance of the service, the service are also load-tested.

Figure 6.16 shows the request count of endpoint */{repository}/{id}/{property}* of the first run. It is remarkable, that the reference load test execution does not differ significantly

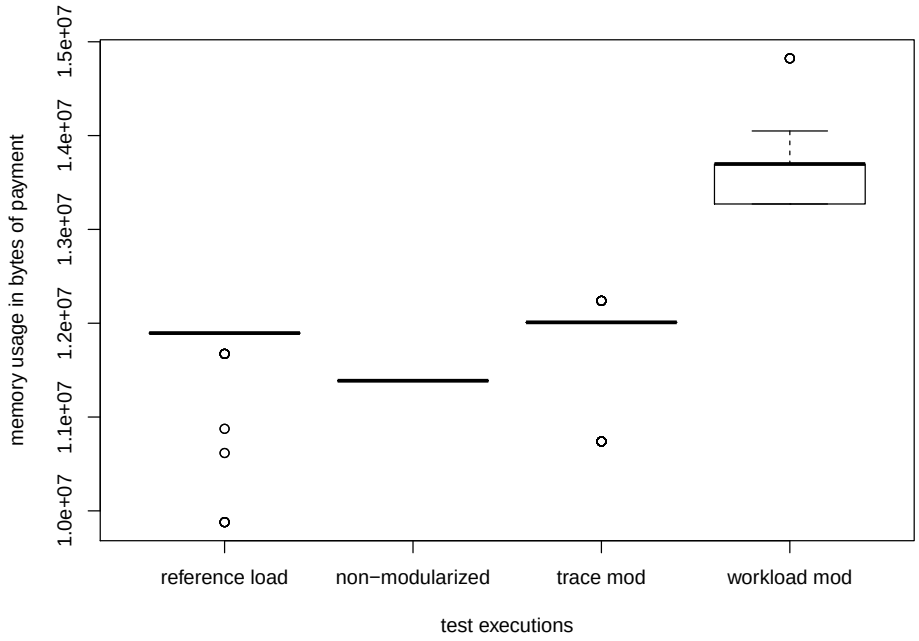


Figure 6.15: Load testing of payment: Memory usage of payment, run 2

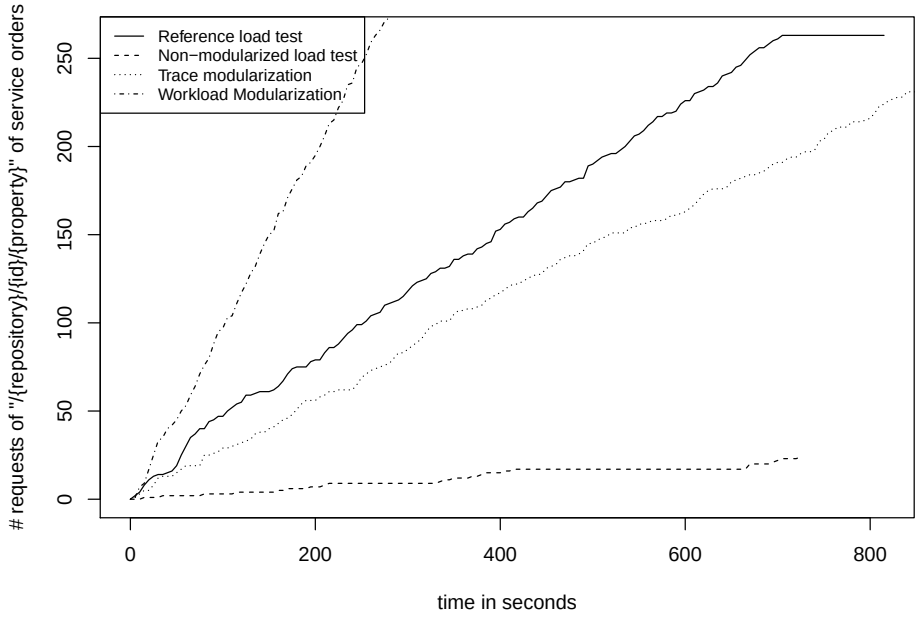


Figure 6.16: Load testing of orders: Request count of orders: */{repository}/{id}/{property}* , run 1

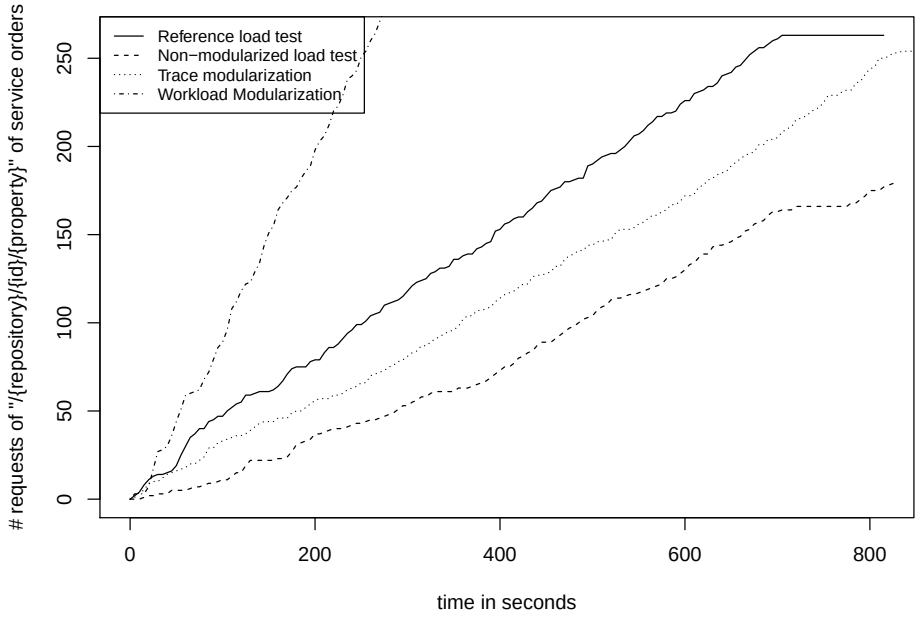


Figure 6.17: Load testing of orders: Request count of orders: */{repository}/{id}/{property}* , run 2

from the trace modularization; the slope of both executions also does not significantly differ. However, the non-modularized load test execution as well as the workload model modularization have completely different characteristics: Whereas the slope of the workload model modularization is relatively high compared to the reference load test, the slope of the non-modularized slope is relatively low.

Figure 6.17 shows the request count of the second run. The reference load test, the non-modularized load test as well as the trace modularization approach do not differ much in the slope. Hence, the request rate does not differ significantly. However, the slope of the workload model modularization approach does not fit to the slope of the reference load test.

Other than the results of endpoint */{repository}/{id}/{property}*, both modularization approaches have similar characteristics concerning the endpoint */orders*. The results of the first run are represented in Figure 6.18 In this test scenario, no data for the non-modularized test execution is available, since the test did not invoke the current endpoint. The similarity stays also for the second run, which is presented in Figure 6.19: Both modularization approaches lead to a similar slope.

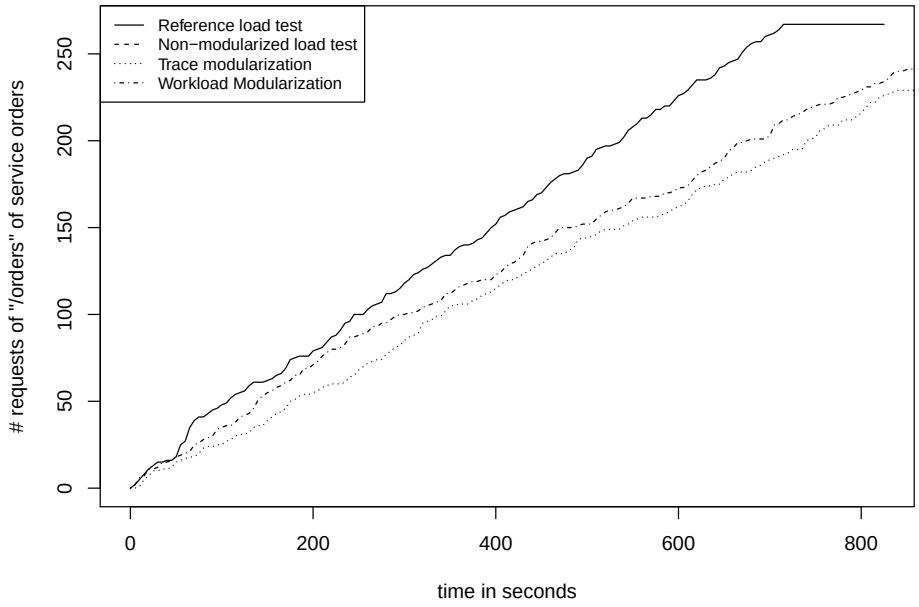


Figure 6.18: Load testing of orders: Request count of orders: */orders*, run 1

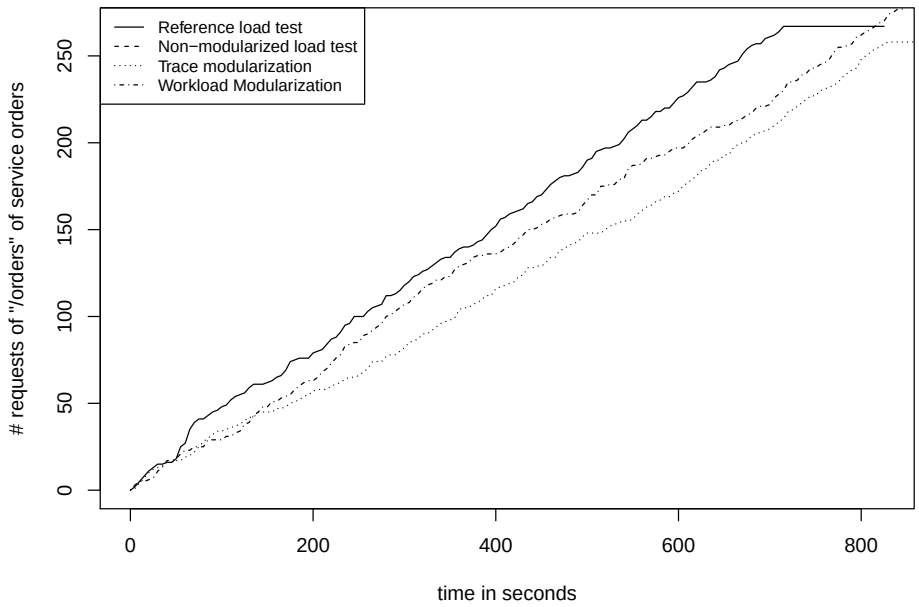


Figure 6.19: Load testing of orders: Request count of orders: */orders*, run 2

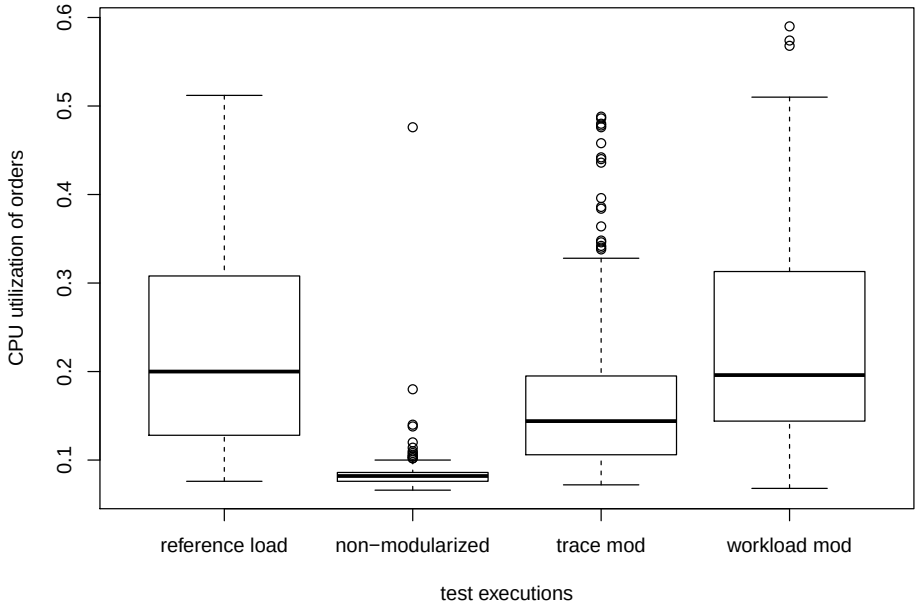


Figure 6.20: Load testing of orders: CPU utilization of orders, run 1

Besides the request count, we analyze the CPU utilization of microservice *orders*. Figure 6.20 and Figure 6.21 show the distribution of the CPU utilization during the execution of the different loads. Except of small differences, run 1 and run 2 have the same CPU utilization characteristics. Even the respective load test executions do not differ significantly for each run. Except the non-modularized execution, which did not cover all endpoints of *orders*, the modularization approaches fit to the characteristics of the reference load test.

Apart from analyzing the CPU utilization, we measured the memory usage of the microservice *orders*. Figure 6.22 and Figure 6.23 show the distribution of the memory usage during the different test runs. For both runs the median of the non-modularized execution is relatively low. Again, this might be caused by the fact, the non-modularized load test only executed one of the two endpoints in both runs. However, the distribution for the memory usage behaves completely different for the trace modularization: Whereas in run 1 the trace modularization constantly needs approximately 30 MB less of memory, in run 2, the box plots are overlapping and the median of the trace modularization is relatively higher.

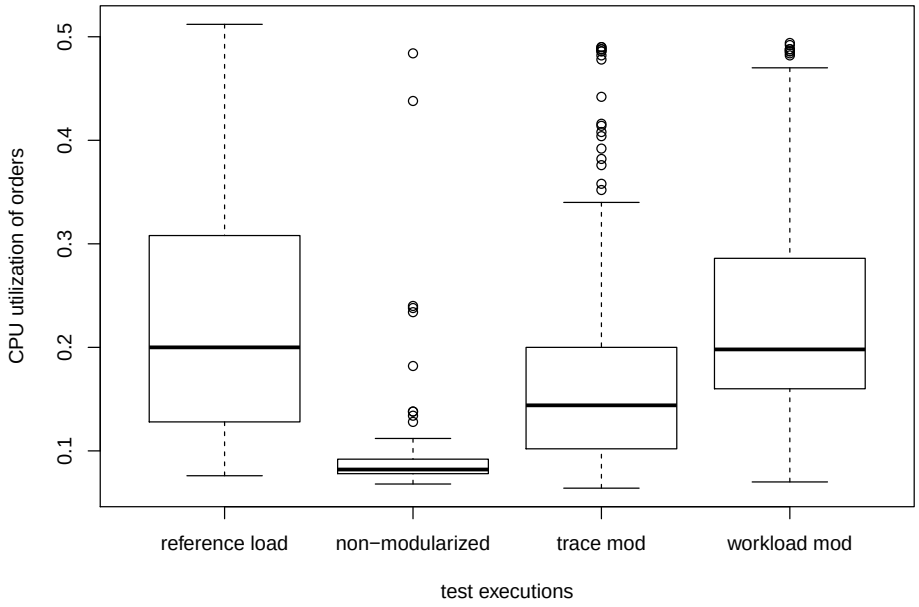


Figure 6.21: Load testing of orders: CPU utilization of orders, run 2

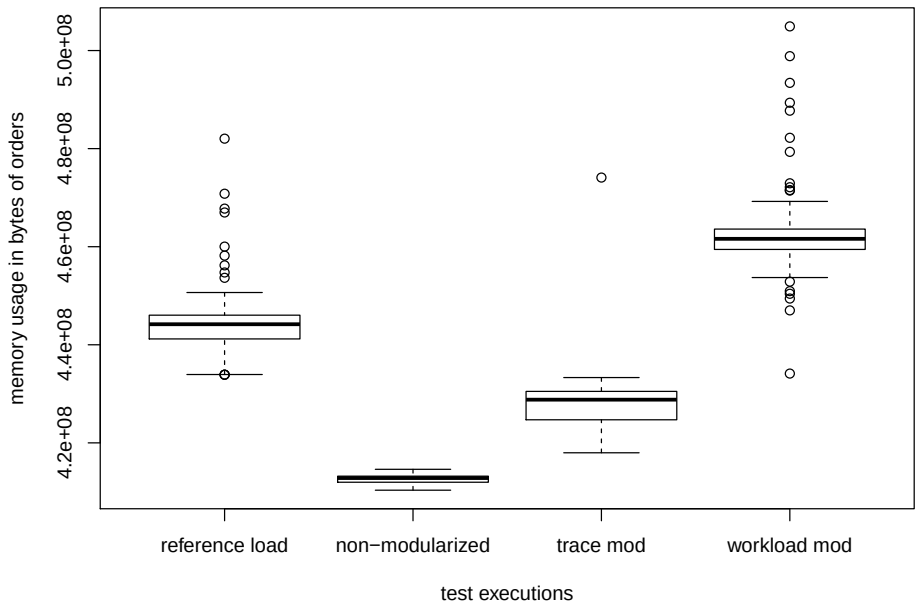


Figure 6.22: Load testing of orders: Memory usage of orders, run 1

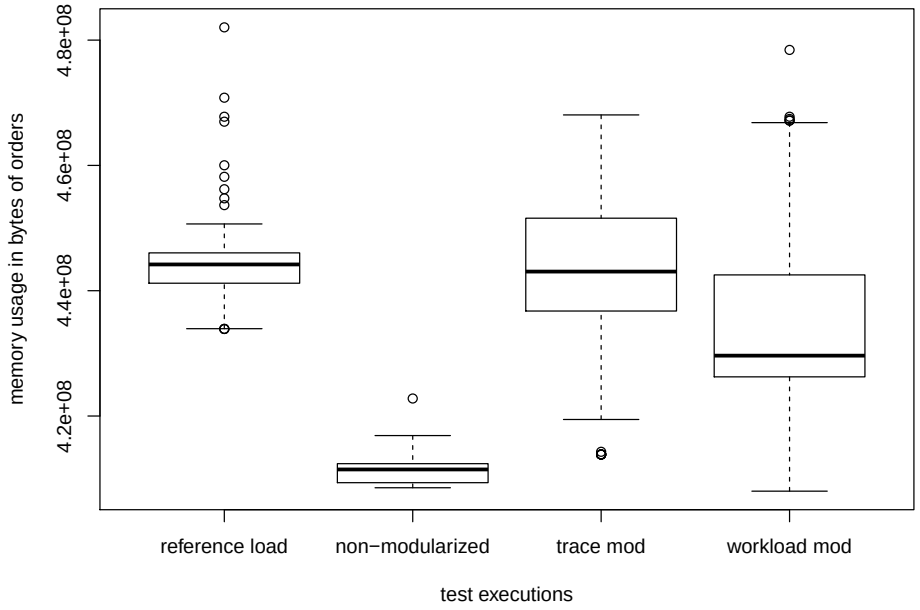


Figure 6.23: Load testing of orders: Memory usage of orders, run 2

6.4 Experiment discussion

In the following, the results of the experiments are discussed with respect to the research questions of Section 6.1.

6.4.1 RQ2: How Representative are the Modularization Approaches compared to a Representative System-level Load Test?

In order to answer this research questions, it is necessary compare the metrics request count, CPU utilization and memory usage during the executions of the modularized load tests. By analyzing the presented results, a trend is not obvious and clear. Since the results vary from the first run to the second run, the results are not as meaningful as expected. However it is worth to mention, for the modularized load test of *shipping* and *orders*, the trace modularization tends to provide similar results as the reference load test. Especially for the request count of orders in Figure 6.16 and Figure 6.17, it is remarkable, that the modularized approach leads to even more similar results than the non-modularized load test. However, the CPU utilization cannot be used properly, in order to compare the different approaches, because the executed load test do not have a

significant impact on the CPU utilization and the memory usage. This indicates, that the services, which were tested, are not CPU intensive. Furthermore, in the conducted experiment, the measured memory usage of the particular services cannot be used properly, in order to answer the research question R2. Because variation of memory usage of the different runs differ significantly, it is not possible to extract any trends. Although the representativeness of the modularized approaches cannot be quantified due to less meaningful result, it is worth it to further investigate on the representativeness, since the results of service *shipping* and *orders* were very promising. We expect more meaningful results, if the experiment is conducted on a more powerful hardware.

6.4.2 RQ3: How is the Representativeness influenced by the Architecture and Dependencies of an Application?

Based on the presented results, it not possible to draw any conclusions concerning the influence of the architecture. However, it can be stated, that the representative load test generation is only affected by the session logs of a system. Since the session logs of an application are independent of the architecture, the representative load test generation and also the modularized approaches should not be influenced by the architecture of an application. In order to prove conclusions, the experiment should have been conducted with several applications, which use different architectures.

6.4.3 RQ4: How much does the Modularization simplify the use of automated Representative Load Testing in the Context of Microservice Architecture Development?

However, without conducting a case study, the additional overhead of the microservice development team can be approximated. Besides providing an application model and a corresponding annotation model for the whole application, the development teams additionally have to provide an application model and annotation model for their specific microservice. Once the interfaces of the microservice are changing, the models have to be updated.

Depending on the dependency structure of the application, the microservice teams now only have to deploy the microservice under test and the depending microservices. Since this leads to a decrease of the resource consumption, multiple representative load test scenarios with different contexts can be parallelized. Hence, the duration of the overall load test execution can be decreased and the feedback to developers is much faster.

In order to prove the conclusions, a case study with a real development team and a real microservice application would be appropriate.

6.4.4 RQ5 and RQ6: How does the Modularization Approach affect the Resource Consumption and the Duration of a Load Test?

Depending on the dependency structure of the application, the modularization approach can safe resources, because not all services have to be deployed. Although the approaches itself consume additional resources while generating the load test, the execution of the load test is not affected.

Besides the resource consumption, the modularization approach also affects the duration of the load test. The resource saving of the modularized load test execution enables the parallelized execution of different modularized load tests, which would have to be executed sequentially with the modularization. Hence the duration of the aggregated duration of all load tests can be decreased. However, currently, this is only possible, if not all services have to be deployed in order to test the target service. By using the approach of Alghmadi et al. [ASSH16], the duration can be even decreased. The approach detects the point of time at which the test execution is repetitive and stops the load test.

6.5 Threats to Validity

In the following, we discuss the limitations and drawbacks of our experiment strategy.

6.5.1 Internal Threats

Based on the collected results, we cannot accept or decline our hypothesis, that the modularized load tests are as representative as using the non-modularized load test approach. Although a trend was detectable, the chosen metrics are not meaningful enough. In addition, the load test duration of 15 minutes might not be enough. We assume, that the available resources of the experiment machines might not be enough. In addition, the broken non-modularized load test increases the problem of having non-meaningful results.

6.5.2 External Threats

The experiment was only conducted with a few microservices of the microservice demo application Sock Shop. In order to have more representative experiment results, different applications should be tested and compared. In addition, the modularization approaches should be evaluated with applications, which are used by real users.

6.5.3 Construct Threats

The used metrics *CPU utilization*, *memory usage* and *request count* might not be appropriate enough, in order to show the similarity of two test executions. It is worth it to put additional metrics into consideration.

6.5.4 Conclusion Threats

Since many factors can influence the memory usage and the CPU utilization, it cannot be excluded, that the wrong trends were detected.

Chapter 7

Conclusion

In today's software development, applications are separated in several business components, so-called microservices. Each microservice is independent of each other and can be scaled separately. In the context of microservice application development, each microservice is developed by a separate team, which can decide on the used programming language. As today's software development cycle aims to be as fast as possible, the development teams are using continuous delivery pipelines, in order to be able to integrate and deploy new versions automatically and fast. However, before deploying a new version of a microservice in production, the microservice has to be tested by applying functional and non-functional tests. Besides the functional unit tests, representative load tests are necessary, in order to ensure, that the new version is capable to handle the real load. However, to the best of our knowledge, existing representative load testing approaches require to load test the whole application, since the load tests are generated based on the session information of the requests, which targeting the entry points of the application. Hence, each microservice development team has to deploy the whole application in order to test the microservice with a representative load. Depending on the number of microservices, this significantly increases the resource consumption and contradicts to the microservice development paradigm.

In this thesis, we address the topic by modularizing a representative load test for certain microservices under test. With the help of the proposed approach, microservice development teams can execute their own representative load test, without deploying all services. Depending on the dependency structure of the application, only a few additional dependent services have to deployed in addition.

We have elaborated three different approach modularization. The trace modularization is using monitoring traces of the target system and searches for nested requests, which directly targeting the SUT. The traces are then used to extract the probabilistic model of the user behavior and generate a representative load test. Analogously, the requests of the session logs can be replaced by nested requests, which directly target the SUT.

In our third approach, the already generated workload model, represented as Markov chain is manipulated. If a Markov state does not represent an endpoint, which directly targets the SUT, a Sub-Markov chain is generated, which consists of all nested endpoints invoked by the replacing Markov state. All generated Markov chains are merged into the non-modularized Markov chain. Since the trace modularization and the session logs request replacement approach lead to a semantically equivalent result, only the trace modularization and the workload model modularization are implemented. In order to evaluate the practicability, the resource saving, and the representativeness of the approach, we conducted an experiment. Both approaches were applied on three services of a demo application and compared to a reference load and a non-modularized test execution. All in all, the trace modularization provides the most promising results. However, the results are not significant enough, in order to evaluate the representativeness of the approaches in detail.

Future Work

Although this thesis analyzed and evaluated the introduced modularization approaches, we plan to do further investigations in the following topics:

- Test multiple microservice applications with different architectures: In the context of this thesis, we only tested the approaches with a subset of services of a sample application. In future, we plan to conduct the experiment with other applications which include more microservices. Hence, the results can be generalized.
- Conduct the experiment in a huge environment with many physical resources: The conducted experiment only was deployed on a relatively small environment. We plan to execute the same experiments in a much bigger environment, with a higher number of users.
- Evaluate the influence of performance stubs: By now, all dependent services of the SUT have to be deployed. By using performance stubs, the overhead can be even decreased: Instead of the dependent services, stubs are placed, which have a similar performance behavior as the real deployed services. We plan to evaluate, how the performance stubs influence the accuracy and the resource consumption of the approach.
- Conduct a case study with a microservice development team: Although we expect, that our approach simplifies representative load testing, the conclusion has to be further investigated by testing the approach with a real microservice development team. We expect, that the case study will provide meaningful insights.

Appendix

Bibliography

[AATP12] F. Abbors, T. Ahmad, D. Truscan, I. Porres. “MBPeT: a model-based performance testing tool.” In: *2012 Fourth International Conference on Advances in System Testing and Validation Lifecycle*. 2012 (cit. on p. 18).

[AMPJ17] C. M. Aderaldo, N. C. Mendonça, C. Pahl, P. Jamshidi. “Benchmark requirements for microservices architecture research.” In: *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*. IEEE Press. 2017, pp. 8–13 (cit. on p. 64).

[ASSH16] H. M. AlGhmadi, M. D. Syer, W. Shang, A. E. Hassan. “An automated approach for recommending when to stop performance tests.” In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2016, pp. 279–289 (cit. on pp. 5, 84).

[BBR+16] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, C. Rosenthal. “Chaos engineering.” In: *IEEE Software* 33.3 (2016), pp. 35–41 (cit. on p. 20).

[BC98] P. Barford, M. Crovella. “Generating representative web workloads for network and server performance evaluation.” In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 26. 1. ACM. 1998, pp. 151–160 (cit. on pp. 13, 17).

[BDIS04] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni. “Model-based performance prediction in software development: A survey.” In: *IEEE Transactions on Software Engineering* 30.5 (2004), pp. 295–310 (cit. on p. 21).

[BKR09] S. Becker, H. Koziolek, R. Reussner. “The Palladio component model for model-driven performance prediction.” In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22 (cit. on pp. 13, 21).

[com18] C. community. *ContinuITy GitHub project*. 2018. URL: https://github.com/ContinuITy-Project/ContinuITy (cit. on p. 55).

[CPG13] V. Chandel, S. Patial, S. Guleria. "Comparative Study of Testing Tools: Apache JMeter and Load Runner." In: *International Journal of Computing and Corporate Research* (2013) (cit. on p. 11).

[CSMS16] A. de Camargo, I. Salvadori, R. d. S. Mello, F. Siqueira. "An architecture to automate performance tests on microservices." In: *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services*. ACM. 2016, pp. 422–429 (cit. on p. 16).

[DGH+06] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, G. Weber. "Realistic load testing of web applications." In: *Proceedings of the 10th European Conference on. Software Maintenance and Reengineering, 2006. CSMR 2006.* IEEE. 2006, 11–pp (cit. on pp. 10, 17).

[FCW18] T. Field, R. Chatley, D. Wei. "Software Performance Testing in Virtual Time." In: *Companion of the 2018 ACM/SPEC ICPE*. ACM. 2018, pp. 173–174 (cit. on p. 21).

[Fei02] D. G. Feitelson. "Workload modeling for performance evaluation." In: *IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation*. Springer. 2002, pp. 114–141 (cit. on p. 12).

[FP18] V. Ferme, C. Pautasso. "A Declarative Approach for Performance Tests Execution in Continuous Software Development Environments." In: *Proceedings of the 2018 ACM/SPEC ICPE*. ACM. 2018, pp. 261–272 (cit. on p. 16).

[Gat18] GatlingCorp. *Open-source load and performance testing*. 2018. URL: https://gatling.io/ (cit. on p. 11).

[Gmb18] N. C. GmbH. *inspectIT*. 2018. URL: https://inspectit.rocks (cit. on pp. 64, 67).

[HF10] J. Humble, D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010 (cit. on pp. 1, 8, 19).

[HHMO17] C. Heger, A. van Hoorn, M. Mann, D. Okanović. "Application performance management: State of the art and challenges for the future." In: *Proceedings of the 8th ACM/SPEC on ICPE*. ACM. 2017, pp. 429–432 (cit. on p. 14).

[HLM+15] V. Hork, P. Libič, L. Marek, A. Steinhauser, P. Tma. "Utilizing performance unit tests to increase performance awareness." In: *Proceedings of the 6th ACM/SPEC ICPE*. ACM. 2015, pp. 289–300 (cit. on p. 20).

[HRJ+16] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, V. Sekar. “Gremlin: systematic resilience testing of microservices.” In: *36th International Conference on Distributed Computing Systems (ICDCS), 2016 IEEE*. IEEE. 2016, pp. 57–66 (cit. on p. 20).

[JH15] Z. M. Jiang, A. E. Hassan. “A survey on load testing of large-scale software systems.” In: *IEEE Transactions on Software Engineering* 41.11 (2015), pp. 1091–1118 (cit. on pp. 1, 10, 11).

[KRM06] D. Krishnamurthy, J. A. Rolia, S. Majumdar. “A synthetic workload generation technique for stress testing session-based systems.” In: *IEEE Transactions on Software Engineering* 32.11 (2006), pp. 868–882 (cit. on pp. 13, 18).

[LF14] J. Lewis, M. Fowler. “Microservices: a definition of this new architectural term.” In: *MartinFowler.com* 25 (2014) (cit. on pp. 14, 26).

[LT03] Z. Li, J. Tian. “Testing the suitability of Markov chains as Web usage models.” In: *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*. Nov. 2003, pp. 356–361 (cit. on p. 12).

[Mar11] R. C. Martin. *The clean coder: a code of conduct for professional programmers*. Pearson Education, 2011 (cit. on p. 7).

[Mye06] G. J. Myers. *The art of software testing*. John Wiley & Sons, 2006 (cit. on p. 9).

[org18] O. volunteer organization. *Zipkin*. 2018. URL: https://zipkin.io/ (cit. on p. 67).

[Piv18a] Pivotal. *Prometheus*. 2018. URL: https://prometheus.io/ (cit. on pp. 52, 64).

[Piv18b] Pivotal. *RabbitMQ*. 2018. URL: https://www.rabbitmq.com/ (cit. on p. 54).

[Piv18c] Pivotal. *Spring Framework*. 2018. URL: https://spring.io/ (cit. on p. 54).

[RSSP04] G. Ruffo, R. Schifanella, M. Sereno, R. Politi. “WALTy: A user behavior tailored tool for evaluating web application performance.” In: *Third IEEE International Symposium on Network Computing and Applications, 2004.(NCA 2004). Proceedings.* IEEE. 2004, pp. 77–86 (cit. on p. 17).

[SAH18] H. Schulz, T. Angerstein, A. van Hoorn. “Towards Automating Representative Load Testing in Continuous Software Engineering.” In: *Companion of the 2018 ACM/SPEC ICPE*. ACM. 2018, pp. 123–126 (cit. on p. 13).

[SKF06] M. Shams, D. Krishnamurthy, B. Far. “A model-based approach for testing the performance of web applications.” In: *Proceedings of the 3rd international workshop on Software quality assurance*. ACM. 2006, pp. 54–61 (cit. on p. 18).

[Sma18a] SmartBearSoftware. *Swagger*. 2018. URL: https://swagger.io (cit. on p. 54).

[Sma18b] SmartBearSoftware. *Swagger OpenAPI Specification Data Types*. 2018. URL: https://swagger.io/docs/specification/data-models/data-types/ (cit. on p. 51).

[SRT15] D. I. Savchenko, G. I. Radchenko, O. Taipale. “Microservices validation: Mjolnirr platform case study.” In: *38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015*. IEEE. 2015, pp. 235–240 (cit. on pp. 10, 14).

[SWH06] B. Schroeder, A. Wierman, M. Harchol-Balter. “Open Versus Closed: A Cautionary Tale.” In: *Nsdi*. Vol. 6. 2006, pp. 18–18 (cit. on p. 11).

[VHS+18] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, H. Krcmar. “WESSBAS: Extraction of probabilistic workload specifications for load testing and performance prediction—A model-driven approach for session-based application systems.” In: *Software & Systems Modeling* (2018), pp. 1–35 (cit. on pp. 1, 13, 17, 18, 21).

[VMC+16] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, Y. J. Song. “Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services.” In: *OSDI*. 2016, pp. 635–651 (cit. on p. 19).

[Wea18] Weaveworks. *SockShop*. 2018. URL: https://microservices-demo.github.io/ (cit. on p. 64).

[ZZL14] J. Zhou, B. Zhou, S. Li. “LTF: A model-based load testing framework for web applications.” In: *Conference on Quality Software (QSIC), 2014 14th International*. IEEE. 2014, pp. 154–163 (cit. on p. 19).

All links were last followed on December 21, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature