Gathering Solutions and Providing APIs for their Orchestration to Implement Continuous Software Delivery

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

Vorgelegt von
Johannes Wettinger
aus Stuttgart

Hauptberichter: Prof. Dr. Dr. h. c. Frank Leymann
Mitberichter: Prof. Dr. Kostas Magoutis
Mündliche Prüfung: 2. März 2017

Institut für Architektur von Anwendungssystemen der Universität Stuttgart

2017
CONTENTS

1 Introduction .................................................. 13
   1.1 Research Challenges and Contributions .................. 16
   1.2 Scientific Publications .................................... 21
   1.3 Thesis Structure .......................................... 25

2 Fundamentals and Related Work .............................. 27
   2.1 Continuous Delivery ....................................... 27
   2.2 Agile Software Development and DevOps ................ 31
   2.3 Application Topologies, Deployment and Orchestration .... 32
   2.4 Application Programming Interfaces (APIs) ............. 37
   2.5 Classification of Solutions and Implementations ....... 41
   2.6 Content Discovery and Crawling .......................... 44
   2.7 Chapter Summary and Discussion ........................ 45

3 Continuous Delivery based on Gathered Solutions .......... 49
   3.1 Gather’n’Deliver Method ................................... 50
   3.2 Collaboration among Roles and Organizations ............ 56
   3.3 Implementing Gather’n’Deliver ............................ 58
   3.4 Chapter Summary and Discussion ........................ 60
4 Application Environment Metamodel

4.1 Core Metamodel ........................................ 63
4.2 Instance of Metamodel .................................. 67
4.3 Solutions as Building Blocks of Application Environments .... 71
4.4 Chapter Summary and Discussion ....................... 81

5 Building Application Environment Topologies ........... 83

5.1 Classification Dimensions of Requirements and Solutions .... 83
5.2 Label Taxonomies ....................................... 89
5.3 Resolution of Application Environment Requirements .......... 94
5.4 Generating Application Environment Topology Skeletons .......... 106
5.5 Chapter Summary and Discussion ........................ 108

6 Collaborative and Automated Gathering of Solutions ...... 111

6.1 Collaborative Solution Repositories .......................... 111
6.2 Automated Discovery and Capturing of Solutions ............... 117
6.3 Continuous Delivery of Knowledge Base Variants ............... 119
6.4 GATHERBASE Framework ............................... 122
6.5 Chapter Summary and Discussion .......................... 127

7 Streamlined Orchestration of Diverse Solutions ............ 129

7.1 TOSCAfy Framework – Standards-driven Transformation .... 130
7.2 PLUGINVOLVE Framework – Pluggable Invocation .......... 141
7.3 ANY2API Framework – Automated APIfication .............. 147
7.4 Chapter Summary and Discussion .......................... 156

8 Integrated Architecture and Validation .................... 161

8.1 Architecture Overview .................................... 162
8.2 Prototype Implementations .................................. 164
8.3 Validation in the Scope of Continuous Delivery ................. 174
8.4 Validation beyond Continuous Delivery ...................... 188
8.5 Chapter Summary and Discussion .......................... 196
ZUSAMMENFASSUNG


Um von den Vorteilen zu profitieren, die Continuous Delivery mit sich bringt, ist ein hohes Maß an Automatisierung erforderlich. Dies wird aus tech-
Abstract

In traditional IT environments, it is common for software updates and new releases to take up to several weeks or even months to be eventually available to end users. Therefore, many IT vendors and providers of software products and services face the challenge of delivering updates considerably more frequently. This is because users, customers, and other stakeholders expect accelerated feedback loops and significantly faster responses to changing demands and issues that arise. Thus, taking this challenge seriously is of utmost economic importance for IT organizations if they wish to remain competitive. Continuous software delivery is an emerging paradigm adopted by an increasing number of organizations in order to address this challenge. It aims to drastically shorten release cycles while ensuring the delivery of high-quality software. Adopting continuous delivery essentially means to make it economical to constantly deliver changes in small batches. Infrequent high-risk releases with lots of accumulated changes are thereby replaced by a continuous stream of small and low-risk updates.

To gain from the benefits of continuous delivery, a high degree of automation is required. This is technically achieved by implementing continuous delivery pipelines consisting of different application-specific stages (build, test, production, etc.) to automate most parts of the application delivery process. Each stage relies on a corresponding application environment such
as a build environment or production environment. This work presents concepts and approaches to implement continuous delivery pipelines based on systematically gathered solutions to be used and orchestrated as building blocks of application environments. Initially, the presented Gather’n’Deliver method is centered around a shared knowledge base to provide the foundation for gathering, utilizing, and orchestrating diverse solutions such as deployment scripts, configuration definitions, and Cloud services. Several classification dimensions and taxonomies are discussed in order to facilitate a systematic categorization of solutions, in addition to expressing application environment requirements that are satisfied by those solutions. The presented GatherBase framework enables the collaborative and automated gathering of solutions through solution repositories. These repositories are the foundation for building diverse knowledge base variants that provide fine-grained query mechanisms to find and retrieve solutions, for example, to be used as building blocks of specific application environments. Combining and integrating diverse solutions at runtime is achieved by orchestrating their APIs. Since some solutions such as lower-level executable artifacts (deployment scripts, configuration definitions, etc.) do not immediately provide their functionality through APIs, additional APIs need to be supplied. This issue is addressed by different approaches, such as the presented Any2API framework that is intended to generate individual APIs for such artifacts. An integrated architecture in conjunction with corresponding prototype implementations aims to demonstrate the technical feasibility of the presented approaches. Finally, various validation scenarios evaluate the approaches within the scope of continuous delivery and application environments and even beyond.
An erster Stelle möchte ich mich ganz herzlich bei meinem Doktorvater Prof. Dr. Dr. h. c. Frank Leymann bedanken, nicht nur für die großartige Unterstützung und Begleitung meiner Forschungsarbeit am Institut für Architektur von Anwendungssystemen (IAAS), sondern auch für das Vertrauen, die vielfältigen Freiheiten und die übertragene Verantwortung. Ich habe in dieser Zeit unglaublich viel gelernt, und damit meine ich neben den fachlichen Themen vor allem die ganz persönliche Entwicklung und Entfaltung. Diese Dinge sind mit Worten nicht zu fassen. Vielen Dank! Auch meinem Zweitgutachter Prof. Dr. Kostas Magoutis für die Übernahme des Mitberichts ein herzliches Dankeschön. Ausdrücklich danken möchte ich zudem meinen Kollegen und Freunden am IAAS für die tolle Zusammenarbeit, die konstruktiven Diskussionen sowie die tatkräftige Unterstützung in Forschung und Lehre. Besonderen Dank an Uwe Breitenbücher für das Korrekturlesen und wertvolle Feedback zu dieser Arbeit. Ohne den Rückhalt und die Unterstützung durch meine Familie wäre die vorliegende Arbeit in dieser Form nicht entstanden. Daher ein herzliches Dankeschön an meine Eltern, Geschwister und insbesondere an meine Frau Sibylle und meine Tochter Anastasia, die mir sehr viel Freude und Kraft geschenkt haben.
Many of today’s users, customers, and other stakeholders of software in the fields of Cloud services, Web applications, mobile apps, and the Internet of things expect rapid responses to changing demands and emergent issues. Thus, shortening the time to market of software features, fixes, and new releases becomes a critical competitive advantage for many companies. In addition, users and customers must be involved in tight feedback loops to ensure building the ‘right’ software, and providing features that actually solve their issues. This eventually improves customer satisfaction and reduces costs by avoiding wasted development effort spent creating and maintaining unused or even disliked functionality. Continuous delivery [HF10, HM11] is a rapidly emerging practice, aiming for drastically shortened software release cycles. Among several positive side effects[1] low risk releases are enabled by continuously delivering changes in small batches instead of deploying numerous changes at once in a ‘big bang release’ manner. Therefore, implementing continuous delivery significantly helps to improve software quality, which is of utmost economic importance. As Martin Fowler puts it[2] “as we push more

and more towards continuous delivery, continuous deployment, (and) features updated over the Internet all the time, that degree of being able to respond to change becomes important” for economic reasons. He continues: “if we do not put that effort on internal quality, we are in the end deceiving our customers, in fact stealing from our customers because we are slowing down their ability to compete”. This is why continuous delivery and the resulting improvements in software quality lead to a competitive advantage: “if I buy the product that is a hundred dollars cheaper and has low internal quality, I win at the moment. But what will happen is that the better internal quality software will be able to make new features (available) more and more rapidly, and soon the slow one cannot keep up anymore”, Fowler concludes. Consequently, additional costs most probably appear when initially implementing continuous delivery to achieve higher internal quality. However, this investment pays off because internal quality should not be considered tradable\(^1\) in order to stay competitive in the long term.

From a technical perspective, a high degree of automation is required to implement continuous delivery. This is achieved by automated continuous delivery pipelines (also known as deployment pipelines \[^{[HF10]}\]), consisting of different application-specific stages such as retrieving code from a repository, building packaged binaries, running tests, and deployment to production. Each stage requires a corresponding application environment such as a build environment and production environment. To achieve a high degree of automation, the deployment and management of these environments must be fully automated as part of the delivery pipeline. The automated deployment of application environments\(^2\) makes them consistent and predictable\(^3\); these are essential properties of continuous delivery pipelines. Among various organizational and technical challenges \[^{[Che15a]}\] that have to be considered when implementing continuous delivery, this dissertation focuses on specific challenges that appear when establishing and maintaining such environments \[^{[HF10]}\]. More specifically, it investigates

\(^{1}\) http://martinfowler.com/bliki/TradableQualityHypothesis.html

\(^{2}\) http://devops.com/2014/07/29/continuous-delivery-pipeline

\(^{3}\) http://drdobbs.com/tools/continuous-deliverys-biggest-challenge/240168437
how reusable solutions can be gathered in order to be utilized, combined, and orchestrated as building blocks of application environments. The orchestration of multiple solutions is typically required to implement the automated deployment of complex application environments. Different kinds of solutions such as Cloud-based infrastructure services, platforms (e.g., Amazon Web Services [SS15]), deployment scripts, and configuration definitions (e.g., Chef cookbooks [TV14]) are considered in this context. The orchestration of solutions is based on APIs that make their functionality available. While some solutions directly reveal their functionality through APIs, especially for lower-level solutions such as executable artifacts (e.g., deployment scripts), additional APIs need to be provided to efficiently combine them with other solutions. Providing and generating such APIs to enable the orchestration of diverse solutions is another specific focus of this work.

Although application environments are primarily motivated by continuous delivery pipelines, they are not exclusively relevant in this context. For example, a legacy application may consist of two environments only: a development environment and a production environment with manually performed build, test, and deployment steps. If such an application needs to be migrated to another infrastructure such as a Cloud-based infrastructure [MG11], a corresponding environment must be established based on the new infrastructure. The occurring challenges are similar to managing pipeline-bound application environments. Consequently, the core concepts and approaches presented in this work can be applied beyond continuous delivery. Some parts are even applicable beyond the scope of application environments, for example, to provide higher-level APIs (e.g., RESTful Web services [RAR13]) for lower-level executables (e.g., deployment scripts) that are used in various fields such as e-science [SHK+11]. Section 1.1 provides an overview of the addressed research challenges and the resulting contributions as part of this dissertation. Moreover, it points out which research contributions are domain-specific to continuous delivery and application environments and which are generic enough to be applied to other domains.
1.1 Research Challenges and Contributions

As outlined in Figure 1.1, the research efforts of this dissertation were driven by several research challenges in the scope of continuous delivery and application environments. By gathering diverse and reusable solutions to be used as building blocks of application environments, a comprehensive knowledge base is established, comprising these solutions. Initially, challenge A is to provide efficient means and mechanisms to collaboratively and continuously populate the knowledge base. Continuously populating the knowledge base is required to keep it current by adding new solutions and updating existing ones. Collaboration is key because diverse experts such as developers and operations personnel should be able to contribute different kinds of solutions.
to the same knowledge base. Therefore, the mechanisms for populating the knowledge base should both consider and respect established procedures employed by different kinds of experts. Complementing the manual process of adding specific expertise to the knowledge base, challenge B considers the fact that a broad variety of reusable solutions are already contained in different kinds of repositories such as Chef Supermarket\(^1\) and Docker Hub\(^2\). Because manually depositing all these solutions into the knowledge base does not scale and updates are missed easily, the automated discovery and capture of such solutions is another key challenge considered in this work. Since the main purpose of the knowledge base is to provide solutions as building blocks of application environments, challenge C concentrates on systematic knowledge utilization. Each environment such as a build environment that is part of a continuous delivery pipeline possesses specific environment requirements. For example, an environment requirement may be that a MySQL server version 5.7 is needed as part of the test environment. In order to resolve these requirements, queries need to be derived from them to be evaluated against the knowledge base. The knowledge base would then return solutions that satisfy the requirements. The Google Cloud SQL API\(^3\), the MySQL Docker container\(^4\), and the MySQL Chef cookbook\(^5\) are examples of different kinds of solutions that satisfy the previously mentioned MySQL server environment requirement. A specific subset of the retrieved solutions can be selected to be used as a foundation to create a deployable environment topology resulting from the technical refinement step. Assuming the knowledge base contains a significant number and variety of solutions, challenge D is to provide decision support for selecting an appropriate solution to resolve a particular environment requirement. This is especially important if a concrete technical solution must be selected from a substantial set of solution candidates retrieved from the knowledge base. Various aspects can be covered by a corresponding decision support approach such as the level of

---

1. \(\text{http://supermarket.chef.io}\)
2. \(\text{http://hub.docker.com}\)
3. \(\text{http://cloud.google.com/sql/docs/admin-api}\)
4. \(\text{http://hub.docker.com/_/mysql}\)
5. \(\text{http://supermarket.chef.io/cookbooks/mysql}\)
virtualization (hypervisor, shared kernel, etc.) of a particular solution or its fixed binding to a specific Cloud provider. Finally, *challenge E* is concerned with integrating diverse solutions at runtime (i.e., when creating instances of application environments) to enable their orchestration in the context of a particular application environment. Since lower-level solutions such as executable artifacts (deployment scripts, configuration definitions, etc.) do not directly provide their functionality through proper APIs, their orchestration with other solutions is challenging. Therefore, additional APIs must be provided to enable the orchestration of diverse solutions on different levels.

The described challenges are addressed by several research contributions as part of this dissertation. The following sections provide an overview of the research contributions as outlined in Figure 1.2.

Initially, **contribution 1** addresses *challenge A* and *challenge C* by providing a **method to implement continuous delivery using gathered solutions**. While the method itself is generic enough to be implemented in various ways, it conceptually covers all required steps in order to capture and maintain solutions, as well as to utilize and orchestrate them as building blocks of application environments as part of continuous delivery pipelines. Collaborative aspects are considered to enable diverse experts to work together.
through the knowledge base. Therefore, the method represents an overarching higher-level framework for the other research contributions to achieve the major goals of this dissertation. This contribution is domain-specific to continuous delivery. Moreover, it is covered by several of the peer-reviewed publications [WAL15a, WAL15b, WBKL16] listed in Section 1.2.

Next, contribution 2 elaborates on the concept of continuous delivery pipelines based on application environments, therefore essentially refining contribution 1 by tackling challenge A and challenge C on a more fine-grained level. A comprehensive metamodel centered around application environments in the context of continuous delivery is presented as a technical foundation of the knowledge base. It covers application environment requirements to drive the selection of specific solutions that eventually compose concrete application environment topologies. Furthermore, different kinds of relations between solutions are considered to express dependencies and conflicts. The concept of labels attached to solutions is utilized to specify technical capabilities. Requirements use such labels to enable the mapping between solutions’ capabilities and application environment requirements. This contribution is domain-specific to application environments, but could also be applied to scenarios dealing with environments beyond the scope of continuous delivery. Moreover, it is covered by several of the peer-reviewed publications [WAL15a, WAL15b, WBKL16, WBL16a] listed in Section 1.2.

Based on the previously outlined application environment metamodel and its labeling mechanism, contribution 3 aims to consider challenge D using a systematic classification of solutions to build application environments. This approach provides more fine-grained decision support for solution selection. Various classification dimensions for application environment requirements and solutions are discussed. These dimensions are the foundation of multiple label taxonomies. Hierarchical labels can then be employed to express the specific capabilities of solutions on varying levels of granularity. Furthermore, it is described how application environment requirements are resolved based on such labels to identify solution candidates. This contribution is domain-specific to application environments, but could also be applied to scenarios dealing with environments beyond the scope of
continuous delivery. Moreover, it is covered by several of the peer-reviewed publications \cite{WASL13, WASL14, WBL14a, WBL14c, WBL15b, WALS16} listed in Section 1.2.

Both challenge A and challenge B are concerned with populating the knowledge base, i.e., adding and updating solutions. These are addressed by contribution 4, covering the collaborative and automated gathering of solutions. The concept of collaborative solution repositories builds upon established collaborative software development approaches to enable diverse experts to work together to maintain solutions and their metadata. While this helps to manage the manually captured solutions, a complementary approach is presented to discover and retrieve solutions from existing repositories in an automated manner. Solution repositories aim to provide the foundation for systematically maintaining solutions. However, they do not provide fine-grained query mechanisms to efficiently find solutions that fit specific application environment requirements. Therefore, the concept of continuous delivery pipelines is adopted to build different kinds of knowledge base variants from the solution repositories. These variants provide corresponding query mechanisms. Although this contribution is applied in the context of continuous delivery and application environments, it is not domain-specific and thus generic enough to be applied to further contexts. Moreover, it is covered by several of the peer-reviewed publications \cite{WAL15a, WBKL16, WBFL16} listed in Section 1.2.

Moving toward runtime, contribution 5 aims to address challenge E by covering the API-driven orchestration of diverse solutions. APIs are utilized in the context of this work to orchestrate different kinds of solutions. Since some solutions such as lower-level executable artifacts do not directly provide their functionality through proper APIs, their orchestration with other solutions is challenging. Therefore, various approaches are described to address this issue by providing additional APIs, for example, by generating individual APIs for lower-level executable artifacts to make their functionality available through higher-level APIs. Although this contribution is applied in the context of continuous delivery and application environments, it is not domain-specific and thus generic enough to be applied to
further contexts. Moreover, it is covered by several of the peer-reviewed publications [WBB+13; WBB+14b; WBB+14a; WBL14b; WBL14c; WBL15a; WBL16b] listed in Section 1.2.

Finally, contribution 6 provides an integrated architecture, implementation, and validation, considering all of the previously described research challenges and contributions of this dissertation. The presented prototype implementations aim to show the feasibility of the presented approaches. Various validation scenarios based on these prototypes evaluate the approaches in the scope of continuous delivery and beyond. This contribution is covered by several of the peer-reviewed publications [WBL15c; WBL15b; SBLW15; WBKL16] listed in Section 1.2.

1.2 Scientific Publications

The following peer-reviewed journal and conference publications, as well as book chapters resulted from the research efforts related to this dissertation:


These scientific publications cover the previously described research contributions of this dissertation. In addition, further peer-reviewed publications of the author significantly contributed to the research results presented in this work:


1.3 Thesis Structure

The subsequent Chapter 2 presents fundamentals from various fields related to the approaches discussed in this dissertation. In addition, it points to related works and positions them in the context of the research efforts of
this work. The remainder of this thesis is structured following the order of the previously described research contributions. Chapter 3 covers contribution 1 by introducing a method to implement continuous delivery based on gathered solutions. Next, Chapter 4 presents the application environment metamodel, thereby covering contribution 2. Building application environment topologies based on the systematic classification of solutions (contribution 3) is discussed in Chapter 5. Concerning contribution 4, Chapter 6 describes various concepts that enable the gathering of diverse solutions in a collaborative and automated manner. Chapter 7 covers contribution 5 by presenting approaches to streamline the orchestration of diverse executables based on APIs. An integrated architecture, prototype implementations, and several validation scenarios (contribution 6) are discussed in Chapter 8. Finally, Chapter 9 concludes this dissertation by summarizing the research contributions, pointing out their capabilities and limitations, as well as outlining further research opportunities for potential future work.
The main purpose of this chapter is twofold: (i) key fundamentals are described to understand the concepts and approaches presented in this dissertation. Furthermore, (ii) related works are discussed in this context to outline how they differ from this work and to which degree they serve as a foundation for the presented concepts and approaches. To provide this information in a properly structured manner, each section of this chapter focuses on a specific field, discussing a selection of its relevant existing works.

2.1 Continuous Delivery

Continuous delivery [HF10; HM11; Che15a; Che15b] is a rapidly emerging paradigm and practice [Swa12; SRF13; GRH15; OE16; RHL+16], often considered in the context of release engineering [Nyg07; AM16]. It aims to significantly shorten software release cycles because users, customers, and other stakeholders especially in the fields of Cloud services, Web applications, mobile apps, and the Internet of things expect quick responses to changing
demands and occurring issues. Consequently, shortening the time to make
new releases available becomes a critical competitive advantage. In addition,
tight feedback loops involving users and customers based on continuous
delivery ensure to build the 'right' software, which eventually improves
customer satisfaction, shortens time to market, and reduces costs. Beside
cost reduction and accelerated release cycles, continuous delivery aims for
producing high-quality software in terms of functional and non-functional
properties, including performance, security, and user experience. Moreover,
recent research\textsuperscript{1} shows further side effects of continuous delivery: low risk
releases are enabled by delivering changes in small batches instead of doing
risky 'big bang releases' by deploying numerous changes at the same time.
Last but not least teams who adopt continuous delivery are usually happier
because releases are less painful and resulting feedback loops are better.
Consequently, teams can focus much better on improving the software the
way it is desired by its users.

Technically, continuous integration [DMG07] and proper configuration
management [GGL+09; DJV10; Hal13; uRah14; TV14; Ali15a] are two
fundamental building blocks and enablers for continuous delivery [HF10].
As a large-scale and prominent example, Tang et al. [TKV+15] describe how
holistic configuration management is done at Facebook\textsuperscript{2} to deliver their social
media platform based on their development and continuous deployment
process [FFB13; SDG+16]. A key aspect of configuration management is
to maintain the entire infrastructure through properly tested code artifacts
the same way as application functionality is implemented. This idea is
often referred to as \textit{infrastructure as code} [GHS10; HROE13; Mor16] and
\textit{test-driven infrastructure} [Nel13]. Thus, established development, version
control, and collaboration approaches can be used and shared to maintain
both application code and infrastructure code.

A high degree of automation is required as technical foundation to gain
from the benefits of continuous delivery. This is typically achieved by imple-

\textsuperscript{1} http://continuousdelivery.com/evidence-case-studies/#research
\textsuperscript{2} http://facebook.com
Figure 2.1: Continuous delivery pipeline example

A continuous delivery pipeline (CDP) is an individually tailored and maintained software system to automate the delivery process of an independently deployable unit.

An independently deployable unit can be, for example, a monolithic application or a microservice [New15]. Each pipeline consists of diverse application-specific stages such as build, test, and production. Figure 2.1 outlines an example for a typical pipeline consisting of different stages. The initial code commit stage happens during development, followed by the unit test stage to verify that the committed changes do not break any unit-level functionality of an application. If all unit tests pass, the build stage produces application-specific artifacts and bundles such as Java JAR files or Java WAR files. Next, the integration test stage verifies that these artifacts and bundles properly work in combination with each other. Additional test stages covering performance and acceptance tests are typically part of the pipeline to verify further aspects of the built application artifacts before putting them
into production as the final stage. Continuous delivery does not generally imply continuous deployment \cite{LMP+15} into production. The final step of putting a new version of the application into production (i.e., deployment) can also be a manual one. For example, some applications require a manual approval step for legal reasons before updates are put into production. However, the continuous delivery pipeline must ensure that there is always a stable and properly tested version built and available to be pushed into production. In case of continuous deployment, i.e., the pipeline includes the automated deployment into production, doing several deployments per day is common practice \cite{SDG+16}. The feedback loops shown in Figure 2.1 are key to continuously evaluate the results of the different stages and react quickly by committing changes to fix occurring issues.

If a large amount of independently deployable units exist, e.g., in case of a large-scale microservice architecture \cite{New15,Fam15,Ric15,BHJ16}, a large and growing number of individual pipelines have to be maintained. Therefore, an important focus of this dissertation is on systematically building such continuous delivery pipelines by gathering solutions as reusable building blocks and orchestrating them to compose a specific pipeline.

Fitzgerald and Stoll \cite{FS15} propose to widen the scope of continuous delivery toward continuous software engineering or continuous*. The presented research agenda aims to cover various additional aspects beside the actual software delivery process such as planning, budgeting, and innovation. This makes clear that continuous delivery and continuous software engineering is not only concerned with technical challenges such as maintaining highly automated pipelines, but also includes several organizational and cultural aspects. In this context, the DevOps paradigm \cite{EGHS16} is often discussed and applied as natural extension and evolution of established agile software development practices to make the collaboration between developers, architects, testers, operations personnel, and other stakeholders more efficient. These aspects are further discussed in the following Section 2.2.
2.2 Agile Software Development and DevOps

Agile software development practices [BBvB+01] such as Scrum and extreme programming [Mor15] helped to bridge gaps and merge roles on the development side. For example, developers are performing both programming and testing tasks, so a clear distinction between testers and programmers is typically not made in corresponding projects. However, the operations side is usually not targeted by these practices. By following the idea of agile infrastructure [Deb08], DevOps [Lou12; Wal13; SRF13; BWZ15; ZBC16] is often considered as the next logical step toward this direction by bridging gaps between developers and operations personnel such as system administrators [Hüt12]. Typically, cultural and organizational gaps between the different groups appear, so potentially incompatible goals are followed such as ‘push changes to production quickly’ on the development side versus ‘keep production stable’ on the operations side. This often results in contrary processes and mindsets, so a truly collaborative atmosphere cannot be established due to ongoing ‘blame games’, in which the different groups blame each other for occurring issues instead of collaboratively fixing the issues as fast as possible.

Recent reports [Pup15; SNP15] and research articles [Spi12; BCCD13; Roc13; BLY15; dBAC15; CS16; Spi16] confirm the relevance of DevOps and try to establish recommendations, guidelines, and best practices on how to successfully adopt DevOps. Depending on the structure and constraints of an organization, different approaches can be followed. An individual team could be responsible for the entire chain of a particular application or microservice [New15], from the code in a repository to deployment into production. Alternatively, if ‘dev’ and ‘ops’ groups are split organizationally, tightly integrated processes, shared repositories, and common tooling could improve their collaboration. An operations group may act as an in-house Cloud provider, offering managed infrastructure and platforms to developers through self-service portals and APIs. There is dedicated research on patterns and anti-patterns regarding DevOps team topologies¹ and the impact on

¹http://devopstopologies.com
mixing roles and responsibilities [NSP16]. To further improve reliability and resiliency of applications, DevOps and agile principles are often paired with complementary development and operations practices such as chaos engineering [BBdR+16]: it is based on the fact that everything can and will fail at some point in time. Consequently, applications and their components must be designed for failure [Tse13], so the overall system does not fail in case a particular part of the system fails. In addition, the chaos engineering approach aims to continuously ‘kill’ certain parts of the system on purpose such as running virtual servers that are part of the production cluster. This is to validate that the overall system does not fail, but properly recovers from such incidents. Netflix is a prominent example that shows how a large-scale video streaming platform can be maintained based on chaos engineering [BBdR+16].

Agile practices and DevOps do not only enable quick reactions to required changes regarding an application’s architecture, design, and implementation, but also the adaptation of the corresponding continuous delivery pipeline. This is required because pipelines are not static: if, for example, additional middleware or application components are added, it typically impacts multiple stages of a pipeline such as running an instance of the newly added database during test stages or installing new build tooling at build stage. Although continuous delivery and DevOps are two independent concepts, they are often combined in practice [HM11]: DevOps helps to align goals, as well as organizational and cultural constraints to prepare the effective implementation of continuous delivery by aligning the underlying technical processes through a high degree of automation.

2.3 Application Topologies, Deployment and Orchestration

As discussed previously in Section 2.1, a continuous delivery pipeline consists of different stages. Each stage relies on a specific application environment such as build, unit test, performance test, or production [HF10]. The auto-

\[1\] http://github.com/Netflix/SimianArmy
Figure 2.2: Application topology example for production environment

... automated delivery process of a pipeline requires such application environments to be deployed and managed in an automated manner. This part of the automation can be achieved by utilizing application topologies to specify these environments. Based on Binz [Bin15], an application topology in the context of this work is defined as follows:

**Definition 2.2 (Application Topology – informal)**

An application topology is a machine-readable model, which describes all technical components and their dependencies to enable the automated deployment and management of instances of the model. Therefore, the application topology also includes all required solutions such as executable artifacts in the form of deployment scripts, provisioning plans, virtual machine images, and configuration definitions.

The terms *application topology* and *application topology model* are used interchangeably in this work. It is explicitly emphasized when application topology instances are meant instead of models. In contrast to a higher-level application architecture, an application topology is technically fine-grained
enough to create application instances by deploying the application topology model. Especially when implementing continuous delivery pipelines with different stages and underlying application environments, multiple topologies exist. This is because specific application environments for building and testing the application are typically different from the ‘final’ topology of the application running in production. The Topology and Orchestration Specification for Cloud Applications (TOSCA) [OAS13b; BBLS12; BBKL14b; KML+14; MCE15] is a prominent example for a standardized metamodel and language to specify graph-based topology models. Further related works in this field include CloudML [FSR+14; BRF+15], enterprise topology graphs [BFL+12], Engage [FME12], MADCAT [INS+14], and the Cafe approach [MUL09; Mie10]. Moreover, provider-specific solutions such as Amazon’s CloudFormation and tooling-specific approaches such as MCollective [Rhe14], Terraform and Juju exist to manage application topologies. Bergmayr et al. [BWKG14] compare a selection of Cloud modeling approaches including several application topology modeling languages by providing concrete examples for each approach. Figure 2.2 uses the Vino4TOSCA notation [BBK+12] to outline an example topology for a Web shop application, representing the application’s production environment, consisting of the core application and a database. Both parts are clustered and hosted on the Cloud-based and elastic Amazon EC2 infrastructure to provide a scalable and highly available instance of the application. Further application topologies cover the other pipeline stages. Some stages such as build and unit test may share the same application topology, for example, to make the pipeline more efficient in terms of speed and resource consumption. This is because a reduced number of topology models must be instantiated and managed along the pipeline.

The automated processing of application topologies is primarily enabled by orchestrating various deployment automation solutions, for example,
executable artifacts such as deployment scripts or configuration definitions that are attached to the topology model [BBK+13]. Each node of a topology graph (Figure 2.2) typically has a corresponding solution attached to it to run its deployment and further management activities in an automated manner. A broad variety of deployment automation and orchestration approaches exist. Some of these approaches are based on key properties of the Cloud computing paradigm [AFG+10; MG11; Wil12; FLR+14], including self-service capabilities in the form of Cloud provider APIs, on-demand provisioning of resources such as virtual servers and storage, and fine-grained pay-per-use options. While it is challenging for Cloud infrastructure providers to run a large-scale datacenter in a cost-efficient manner [BCH13], users of Cloud offerings can entirely focus on deploying their applications and workloads without worrying about datacenter management. Deployment automation solutions can be positioned on different levels corresponding to the Cloud computing service models infrastructure-as-a-service (IaaS) and platform-as-a-service (PaaS) [MG11]. Infrastructure-level solutions are often centered around virtual servers: beside provider-specific offerings such as Amazon EC2 [RS15] and Microsoft Azure [Bai14], provider-agnostic open-source tooling such as Vagrant [Has13], Packe1 and Fog2 can be utilized to manage virtual servers and other infrastructure resources. Standards such as the Open Virtualization Format (OVF) [ACC+14; DMT15] are supported by major vendors like VMware [Kum15] to improve portability and interoperability. Open-source Cloud management frameworks such as OpenStack [Pep11] and CloudStack3 can be used to operate private or hybrid Cloud infrastructures as part of application topologies. Configuration management solutions such as Chef [TV14; SW14], Puppet [TM11; Loo11], Ansible [MR14; Hoc14], and CFEngine [Zam12] form another category of deployment automation approaches. They typically provide domain-specific languages to specify configuration definitions to install and configure infrastructure resources.

1 http://packer.io
2 http://fog.io
3 http://cloudstack.apache.org
Moving from plain infrastructure toward the platform level, containers \cite{Vau06, SPF+07, FFRR14} provide a fine-grained virtualization and delivery mechanism to package specific middleware and application components, for example, in the form of microservices \cite{New15}. For example, Docker \cite{Tur16, Mou15, Kar15, And15} provides a broad ecosystem of tooling and reusable containers\footnote{http://hub.docker.com} paired with a highly active open-source community\footnote{http://docker.com/docker-community}. The Open Container Initiative (OCI)\footnote{http://opencontainers.org} is an ongoing effort to standardize a common application container format, as well as a fundamental open-source toolchain to manage such containers. Furthermore, diverse container orchestration and cluster management solutions such as Kubernetes \cite{BGO+16} and Docker Swarm\footnote{http://docker.com/products/docker-swarm} are emerging to run containers at large scale. In this field, the Cloud Native Computing Foundation (CNCF)\footnote{http://cncf.io} is another ongoing standardization effort based on Kubernetes and further open-source solutions to streamline the way how containers are managed on a higher level.

Moving to the next level of abstraction, PaaS solutions aim to provide ready-to-use runtime environments, database back-ends, and other building blocks required as part of an application topology. Several provider-specific offerings such as Heroku \cite{Cou14}, Amazon Elastic Beanstalk \cite{VPWD11} and Google Cloud Platform \cite{KG15, Ali15b} serve APIs to utilize the deployment and management functionality of their platforms. Open-source PaaS frameworks such as OpenShift \cite{Shi14} and Cloud Foundry\footnote{http://cloudfoundry.org} can be used to build and run private PaaS environments, for example, inside a specific organization. These frameworks aim to reduce vendor lock-in by making it easier to move application components between different PaaS environments that rely on the same framework. Therefore, Cloud providers such as IBM Bluemix\footnote{http://bluemix.net} based on Cloud Foundry try to differentiate themselves from other established PaaS
providers such as Heroku by advertising an improved portability. However, this holds true for the core platform only because once provider-specific services are used that do not exist in other PaaS environments, vendor lock-in is still a major issue.

Different kinds of solutions (infrastructure, containers, PaaS, etc.) can be mixed and attached to an application topology, for example, to implement the automated deployment of all components that are involved. Multi-cloud applications \([\text{LKS+11; Pet14; FSR+14; CCP14; PMM15}]\) can be deployed this way. Such applications are based on multiple Cloud environments, i.e., a public and private Cloud infrastructure, or combination of public Cloud providers. As an example, an application’s database can be hosted on a private Cloud infrastructure to avoid storing sensitive data externally; however, the application’s front-end and business logic may run in a public Cloud environment to improve scalability and to reduce operational costs. Corresponding application topologies contain and utilize diverse deployment automation solutions to cover all Cloud environments that are involved. Moreover, \([\text{Breitenbücher}]\) presents a pattern-based approach to automate the management of application topologies. Technically, management plans are dynamically produced by combining specific ‘management planlets’ (i.e., reusable plan fragments) to achieve a particular management goal such as migrating a deployed application component to a different infrastructure. While the management of application topologies that occurs after its deployment is not addressed by this dissertation, the underlying planlets also rely on diverse solutions to implement management logic. Since deployment plans are a specific kind of management plans, this approach can be utilized to generate deployment plans that are executed to deploy application topologies.

2.4 Application Programming Interfaces (APIs)

Many of today’s applications, especially Web applications, as well as back-end systems and platforms for mobile apps, provide application programming
The main purpose of an API is to provide a well-defined and documented interface, which is supplied to access and utilize application functionality in a programmatic manner. APIs hide and abstract from implementation-specific details such as invocation mechanisms and lower-level data models inherited from the technology stack on which a particular application is built upon. This is the foundation in the context of this work for integrating and orchestrating different applications and application components, enabling systematic development and reliable operations of distributed applications, mash-up applications, and mobile apps. Furthermore, APIs are used to integrate applications with business partners, suppliers, and customers. Even devices can be interconnected to provide the technical foundation for the emerging Internet of things (IoT).

Technically, APIs can be provided and utilized in different forms. (i) Libraries and toolkits that are bound to a particular programming language, as well as (ii) language-agnostic Web services, e.g., HTTP-based REST APIs and WSDL/SOAP-based services are widespread forms of providing and using APIs. Concerning the terminology used in this work, a SOAP API is a Web service API that is specified using WSDL and provides a binding based on the SOAP messaging framework.

Web service APIs play a key role in service-oriented architectures (SOA) with their enterprise service bus (ESB) integration style and microservices as a ‘modernized flavor’ of SOA. Popular providers such as Twitter, GitHub, Facebook, and Google offer such libraries and Web services. However, libraries and Web services are not mutually exclusive, meaning libraries often use Web services in the background, but adding an additional layer of abstraction to seamlessly integrate with the programming model of the corresponding language. Consequently, APIs often serve as platform-independent and language-agnostic means for integration and orchestration purposes, optionally enhanced by additional language-specific libraries.

The number of publicly available APIs is constantly growing. As of today,
API directories such as ProgrammableWeb\(^1\), PublicAPIs\(^2\), and APIs.io\(^3\) list a huge variety of APIs. Popular providers such as Google, Facebook, and Twitter are serving billions of API calls per day.\(^4\) These statistics underpin the importance and relevance of APIs. Existing literature \([\text{Mas11}; \text{RAR13}]\) and frameworks such as Hapi\(^5\) (Node.js) and Jersey\(^6\) (Java) provide holistic support, best practices, and templates for building APIs. While this is state of the art for creating Web applications and back-ends for mobile apps, APIs as a platform-independent and language-agnostic means for integration and orchestration purposes are heavily utilized for automating the deployment and management of Cloud applications \([\text{MG11}; \text{WBB+14b}]\). This leads to significant cost reduction and enables applications to scale: Cloud providers offer management APIs that can be programmatically used in a self-service manner, e.g., to provision virtual servers, deploy applications using platform services, or to configure scaling and network properties. Beside provider-specific APIs, various standardization efforts aim to align and establish corresponding technical interfaces on different levels. For example, the Cloud Infrastructure Management Interface (CIMI) \([\text{DMT13}]\) and the Cloud Data Management Interface (CDMI) \([\text{SNI15}]\) focus on the infrastructure and storage layer. On the platform-as-a-service (PaaS) level, the Cloud Application Management for Platforms (CAMP) specification \([\text{OAS14}]\) aims “to enable interoperability among self-service interfaces to PaaS clouds by defining artifacts and formats that can be used with any conforming cloud and enable independent vendors to create tools and services that interact with any conforming cloud using the defined interfaces”.

However, because such management APIs typically focus on providing basic functionality, they are combined with further configuration management systems to realize non-trivial deployment scenarios: a huge number of reusable executable artifacts such as scripts (e.g., Chef cookbooks \([\text{Nel13}]\),

\(^1\)\url{http://www.programmableweb.com/apis/directory}
\(^2\)\url{http://www.publicapis.com}
\(^3\)\url{http://apis.io}
\(^4\)\url{http://slideshare.net/jmusser/j-musser-apishotnotgluecon2012}
\(^5\)\url{http://hapijs.com}
\(^6\)\url{http://jersey.java.net}
Juju charms\(^1\), Unix shell scripts) and templates like Docker container images \(^{[Tur16]}\) are shared by open-source communities to be reused in conjunction with provider-supplied services. As part of this work, such reusable artifacts are gathered as solutions and managed through a knowledge base. While APIs are orchestrated based on well-known and common protocols such as HTTP, the technical integration with these different artifacts and heterogeneous management systems is an error-prone, time-consuming, and complex challenge \(^{[BBK+13; WBB+14b]}\). Thus, to build, deploy, and manage non-trivial applications, it is of vital importance to handle the invocation of different artifacts, technologies, and service providers in a technically uniform manner to focus on the orchestration level, neglecting lower-level technical differences \(^{[WBL15a]}\). Therefore, a major goal of this work is to provide additional APIs, for example, by generating individual APIs for such executable artifacts, which do not directly serve their functionality through an API. This is to simplify the combined usage of existing and additionally provided APIs by focusing on the orchestration of these APIs instead of dealing with lower-level heterogeneous executable artifacts at the orchestration level.

Regarding the aspect of generating APIs, various related works \(^{[GEM06; CFFT08; PG08; ACD10]}\) in the field of SOA focus on modernization and migration of legacy applications by exposing their functionality through Web service APIs. However, those approaches possess limitations that are addressed by this work. More specifically, the approach for generating APIs presented in this work aims to explicitly cover the following aspects: (i) API implementations such as Web service wrappers are packaged in a self-contained and portable manner containing the entire stack from the API endpoint implementation to the underlying executable artifact such as a deployment script. (ii) Consequently, generated API implementations do not require any special or centralized middleware component to support the invocation and usage of such executables; an API implementation can run anywhere using a general-purpose virtualization environment such as

\(^{1}\text{http://juju charms.com}\)
Docker [Mou15] or Vagrant [Has13]. (iii) The kind of API is not limited to SOA-related approaches such as SOAP [W3C07a] and WSDL [W3C07b]. It can be arbitrarily chosen depending on preferences, existing expertise, utilized orchestration tooling, etc. Moreover, the kind of API can be changed on demand, for example, when shifting the orchestration approach from BPEL workflows (invoking SOAP APIs) to scripts (invoking REST APIs). (iv) Finally, since the API is not part of the executable’s implementation, separation of concerns can be properly achieved by keeping the executable focused on the actual logic to be executed such as the deployment logic of specific application components. Dynamically generated APIs aim to significantly improve both accessibility and usability of the underlying executable in various contexts and environments.

2.5 Classification of Solutions and Implementations

Taxonomies and ontologies are commonly used to systematically classify and categorize concepts belonging to a certain domain such as Cloud computing [AVS12]. Such classification approaches help a lot when comparing different Cloud providers and their offerings to find out which solutions fit the specific requirements of an application that should run in a Cloud-based environment. For example, Dukaric and Juric [DJ13] propose a unified taxonomy for Cloud-based infrastructure-as-a-service (IaaS) frameworks and architectures. The proposed taxonomy was derived from various existing IaaS solutions such as Amazon Web Services [WW15; Vya14], OpenStack [Pep11], and VMware [Kum15]. At its core, the taxonomy is centered around several layers that are key to IaaS architectures: management layer, resource abstraction layer, security layer, etc. The NIST definition of Cloud computing [MG11] describes three major service models to be used to categorize Cloud-based solutions: infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). IaaS resources include basic computational and storage resources such as virtual servers and virtual disks. PaaS solutions aim to abstract from the lower-level infrastructure level
and provide runtime environments and database-as-a-service offerings to immediately deploy an application without manually managing servers. SaaS offerings are ready-to-use applications, usually Web applications that can be used through a Web browser without deploying or installing any software. Since these service models are coarse-grained, Kächele et al. [KSHD13] propose an extended Cloud taxonomy, including categories such as hardware-as-a-service, operating-system-as-a-service, and filesystem-as-a-service on the IaaS level, as well as runtime-as-a-service, framework-as-a-service, and data-mapping-as-a-service on the PaaS level. Similarly, Strauch et al. [SKLU11] extend NIST’s basic entities to provide a more fine-grained taxonomy for Cloud data hosting solutions. These refined taxonomies are then applied to existing Cloud services such as provided by Amazon Web Services [WW15]. Further efforts in this field follow a similar approach by refining NIST’s basic service models and map these to existing Cloud providers. This enables a more systematic comparison of provider offerings. Hoefer and Karagiannis [HK10] propose another taxonomy to categorize services offered by different Cloud providers. This taxonomy focuses on higher-level service properties such as payment modes and service level agreements (SLAs). Technically fine-grained properties such as APIs, SDKs, and command-line interfaces to actually utilize and orchestrate services are not considered.

As part of various Cloud-related research projects, further taxonomies and ontologies were proposed and utilized to systematically classify and compare Cloud solutions. For example, Quinton et al. [QHRD13] outline and use ontologies to map features provided by different Cloud environments. The motivation for this approach are applications that are distributed across multiple Cloud environments (multi-cloud deployments). Feature models are created for diverse Cloud environments; these feature models are then mapped using the proposed ontologies to enable the matchmaking between Cloud environment capabilities and requirements of multi-cloud application. This helps to find out which part of a specific application can run in which environment. The ontologies cover different kinds of resources

1 [http://theenterprisearchitect.eu/blog/categorize-compare-cloud-vendors](http://theenterprisearchitect.eu/blog/categorize-compare-cloud-vendors)
on the IaaS and PaaS level such as virtual servers, application servers, and databases. Similarly, [MAM+11; MAD+11] present the mOSAIC ontology, which builds on existing Cloud taxonomies. A key motivation behind the mOSAIC ontology is to improve the interoperability of Cloud applications. For example, the ontology provides the foundation for migrating an application or parts of it to another Cloud environment. The mOSAIC ontology is the basis for further efforts in the field of Cloud interoperability such as an ‘inter-cloud ontology’ and an ‘inter-cloud resource catalogue’ [DCE14; DCE+15]. Further taxonomies such as the ‘taxonomy of inter-cloud challenges’ proposed by Toosi, Calheiros, and Buyya [TCB14] are positioned on a conceptual level to provide higher-level decision support. This taxonomy covers various aspects such as data portability, network connectivity, and legal issues to drive both technical and non-technical decisions to be made when comparing and choosing Cloud solutions. Beside Cloud-centric taxonomies and ontologies, classifications of different kinds of middleware solutions such as message-oriented middleware and application servers exist [Tho97; Ibr09]. Their goal is to systematically categorize and position various types of middleware regarding their specific properties such as the communication model (request/reply vs. message queuing), program execution (blocking vs. non-blocking), or communication mode (synchronous vs. asynchronous). Since many applications rely on such middleware solutions, they are essential building blocks of Cloud environments beside infrastructure-level solutions. Similar efforts exist in the field of database solutions. For example, Sadalage and Fowler [SF12] propose a classification of NoSQL databases.

Cloud providers and software vendors such as Heroku\(^1\) and New Relic\(^2\) provide a basic categorization of selected solutions and implementations. They do not only include Cloud-based middleware and infrastructure services, but also deployment, monitoring, and further supplementary tooling to support application development, operations (DevOps), and continuous delivery. Therefore, such supplementary tooling is sometimes referred to as DevOps tooling. Beyond the categorization and classification of different

---

\(^1\)http://elements.heroku.com/addons
\(^2\)http://newrelic.com/devops/toolset
kinds of solutions, Magoutis et al. [MPP+15] propose the usage of a social networking platform for Cloud deployment specialists. Such a specialized social network can be used to drive the decision making process which solutions to choose based on experiences and recommendations provided by experts.

The overarching goal of this dissertation is gathering different kinds of solutions (infrastructure, middleware, and supplementary DevOps tooling, also referred to as DevOpsware in this work), as well as providing APIs to simplify their orchestration. These solutions can then be utilized, for example, to build continuous delivery pipelines with their different stages and underlying application environments. Some of the previously described classifications are used, extended, and combined in this work to establish a comprehensive and technically fine-grained taxonomy that helps to perform the matchmaking between solutions’ capabilities and application environment-specific requirements. However, in contrast to some of the related works this work does not exclusively focus on Cloud-related solutions, but aims to cover all solutions that can be used as building blocks of application environments.

2.6 Content Discovery and Crawling

The basic idea of conventional Web crawling follows a straightforward process: “(1) select a URL to crawl, (2) fetch and parse page, (3) save the important content, (4) extract URLs from page, (5) add URLs to queue, and (6) repeat” [Mat14]. A broad variety of crawling approaches exist both in research and industry. The basic concepts of Web crawling are well understood and implemented by several open-source libraries and frameworks [Mat14]. Consequently, implementing a small-scale crawler, for example, to fetch a fixed and limited set of documents from the Web and store them ‘as is’ is basically a pure programming challenge. However, it is not trivial to implement large-scale crawlers, which repeatedly fetch large sets of documents, semantically inspect and normalize their content, detect updates compared to previous versions, and classify these documents. Therefore, several research
efforts focus on how to design and establish highly scalable and distributed crawlers to achieve good performance for such large-scale crawling scenarios \( [\text{BCSV04; DVG} + 99; \text{SS02; HN99; The01; EMT01}] \). Two major categories of crawlers can be identified: (i) general-purpose crawlers that potentially fetch and inspect any kind of document, for example, to build an index for a search engine; (ii) specialized and focused crawlers \( [\text{CVD99}] \), which only fetch and inspect documents belonging to certain topics. Document classification techniques are typically applied to rate how relevant a particular document is with regards to a given topic. Focused crawling is utilized, for example, to establish a domain-specific knowledge base. General-purpose crawlers often serve as a foundation to collect large sets of data, which are then analyzed using data mining techniques \( [\text{Mat14; The01}] \).

A specific goal of this dissertation is the automated capturing of a large number of domain-specific solutions such as reusable deployment scripts, virtual machine images, container images, and similar artifacts. These solutions are fed into a knowledge base to serve as reusable building blocks for building specific application environments. Therefore, a domain-specific, modular, and targeted content discovery and gathering framework is presented based on the previously outlined existing works.

### 2.7 Chapter Summary and Discussion

An important focus of this dissertation is on systematically building continuous delivery pipelines based on the orchestration of gathered solutions as reusable building blocks. Therefore, the rapidly emerging paradigm of continuous delivery and existing works in this field were initially discussed as part of this chapter. Beside the technical aspects of continuous delivery such as maintaining highly automated deployment pipelines, organizational and cultural issues must be addressed, too. Although continuous delivery and the implementation of corresponding pipelines does not inherently depend on specific development and operations methods, agile software development and DevOps practices often provide the required foundation. Thus, the state
of the art of these practices was shortly outlined. This dissertation mostly focuses on the technical aspects of continuous delivery pipelines, consisting of multiple application-specific stages: build, test, production, etc. Stages rely on application environments such as a build environment and a test environment. To enable fully automated pipelines, these application environments are specified as application topologies, which contain and utilize various deployment automation solutions. The orchestration of such solutions using topology models eventually enable the automated deployment of the required application environments as part of a pipeline. Therefore, the key concepts of application topologies combined with diverse deployment automation approaches and their orchestration were discussed in this chapter. Since the orchestration of diverse solutions such as executable artifacts (deployment scripts, configuration definitions, etc.) is a major challenge addressed by this dissertation, a separate section elaborates on application programming interfaces (APIs) and how they could ease the orchestration. A specific goal in this context is to provide additional APIs for executable artifacts that do not directly serve their functionality through an API.

In order to create application topologies for the diverse application environments of a continuous delivery pipelines, existing solutions are gathered as reusable building blocks. A systematic classification of such solutions is required to identify candidates that fit. Thus, existing and related classifications such as taxonomies and ontologies were presented to serve as a foundation for categorizing solutions as part of this work. However, a significant set of existing classifications focus on the middleware and infrastructure dimensions, so DevOps- and continuous delivery-related solutions are considered only rudimentary. Moreover, the presented taxonomies and ontologies are often not fine-grained enough to provide solution candidates for application environments. For example, Amazon EC2\(^1\) as a provider offering is often considered as a solution on the most fine-grained level. However, in order to use Amazon EC2 as part of a specific application environment, a more fine-grained technical solution such as the Amazon Java SDK\(^2\) or the Amazon

\(^1\)http://aws.amazon.com/ec2
\(^2\)http://aws.amazon.com/sdk-for-java
EC2 API\(^1\) is required. These deficiencies are addressed as part of this work by proposing refined and more fine-grained taxonomies for categorizing diverse solutions. Manually collecting and classifying such solutions on a large scale is extremely time-consuming and error-prone. Therefore, content discovery and crawling approaches such as Web scraping were discussed to provide the foundation for the automated gathering of domain-specific solutions as part of this work. The planned solutions gathering approach is targeted because the crawling targets are well-defined repositories and registries (Chef Supermarket\(^2\), Docker Hub\(^3\), etc.) containing reusable deployment scripts and container images. Beside building on established Web scraping techniques, open-source crawling libraries such as Scrapy\(^4\) and X-Ray\(^5\) are technically utilized to implement the automated gathering approach by capturing mostly structured data that are provided in different formats including JSON and XML.

---

Continuous delivery of software applications is implemented using pipelines as described in Section 2.1. Each stage of such a continuous delivery pipeline possesses specific requirements regarding their underlying application environment (build, unit test, performance test, production, etc.). Consequently, non-trivial pipelines are typically based on a set of diverse application environments to run their different stages. Building these environments requires specific combinations of solutions on different levels such as Cloud provider APIs, virtual server images, deployment scripts, as well as technical tutorials and documentation. Some of them are application-specific such as configuration scripts for custom application components. These highly specific solutions typically need to be developed and maintained individually for an application. Individual solutions can either be developed from scratch or by cloning and modifying an existing solution that is similar to the required one. Beside these application-specific solutions, a huge variety of reusable solutions such as Cloud provider APIs, as well as configuration management tooling exist. Examples on the infrastructure level include APIs and SDKs to
provision virtual servers, as well as deployment scripts to install and configure common middleware components such as relational database servers. Similarly, solutions provided on the platform level include, for example, APIs to access and manage database-as-a-service instances. The major goal of this work is to enable the systematic search, selection, and reuse of as many solutions as possible to be used as building blocks of application environments of continuous delivery pipelines.

3.1 Gather’n’Deliver Method

To establish a basis for the efficient reuse of diverse solutions to implement application delivery processes, the Gather’n’Deliver method is presented in this section. Figure 3.1 outlines the individual steps of the method, which address the issue of systematically capturing and consolidating knowledge in the form of reusable solutions. These solutions are then utilized to implement continuous delivery pipelines. As depicted in Figure 3.1, a knowledge base connects the two major phases of the Gather’n’Deliver method: (i) gathering solutions as input for the knowledge base and (ii) utilizing this knowledge to build pipelines for continuously delivering software applications. Steps A to C make up the gather phase, whereas the deliver phase comprises steps 1 to 8:

(A) Discover solutions for application environments

Initially, existing solutions have to be discovered, which represent potential building blocks of application environments. These solutions can be of various kinds such as Cloud management toolkits or APIs, virtual machine images, deployment scripts for common middleware components, tutorials on how to run certain components on a specific infrastructure, etc. Some solutions are therefore immediately machine-processable such as scripts and images, while others may need manual effort to utilize them, for example, by implementing a script based on instructions given by a tutorial. The sources of such solutions may include public repositories.
of open-source communities such as the Chef Supermarket\footnote{http://supermarket.chef.io} provider-specific offerings as, for example, the AWS Marketplace\footnote{http://aws.amazon.com/marketplace} and existing implementations that are internally used in certain organizations.

(B) Capture solutions in knowledge base

Discovered solutions are captured inside a knowledge base in conjunction with metadata. These metadata provide knowledge on how particular solutions can be employed, for example, by specifying their dependencies and configuration parameters to enable their usage for different
environments. Capturing solutions may occur in an automated manner based on corresponding tooling, but can also be performed by experts in a semi-automated or manual way. As an example, system administrators and operations personnel may populate the knowledge base with self-developed deployment scripts that are reusable for different applications in a certain context such as a specific company or team.

(C) Link solutions and refine their metadata

In order to make the knowledge base more than a plain database holding a flat set of solutions, links between solutions are established. Beside dependency links, these include recommendation links and conflict links, which point to other solutions. This is to express that a particular solution plays well with a second, but does not function in conjunction with a third. Maintaining and refining links is an iterative and ongoing task carried out by experts and users of the knowledge base, for example, based on gained experience. While these links are a vital aspect of the solutions’ metadata, further information is attached to a solution. For instance, additional metadata may refine the classification of a particular solution, denoting that it can only be used in a certain region in the world or in conjunction with a specific provider.

(1) Design and implement application

Creating the software application is the initial step, which must be performed to deliver a specific application to its users. Typically, corresponding experts such as software architects and developers collaborate to fulfill this task. This step forms the foundation for establishing a pipeline to continuously deliver new iterations of the application.

(2) Design delivery pipeline with its environments

The pipeline that implements the continuous delivery process for a particular application is designed in this step, including its different stages (build, unit test, performance test, production deployment, etc.). Each
stage is based on a specific application environment. As an example, a build environment running on-premise (e.g., aiming for minimum resource usage) may be completely different from a production environment hosted in a Cloud environment (e.g., focusing on scalability and high availability).

(3) Define environment-specific requirements

The pipeline design, which was created as part of the previous step, is now refined by defining environment-specific requirements. These requirements specify which building blocks are needed to establish a corresponding application environment. For example, a Java-based Web application requires a Java Virtual Machine and Maven\(^1\) for its build environment; additionally, JUnit\(^2\) is required for its unit test environment.

(4) Select solutions to resolve requirements

The previously specified requirements are then resolved by selecting solutions from the knowledge base. These solutions are responsible for technically establishing the diverse environments assigned to the stages of an application’s delivery pipeline. For example, various solutions such as Google App Engine’s Java runtime\(^3\) or the Docker image \textit{maven}\(^4\) can be utilized to satisfy the environment-specific requirements of a Java-based Web application, which aims for continuous delivery based on a set of environments.

(5) Adapt, refine, or create solutions (optional):

In some cases, existing solutions captured in the knowledge base do not immediately fit the environment-specific requirements. Depending on the reason for this issue, it can be addressed in three different ways. (i) In the simplest case, an existing solution is \textit{adapted} to fit the given

\(^{1}\) \url{http://maven.apache.org}
\(^{2}\) \url{http://junit.org}
\(^{3}\) \url{http://cloud.google.com/appengine/docs/java}
\(^{4}\) \url{http://hub.docker.com/_/maven}
requirements. As an example, an existing installation script that is only capable of installing Java version 7 needs to be updated to install Java version 8. (ii) If a solution fits the requirements, but is not immediately executable, it has to be refined. For example, a tutorial may provide a technically detailed solution to deploy a scalable relational database management system in a certain Cloud environment; however, the individual steps must be codified as a deployment script to provide an executable solution, which can then be used to provision application environments. (iii) In the worst case, no solution fits the given requirements, so a new solution must be created from scratch. This may be the case for highly application-specific parts of an environment. Updated and newly created solutions could finally be fed back into the knowledge base if their reuse is intended.

(6) Orchestrate diverse solutions

The output of the two previous steps is a set of solutions per application environment. These solutions satisfy the previously defined environment-specific requirements. Orchestrating these solutions connects the individual building blocks and allows the establishment of application environments. The orchestration may be technically implemented using declarative application topology models or imperative orchestration plans such as executable scripts or workflows [LR00]. Additional glue code may be required to combine the selected building blocks of a particular environment.

(7) Build pipeline by wiring environments

In this step, the application environments are wired in accordance with the different stages of the previously designed pipeline (step 2). This can be technically implemented by an overarching continuous delivery plan (e.g., script or workflow), which includes the provisioning of corresponding application topologies in order to implement the application delivery process. Alternatively, dedicated continuous delivery tooling or
Cloud-based continuous delivery services can be utilized for this purpose.

**8. Operate pipeline**

Finally, the pipeline is operated, providing the automated and continuous delivery of updated versions of a particular application. In addition to the delivery logic, monitoring and analytics features may be added to the pipeline to check and analyze the status of activities occurring in the different application environments.

Both the *gather* and the *deliver* phase are intended to run in parallel, yet in a decoupled manner. As an example, new solutions are added to the knowledge base (step B) and captured ones are refined (step C), while certain solutions are selected from the knowledge base (step 4) and used to satisfy the environment-specific requirements of potentially multiple pipelines that deliver different applications. Moreover, the method is not intended to be applied only once. The knowledge base is populated iteratively with new and refined solutions, as well as updated links between them. Similarly, applications delivered using a continuous delivery pipeline, are typically developed and maintained using agile principles [BBvB+01]. As a result, the pipeline must be continuously adapted to changing architectures and deployment topologies of applications.

The given order of the individual steps of the *Gather’n’Deliver* method is not fixed, but rather provides orientation for the process. It is possible to return to a previous stage in the process, for example, from selecting solutions (step 4) back to defining environment-specific requirements (step 3), or even further back to designing the pipeline (step 2). This may be necessary if appropriate solutions cannot be retrieved from the knowledge base. In this case, the pipeline design and the underlying environment requirements are adapted correspondingly to better reuse existing and proven solutions. Regardless of the order, some steps may happen in parallel, for example, implementing the application (step 1) and designing the pipeline (step 2).

The filled shapes shown in Figure 3.1, i.e., steps A, B, C, 3, 4, 6, and the knowledge base are the focus of this work toward the major goals of
(i) capturing and finding appropriate solutions, as well as (ii) simplifying their usage and orchestration to build application environments as part of continuous delivery pipelines.

3.2 Collaboration among Roles and Organizations

As depicted previously in Figure 3.1, the knowledge base forms the connection between the *gather* and *deliver* phases of the GATHER’n’DELIVER method. Therefore, the knowledge base enables the collaboration between experts working in different roles in the context of the GATHER’n’DELIVER method. In particular, multiple collaboration dimensions are covered:

**Collaboration between developers and operations personnel**

The GATHER’n’DELIVER method supports implementing the DevOps paradigm [BWZ15; Hüt12] inside an organization by bridging the gap between the developers and operations personnel responsible for an application. They work closely together on creating, selecting, and maintaining the solutions required by specific application environments. In addition to application developers, software architects may utilize the knowledge base at an early stage in the process of creating the application design and architecture. This is to ensure that required solutions are available for establishing corresponding application environments subsequently and may thus drive the decision making for selecting certain middleware or infrastructure solutions.

**Collaboration between maintainers of different applications**

Different applications are often maintained (i.e., developed and operated) by different people. However, some applications share similar stacks in terms of their utilized middleware and infrastructure, especially within a particular organization. The knowledge base aims to preserve reusable solutions, which can be utilized in different application stacks. Thus, the collaboration between maintainers of different applications is enabled
by the **Gather’n’Deliver** method. As an example, operators of diverse application stacks can collaboratively create and maintain solutions for commonly used middleware components.

**Collaboration between open-source communities and enterprises**

Diverse open-source communities share a broad variety of reusable solutions, for example, in the form of deployment scripts (e.g., Chef Supermarket\(^1\)) or container images (e.g., Docker Hub\(^2\)). Enterprises are interested in utilizing this content, and some are even willing to contribute improvements back to the open-source communities as in the case of companies such as Netflix in the scope of their open-source initiatives\(^3\). The knowledge base promoted by the **Gather’n’Deliver** method can be implemented in a distributed manner, therefore solutions originating from diverse sources (e.g., open-source communities, enterprise repositories) can be aggregated and combined. With this approach, the collaboration between enterprises is covered, too. This could be achieved if several companies follow the open-source approach to share solutions in this context, which currently happens in practice. OpenStack\(^4\) and the Cloud Native Computing Foundation\(^5\) with Kubernetes\(^6\) are prominent examples. Alternatively, directly shared parts of the knowledge base between certain enterprises can be maintained to avoid open-sourcing potentially confidential solutions.

Typically, there is a significant overlap of roles owned by people, who are collaborating as part of the **Gather’n’Deliver** method. For instance, authors or discoverers of solutions typically use and apply these solutions to implement continuous delivery for a particular application. On the other hand, people who are browsing the knowledge base to find solutions for

---

1. [http://supermarket.chef.io](http://supermarket.chef.io)
2. [http://hub.docker.com](http://hub.docker.com)
3. [http://netflix.github.io](http://netflix.github.io)
4. [http://openstack.org](http://openstack.org)
5. [http://cncf.io](http://cncf.io)
6. [http://kubernetes.io](http://kubernetes.io)
environment-specific requirements may refine these and feed them back into the knowledge base. This overlap between producers and consumers of the knowledge base significantly increases the possibility that the knowledge base will be kept up to date. If producers did not have this immediate advantage, it would be much harder to motivate them to contribute to the knowledge base. This would include the risk that the knowledge base would eventually fall into disuse because it no longer provided any added value.

3.3 Implementing Gather’n’Deliver

The Gather’n’Deliver method, which was previously discussed on an abstract and generic level, can be implemented in different ways. To render the various aspects of the method more substantial, this section discusses the technical possibilities concerning the implementation of both the gather and deliver phases. Moreover, the aspects of the method this dissertation focuses on are indicated by referencing corresponding chapters. Starting with the gather phase, step A covers the discovery of solutions. This can be achieved by a manual process conducted by experts or by automated crawling techniques [WAL15a; WBKL16]. These two options are not intended to be mutually exclusive, but can be integrated, e.g., by automatically discovering solutions with machine-readable metadata attached and manually deriving solutions from human-readable technical documentation. Such an integrated approach is presented as part of this dissertation in Chapter 6. Next, step B deals with capturing previously discovered solutions inside the knowledge base. A comprehensive and domain-specific metamodel must be established and incorporated into the knowledge base. While Chapter 4 presents the foundation of such a metamodel, Chapter 6 complements the previously discussed solutions discovery approach with a corresponding capture approach. Furthermore, the integration with the knowledge base is discussed. Step C of the gather phase aims to refine the solutions’ metadata. A key aspect of this metadata refinement are the links between solutions. Links can be immediate, so a solution directly points to another solution
as dependency, recommendation, or conflict. Alternatively, links can also be indirect by referring to a group or class of solutions. Therefore, several classification dimensions are presented in Chapter 5. Since certain classes are associated with particular solutions in order to categorize them, links can point to such classes instead of specific solutions.

Continuing with the deliver phase, step 1 is concerned with designing and implementing the actual application. This is effected using existing modeling and development tools such as UML designers and integrated development environments (IDEs). Step 2 then considers preparing the continuous delivery of an application by designing an appropriate delivery pipeline. In addition to the different stages specified by the pipeline design, several application environments are designed and associated with these stages such as a build environment, test environment, etc. Higher-level application topology models [OAS13b, ASLW14] that are not yet deployable and corresponding modeling tooling [KBBL13] can be used to design such application environments. This is the bridge to step 3, which is concerned with defining environment-specific requirements such as the need for certain middleware components or build tools in specific application environments. Therefore, Chapter 4 introduces a comprehensive metamodel, including the concept of application environment requirements. Based on this metamodel, Chapter 5 presents various classification dimensions, which are used to express diverse requirements. With this approach, step 4 is covered, too. Environment-specific requirements can now be resolved by solutions stored in the knowledge base. These solutions respect the same classification dimensions, so there is a common foundation for matchmaking in the resolution process.

Step 5 is optional, but becomes relevant in the event that not all environment-specific requirements can be immediately satisfied by existing solutions stored in the knowledge base. If the sources of similar solutions are available such as open-source artifacts, these can be retrieved and adapted to create solutions that eventually fit the specified requirements. In an alternate case of a solution being available, but not immediately executable (such as a human-readable technical tutorial), this solution can be refined by creating
an executable solution such as a deployment script. Technically, common
development tools and techniques can be used for adapting, refining, or
creating such solutions that may also be added to the knowledge base if
their reuse is intended. This is covered by steps B and C of the gather phase.

Once all solutions that satisfy the environment-specific requirements are
set, their orchestration is investigated in step 6. Since the set of solutions
can be highly diverse, due to their different origins and the diversity of
the knowledge base itself, their orchestration is a significant challenge. To
retain maintainability by preventing the orchestration layer from being
polluted with lots of lower-level technical logic, existing solutions can be
wrapped to provide their functionality through a particular kind of interface.
Chapter 7 presents approaches to streamline and ease the orchestration of
diverse solutions based on automated transformations and generating unified
interfaces. The successful orchestration of solutions eventually enables the
corresponding application environments to be instantiated and used as part
of the delivery pipeline. Finally, step 7 considers building the pipeline by
wiring the environments, therefore the pipeline can be operated as part
of step 8. Established tooling in the fields of continuous integration and
continuous delivery such as Go CD\footnote{http://www.go.cd} GitLab CI\footnote{http://about.gitlab.com/gitlab-ci} or Snap CI\footnote{http://snap-ci.com} are utilized for
this purpose. Alternatively, an overarching plan can implement the wiring
of the pipeline, for example, based on workflow technology or orchestration
scripts.

3.4 Chapter Summary and Discussion

This chapter presented the Gather'n'Deliver method, consisting of two
phases that are connected through a knowledge base: (i) gathering solutions
as input for the knowledge base and (ii) utilizing this knowledge to build
pipelines for continuously delivering software applications. Therefore, the
method is the initial step toward the major goals of this work: capturing
and finding proper solutions, as well as simplifying their usage and orchestration to build application environments as part of continuous delivery pipelines. Moreover, it discussed how various kinds of collaboration among roles and organizations are enabled. This is key because various experts need to collaborate efficiently across traditional borders (e.g., development and operations) in order to implement continuous delivery processes successfully. Since the Gather’n’Deliver method is intentionally generic enough to be implemented in various ways, several implementation possibilities and directions were outlined to clarify how the method could be technically realized. In this context, it was also described on which parts of the method this dissertation focuses. These parts are discussed in detail in the following chapters, including the application environment metamodel (Chapter 4), how to build application environment topologies (Chapter 5), the collaborative and automated gathering of solutions (Chapter 6), and finally how to streamline the orchestration of diverse solutions (Chapter 7).
As explained previously in the context of the Gather’n’Deliver method (Chapter 3), diverse application environments such as build, test, and production environments are the major building blocks of continuous delivery pipelines. Therefore, this chapter introduces a comprehensive application environment metamodel, specifically considering the implementation of continuous delivery. The metamodel aims to provide the conceptual foundation for dealing with application environment requirements and topologies.

4.1 Core Metamodel

Figure 4.1 provides an overview of the core metamodel, depicted in UML class diagram notation [OMG11b]. A specific application can be delivered through one or multiple pipelines. For example, in case of a single-instance but potentially large-scale SaaS application, implementing a single pipeline would make sense. However, if an application is delivered in multiple variants, e.g., to different customers using their dedicated instances, several
pipelines may be maintained. The metamodel supports various application architectures such as microservices or monoliths [New15]. Further application architectures that follow very different delivery and deployment procedures such as embedded systems are not immediately covered by this metamodel. A key assumption made by the metamodel is that an application must represent an independently deployable unit. Consequently, an application in the context of this metamodel does not have to be a completely self-contained software system, but can be an independently deployable building block (e.g., a microservice) of a larger system. Independent of the kind of application, each pipeline consists of specific stages such as build, unit test, performance test, staging, etc. [HF10]. Every new version of the application, e.g., providing a new feature or a bug fix must pass all these stages.

Application environment requirements (AERs) can be specified on two levels. (i) An application can define AERs that must be satisfied independent of a particular environment. As an example, if a particular application relies on
a middleware component such as a specific version of a runtime environment (e.g., Java version 7), this is relevant for all application environments. (ii) Furthermore, there are stage-specific AERs that are complementing the application-specific ones. For example, a Java-based application may require Maven\footnote{http://maven.apache.org} to resolve Java library dependencies in the build stage or JUnit\footnote{http://junit.org} to run its unit tests. Figure 4.2 shows sample requirements to position AERs as non-functional requirements regarding different kinds of application requirements. Functional requirements cover the specific features of an application such as a shopping cart functionality and the Web-based user interface (UI) of, e.g., a Web shop application. Non-functional requirements complement those by addressing additional aspects that are not directly related to an application’s functionality, but consider various technical and non-technical constraints to successfully operate the application. Beside AERs, non-functional requirements consider, for example, user experience, performance, and compliance aspects. Technically, AERs are partly derived from other existing non-functional requirements. For example, a compliance requirement prescribing that all data of an application must be hosted in
Europe automatically excludes all Cloud provider solutions that are hosted outside Europe.

*Application environment topology skeletons (AET skeletons)* are represented by application topology models as described in Section 2.3. Topology models specify the structure of specific application environments by resolving relevant AERs regarding required infrastructure, middleware, and tooling. These AET skeletons consist of a set of solutions to satisfy given AERs, but they are typically not immediately deployable. This is because of various reasons: beside wiring and integrating the set of solutions, they need to be properly configured. In addition, custom glue code may have to be developed for integration purposes. Therefore, a *deployable AET* technically refines an AET skeleton to eventually enable its automated deployment. Having deployable AETs enables *plans* to run based on them in corresponding environments. These plans implement the stage-specific logic, which runs as part of the pipeline. A deployable AET can be used for multiple stages. For instance, a test plan and build plan of an application can run in the same environment if their AERs are similar. This reduces the number of dedicated environments that have to be established, thereby simplifying the pipeline. Plans and AETs are solutions on their own, so they may be reusable in the sense of recursive aggregation. As an example, a certain middleware or application component may be reused in multiple application stacks. Therefore, its build plan and build topology may be utilized as solutions to resolve AERs of application environments, which rely on the particular component.

The application environment metamodel is clearly designed for continuous delivery scenarios. However, a simplified variant of it (excluding the entities *pipeline* and *stage*) also fits scenarios without a delivery pipeline. For example, legacy systems may only run in two environments: a development environment to maintain and test the application, as well as a production environment. Although these environments are not wired using a pipeline, AERs can be defined for such a manually deployed application. The AERs can then be systematically resolved to identify and establish alternative AETs. This could be useful to evaluate different deployment options, for example, to migrate such an application or parts of it to a Cloud infrastructure. More-
over, plans may also exist and run beyond the scope of a particular stage in a pipeline. As an example, application management plans [BBKL13] may be utilized in the production environment to scale an instance of the deployed application instance.

### 4.2 Instance of Metamodel

An instance of the previously introduced application environment metamodel is presented in the following to provide a specific example for how the metamodel is conceptually used. Therefore, Figure 4.3 outlines an example based on a continuous delivery pipeline for a Web shop application [WBKL16]. In this scenario the pipeline includes a build stage, unit test stage, and further stages. Each stage specifies concrete application environment requirements (AERs). These AERs require corresponding solutions that are combined to compose an application environment topology skeleton (AET skeleton).

![Diagram](image)

**Figure 4.3: Example instance of application environment metamodel**
Different kinds of solutions such as deployment scripts, configuration definitions, container images, and Cloud provider APIs serve as building blocks of AETs. Figure 4.3 outlines an AET skeleton that is connected to the unit test stage by resolving its specific AERs. A corresponding deployable AET for the unit test stage technically refines the AET skeleton to provide a deployable topology model. The unit test plan implements specific logic to run application-specific unit tests using the deployable AET. Continuous delivery pipelines for specific applications such as the Web shop application may consist of highly individual stages and application environments. This is due to diverse aspects such as application design, security requirements, and organizational constraints. However, a set of common stages and application environments can be identified across different pipelines [HF10; HM11; Che15a]:

Development:

The development environment is typically the starting point of a continuous delivery pipeline. It runs on the developer’s machine and thus needs to be as ‘lightweight’ as possible. Container virtualization is often utilized for this purpose. Beside the required middleware and application components, additional developer tooling such as code editors and code repository clients are part of the environment. Developers commit changes and additions to a code repository to trigger a run of the pipeline.

Unit test:

Unit tests implement a verification activity for individual application components and even more fine-grained units such as classes or modules. This verification aims to verify whether the application works correctly. Beside the required middleware and application components, test runners are typically part of a unit test environment. These runners do not only execute the unit tests but can also generate test reports. A mock database may be part of the unit test environment to replace the production database of the application.
Integration test:

The unit test stage is usually complemented by an integration test stage to verify that the involved application components are interacting and working together properly. Integration tests typically focus on the interfaces between components. Beside the test runners, as well as required middleware and application components, a simplified deployment of the application is performed in this environment. This includes starting and wiring the individual components, as well as connecting them to a more realistic database back-end, which could be, for example, a (potentially anonymized) copy of the production database.

Build:

After unit and integration tests verified that the individual application components are working correctly, the application can be built to be deployed afterward in more production-like environments. Depending on the utilized packaging format this can be, for example, a Java JAR file, a virtual machine image, or a Docker container. Usually, a build environment does not need to run the affected application components (in contrast to test environments). Therefore, the required build tooling is usually the main component needed to carry out the build activities. Additionally, access to an artifact repository may be required to store the built artifacts for reuse during deployment.

Staging:

Moving further toward production, the application environments get more production-like to build confidence that the application runs properly in the actual production environment subsequently. The goal is to systematically achieve production readiness stage by stage. The staging environment, also called pre-production environment, is the foundation for more sophisticated tests such as performance regression tests that obviously must be performed in a production-like environment to be meaningful. Therefore, the previously built application components are
used to create a complete instance of the application topology.

**Performance test:**

A popular way to test the performance of an application is by doing load testing. Its basic idea is to put the application under heavy load and monitor its performance, for example, by measuring latency or response times. Load can either be generated based on synthetic or real usage profiles for a particular application. This stage may also cover performance regression testing, i.e., comparing the currently measured performance with the recorded performance test results of previous versions of the application. Technically, either a separate performance testing environment (e.g., clone of staging environment) could be established or the staging environment itself is used.

**Acceptance test:**

Acceptance testing at its core is a validation activity to find out whether the new version of an application actually addresses the users’ needs. This can be realized, for example, by exposing the new version to a selected or random subset of the application’s users. Both automated and manual feedback capabilities can be implemented to collect and evaluate acceptance test results. Technically, there could be a separate acceptance testing environment (e.g., clone of staging environment) or the staging environment itself is utilized for this purpose.

**Manual test:**

In addition to the previously outlined kinds of tests, manual testing can be performed to verify and validate further aspects of the application. These aspects may include the evaluation of user experience, as well as user interface design of the application, which are hard to test in an automated manner. Such manual tests can be performed in the staging environment.
Production:

Finally, the production environment is a dedicated application environment to serve the application to its users. Typically, this environment is the most complex one in terms of application deployment and management because sophisticated mechanisms are in place to ensure high availability, automatic scaling, disaster recovery, and other relevant properties to reduce downtime and avoid outages. Further ‘smoke tests’ and health checks are executed in some production environments to continuously verify that the application is running properly.

Each stage could potentially be based on a separate application environment. However, different environments may share the same application topology if their requirements are very similar or even identical. This may be true, for example, for an acceptance test environment and a performance test environment because for both the application runs in a production-like setup. Another prominent example are popular continuous delivery services such as Travis CI\(^1\) which run both build activities and unit tests in the same environment for efficiency purposes and to make the pipeline faster.

4.3 Solutions as Building Blocks of Application Environments

Various kinds of solutions provide the building blocks of application environments as outlined by the metamodel depicted in Figure 4.1. To enable the expression of relations between different solutions and linking application environment requirements (AERs) with solutions, this metamodel is refined as shown in Figure 4.4, depicted in UML class diagram notation [OMG11b]. An additional abstract kind of entity is introduced, namely *referenceables*. AERs point to referenceables instead of always immediately requiring solutions. Two kinds of concrete entities represent referenceables: *solutions* and *capabilities*. Solutions provide capabilities and may require referenceables, i.e., other solutions or capabilities that are provided by solutions. As

\(^1\)http://travis-ci.org
an example, a deployment script (solution) to install a MySQL database may require an Ubuntu operating system, which is provided as capability by another solution such as a virtual machine image. Consequently, the deployment script does not immediately point to a particular solution, but to a capability, which is provided by different solutions. In addition, solutions may recommend or conflict with certain referenceables. Such links are used to refer to other solutions or capabilities, which are either incompatible or they are recommended to be used in conjunction with a given solution. Solutions are not necessarily atomic, but can be composed of other existing solutions. Decoupling concrete solutions from requirements and capabilities is common practice [OAS13b] to interlink diverse solutions without an explosion of direct links between them. Moreover, this enables the systematic classification of solutions regarding their requirements and capabilities.

An example instance of the refined part of the conceptual metamodel is depicted in Figure 4.5. It outlines several solutions and provides examples for mechanisms to express their relations. A specific solution such as the MySQL Juju charm can directly point to another solution, for example, to specify a dependency on the Ceph Juju charm in this case. Alternatively, solutions can refer to capabilities in order to express specific requirements. For instance, the Apache Chef cookbook requires any solution that provides
Figure 4.5: Example instance of the refined part of the metamodel

an Ubuntu Linux 14.04 implementation. In addition, solutions such as the Google App Engine PHP API express which functionality they provide by referring to corresponding capabilities. Requirement links that are connected to specific solutions or capabilities can also originate in AERs instead of solutions. Consequently, AERs are resolved by retrieving the specific solution they refer to, or by finding a solution that satisfies the referred capability. Beside requirements and capabilities, recommendations and conflicts can be expressed by corresponding links. The MySQL Juju charm solution may, for example, point to the PHP version 5 capability in order recommend its usage in combination with an arbitrary solution that provides an implementation of PHP version 5.

In order to formally refine the metamodel, capabilities are represented as hierarchically structured labels as described by the following Definition 4.1. Consequently, links such as provides, requires, recommends, and conflicts that do not immediately target solutions are specified by labels.
Definition 4.1 (Label)
Let $\mathcal{L}$ be the set of all labels; let $n$ be a label name and $l_{\text{parent}} \in \mathcal{L}$ an existing label. A label $l \in \mathcal{L}$ is defined by a pair:

$$ l := (n, l_{\text{parent}}), l \neq l_{\text{parent}} $$

Labels are hierarchically structured with $l_{\text{parent}}$ being the parent label of $l$. This is specified by the label tree $\mathcal{T} := (\mathcal{L}, \mathcal{E})$ with $\mathcal{L}$ being the vertex set and $\mathcal{E}$ being the edge set, $(l_{\text{parent}}, l) \in \mathcal{E}$. Label $l_{\text{root}} \in \mathcal{L}$ is the root node of $\mathcal{T}$. ■

A label’s path representation is used informally in this work to describe a label with its complete context including all its parents. More specifically, the slash symbol is used to separate label names in a path. This is similar to the common practice of using the slash symbols as separator for the different parts of URLs and filesystem paths. The following list provides examples for hierarchically structured labels and their informal path representations:

- Label definition: $l_{\text{mw}} := (\text{Middleware}, l_{\text{root}})$
  Path representation: ‘Middleware’

- Label definition: $l_{\text{db}} := (\text{Database}, l_{\text{mw}})$
  Path representation: ‘Middleware/Database’

- Label definition: $l_{\text{rel}} := (\text{Relational}, l_{\text{db}})$
  Path representation: ‘Middleware/Database/Relational’

- Label definition: $l_{\text{my}} := (\text{MySQL}, l_{\text{rel}})$
  Path representation: ‘Middleware/Database/Relational/MySQL’

The informal path representation of labels improve their readability. However, dealing with labels in algorithms and mathematical expressions requires a formal way of expressing the complete context of a label including all its parents. Therefore, Definition 4.2 introduces label sets:
Definition 4.2 (Map: Label Set)

Let $l \in \mathcal{L}$ be a label and $L \subseteq \mathcal{L}$ be a set of labels. The following map associates a label $l$ with its label set $L$:

$$\text{labelSet}(l) := L$$

$L$ contains $l$ (i.e., $l \in L$), the immediate parent label of $l$, and all implicit parent labels of $l$ according to the label tree $\mathcal{T}$:

$$\forall l' \in L : l' = l \lor l' = l_{\text{root}} \lor (l_{\text{root}} \rightarrow^{*} l' \rightarrow^{*} l) \in \mathcal{T}$$

Such label sets are used to match requirements and capabilities. This is achieved by checking whether a requirement label $l_1$ is included in $\text{labelSet}(l_2)$, the label set of a capability label $l_2$, more formally expressed as $l_1 \in \text{labelSet}(l_2)$. Consequently, label sets help to consider the hierarchical structure of labels: $l_1 \in \text{labelSet}(l_2)$ is also true if one of the parent labels of $l_2$ matches $l_1$ according to the label tree $\mathcal{T}$.

4.3.1 Representation of Solutions

Solutions are the major building blocks of application environments as explained previously. The following definitions describe how solutions are formally represented. Initially, Definition 4.3 introduces the concept of a solution boundary to specify requirements, conflicts, and recommendations of specific solutions. This enables the expression of links and dependencies between solutions.

Definition 4.3 (Solution Boundary)

Let $S$ be the set of all solutions $s \in S$. The boundary $B_s$ of a solution is defined as a set of boundary predicates $b \in B_s$. Each boundary predicate is a Boolean
function with an associated type \( t \), qualifier \( q \), and optional description \( d \):

\[
b_{d}^{t,q} : S \rightarrow \{\text{true}, \text{false}\},
\]

\( t \in \{\text{requires}, \text{requiresHost}, \text{requiresPeer}, \text{conflicts}, \text{recommends}\} \)

The qualifier \( q \) is a URI\(^1\) or label according to Definition 4.1. The optional description \( d \) is an arbitrary string of characters.

Furthermore, solutions have associated properties (Definition 4.4) to represent metadata and content such as configuration options and files:

**Definition 4.4 (Solution Properties)**

Let \( S \) be the set of all solutions \( s \in S \). \( \text{prop}_s \) is a function, which represents arbitrary properties and content such as metadata of a solution as name-value pairs. Let \( N_{\text{properties}} \) be the set of all property names; let \( V_{\text{properties}} \) be the set of all property values. The properties function \( \text{prop}_s \) is defined as follows:

\[
\text{prop}_s : N_{\text{properties}} \rightarrow V_{\text{properties}}
\]

The \( \text{prop}_s \) function maps a property name \( n \in N_{\text{properties}} \) to a specific property value \( v \in V_{\text{properties}} \).

Based on the concepts of solution boundaries and properties, Definition 4.5 provides the formal representation of a solution:

**Definition 4.5 (Solution)**

Let \( S \) be the set of all solutions; let \( u_s \) be a uniform resource identifier (URI); let \( L_s \) be a set of labels specifying the capabilities that are provided by the solution; let \( B_s \) be the solution’s boundary and \( \text{prop}_s \) the solution’s properties. A solution \( s \in S \) is defined by a quadruplet:

\[
s := (u_s, L_s, B_s, \text{prop}_s)
\]

The URI \( u_s \) uniquely identifies a solution \( s \).
The boundary predicates \( b^t.q \in B_s \) enable a solution \( s \) to specify different kinds of links that refer to other solutions or capabilities. This is expressed by the associated qualifier \( q \) as a URI or label according to \textbf{Definition 4.1}. These links were introduced as part of the refined metamodel depicted in \textbf{Figure 4.4} and \textbf{Figure 4.5}: \textit{requires}, \textit{conflicts}, and \textit{recommends}. Links of type \textit{provides} are specified as part of the set of labels \( L_s \) associated with a solution \( s \). Moreover, requirements can be specified on a more fine-grained level. Instead of using \( t = \text{requires} \), the more specific link types \textit{requiresHost} or \textit{requiresPeer} can be utilized to express host and peer dependencies. For example, a MySQL server solution may require an Ubuntu Linux solution as its host (i.e., ‘MySQL \textit{requiresHost} Ubuntu’). An application component may connect to a database back-end that is running as a peer, potentially hosted on a different infrastructure (i.e., ‘application \textit{requiresPeer} database’). This kind of distinction between different kinds of dependencies is inspired by existing works: in TOSCA-based application topologies, \textit{requiresHost} and \textit{requiresPeer} links are typically denoted as ‘hosted on’ and ‘connects to’ relationships [OAS13a]. Another example is the Engage deployment system [FME12] that distinguishes between ‘inside dependencies’ and ‘peer dependencies’, corresponding to \textit{requiresHost} and \textit{requiresPeer}.

Solutions can be of different kinds and positioned on different levels such as Cloud provider APIs, virtual server images, deployment scripts, as well as human-readable technical tutorials and documentation. Some of them such as human-readable documents must be refined, for example, by incorporating the described steps into a deployment script to actually utilize the solution as building block of an application environment. The following example outlines how the Docker container \texttt{mysql}\(^1\) could be represented as a solution. Therefore, two labels are defined: \( l_{\text{mysql}} = (\text{MySQL}, l_{\text{root}}) \) and \( l_{\text{docker}} = (\text{DockerEngine}, l_{\text{root}}) \). The solution itself is defined as follows:

\[
\text{s}_{\text{mysqlDocker}} = (\text{http://hub.docker.com/...}, \{l_{\text{mysql}}\}, B_{\text{mysqlDocker}}, \text{prop}_{\text{mysqlDocker}})
\]

Its MySQL capability is expressed using the \( l_{\text{mysql}} \) label. Furthermore the

\(^1\text{http://hub.docker.com/~/mysql}\)
boundary predicate $b_{\text{requiresHost}, l_{\text{docker}}}^{v1.12+} \in B_{\text{mysqlDocker}}$ specifies a requiresHost dependency to express that the container needs a solution providing a Docker engine version 1.12 or better:

$$b_{\text{requiresHost}, l_{\text{docker}}}^{v1.12+}(s) := \begin{cases} 
\text{true} & \text{if } \text{prop}_s(\text{version}) \geq 1.12, \\
\text{false} & \text{otherwise}.
\end{cases}$$

Moreover, $\text{prop}_{\text{mysqlDocker}}$ specifies the properties associated with the previously defined solution:

$$\text{prop}_{\text{mysqlDocker}}(n) := \begin{cases} 
5 & \text{if } n = \text{version}, \\
\text{undefined} & \text{otherwise}.
\end{cases}$$

In this simplified example, a single property is specified only: ‘version = 5’.

### 4.3.2 Representation of Application Environment Requirements

Application environment requirements (AERs) are formally represented using predicate logic. More specifically, AER predicates are used for this purpose as described by [Definition 4.6](#):

**Definition 4.6 (AER Predicate)**

Let $S$ be the set of all solutions; let $\mathcal{R}$ be the set of all AER predicates. An AER predicate $r \in \mathcal{R}$ is a Boolean function with an associated type $t$, qualifier $q$, and optional description $d$:

$$r_{t,q}^d : S \rightarrow \{\text{true, false}\},$$

$$t \in \{\text{inclusive, selective}\}$$

The qualifier $q$ is a URI or label according to [Definition 4.1](#). The special case $q = \ast$ represents a wildcard qualifier to match any solution. The optional description $d$ is an arbitrary string of characters.

AERs can be of two different types: inclusive AERs are utilized to be evaluated as ‘queries’ against a knowledge base to retrieve a set of solutions
that fit; selective AERs are applied to already retrieved solutions to further filter and refine specific solution sets. The qualifier \( q \) is used for inclusive AER predicates to retrieve a set of solutions, which is then filtered by applying the predicate to each solution. In case of selective AER predicates, the qualifier is considered to evaluate whether the given predicate is applicable to a given solution. If, for example, a certain solution does not provide the capability specified by the predicate’s qualifier label, the predicate is not meant to be applied to this particular solution. In case of \( q = * \), the predicate can be applied to any solution. However, using \( q = * \) as qualifier for an inclusive AER predicate usually does not make sense because it would represent a query that retrieves all solutions from the knowledge base.

Since AER predicates and boundary predicates (Definition 4.3) both aim to specify links to certain solutions or capabilities, their formal representation is aligned. In the course of this work both kinds of predicates are therefore treated in combination when it makes sense. The union of AER predicates and boundary predicates is defined for this purpose:

**Definition 4.7 (Union of AER Predicates and Boundary Predicates)**

Let \( B \) be the set of all boundary predicates; let \( R \) be the set of all AER predicates. \( P \) is the union of all boundary predicates and all AER predicates: \( P := B \cup R \)

Since the qualifier \( q \) of a predicate determines to which solutions they are meant to be applied, **Definition 4.8** formally specifies a function to check whether a given predicate is applicable to a specific solution:

**Definition 4.8 (Map: Predicate Applicable)**

Let \( s = (u_s, L_s, B_s, prop_s) \in S \) be a solution (Definition 4.5) and \( p^{t, q} \notin B_s \subseteq P \) be an AER predicate or a boundary predicate. The following map is defined as a Boolean function:

\[
\text{predicateApplicable} : S \times P \rightarrow \{\text{true, false}\},
\]

\[
\text{predicateApplicable}(s, p^{t, q}) = \text{true} :\iff (q = * \lor (q = u_s) \lor (\exists l \in L_s : q \in \text{labelSet}(l))
\]
To make a concrete example for AER predicates and solutions, let \( l_{mysql} = (\text{MySQL}, l_{root}) \) be a label and \( s_{my} = (u_{s_{my}}, L_{s_{my}}, B_{s_{my}}, P_{s_{my}}) \) a solution with \( l_{mysql} \in L_{s_{my}} \). The following AER predicate

\[
\begin{cases}
\text{true} & \text{if } prop_s(\text{version}) \geq 5 \land prop_s(\text{scaling}) = \text{auto}, \\
\text{false} & \text{otherwise.}
\end{cases}
\]

can be defined to filter a set of solutions by returning only solutions that provide a MySQL server implementation version 5 or better and offer an auto-scaling mechanism. The subscript description ‘\( v5+, \text{autoscale} \)’ does not have any formal meaning, but aims to improve the readability of predicates by summarizing what it checks for. This is similar to the formally irrelevant subscript \( my \) of \( s_{my} \) that is used to denote that a MySQL solution is provided. Alternatively to such very specific AER predicates, more generic ones can be defined. The following AER predicate

\[
\begin{cases}
\text{true} & \text{if } prop_s(\text{version}) \geq v, \\
\text{false} & \text{otherwise.}
\end{cases}
\]

is an example for a configurable AER predicate that provides the \( v \) parameter as part of the predicate’s subscript (instead of a static description) to dynamically define the required MySQL version when applying the predicate. This approach helps to reduce the number of predicates that have to be defined by making them reusable. Moreover, the \textit{predicateApplicable} map can be applied as follows

\[
\text{predicateApplicable}(s_{my}, r_{v5+, \text{autoscale}}^{selective, l_{mysql}}) = \text{true}
\]

\[
\text{predicateApplicable}(s_{my}, r_{(5)}^{selective, l_{mysql}}) = \text{true}
\]

to check whether given AER predicates are applicable to specific solutions.
4.4 Chapter Summary and Discussion

Initially, this chapter introduced a comprehensive application environment metamodel in the context of continuous delivery pipelines. The metamodel is used throughout this work. Application environment requirements (AERs) and solutions were described as major building blocks by defining and using AERs to identify proper solutions. AERs have been positioned as one specific kind of non-functional requirements in the context of other types of application requirements. Formal representations of both AERs and solutions were described. AERs differ in what kind of requirements they express on which level. (i) As discussed in this chapter, AERs can be of two different types: inclusive AERs are utilized as ‘queries’ to be evaluated against a knowledge base to retrieve a set of solutions that fit; selective AERs are evaluated to already retrieved solutions to further filter and refine specific sets of solutions. (ii) Some AERs are specific to a certain application environment such as a build tooling requirement, which only needs to be resolved for establishing the build environment for a particular application. Other AERs are relevant for all application environments of a particular application such as a runtime environment that is inherently required to run the application in any environment. (iii) Independent of the environment dimension, AERs are scoped differently. Most AERs are typically application-specific, i.e., they define various requirements that are specific to the application depending on its components, utilized middleware, and further tooling. However, AERs can be organization-specific and may thus be relevant for several applications developed and operated by a particular organization. As an example, an organization may decide to use a single Cloud provider only because of trust or to simplify the landscape of infrastructure solutions. These can be cascading requirements, e.g., AERs that must be respected by the whole company, AERs that are relevant for a certain department or group only, and AERs that are completely application-specific.

These aspects help to systematically group AERs on different levels, so only the relevant ones are selected to identify solutions for a particular
application environment. For example, the previously defined AER predicate

\[ \text{selective, } \text{mysql} \]
\[ r^{v5+, \text{autoscale}} \]

can be categorized as production environment-specific AER because autoscaling is usually not relevant for build and test environments. As application-specific AER it is scoped to a specific application that uses a MySQL database. The type of AER is selective to filter an existing set of solutions such as a collection of various MySQL server implementations.

To link solutions and express requirements, capabilities, recommendations, and conflicts, the concept of labels was formally introduced. Moreover, it was described how AER predicates and boundary predicates associated with specific solutions are used in conjunction with labels to express links between solutions. A concrete instance of the metamodel and its refined variant was outlined to demonstrate how the metamodel is used conceptually. The following Chapter 5 builds upon these fundamentals and presents concepts to build application environment topologies.
In order to run the different stages of continuous delivery pipelines, potentially diverse application environments need to be implemented as key building blocks of the pipeline. This chapter presents concepts to build such application environments based on topology models as introduced in Section 2.3. Following the previously described application environment metamodel (Chapter 4), it is shown how application environment requirements can be systematically resolved based on labels to match such requirements with capabilities provided by solutions. These labels are hierarchically structured using label taxonomies, which are discussed in this chapter. Initially, the underlying classification dimensions for these taxonomies are presented in the following.

5.1 Classification Dimensions of Requirements and Solutions

Application environment requirements (AERs) must be satisfied by specific solutions that provide corresponding capabilities. To enable a systematic
matchmaking between requirements and solutions, labels are utilized as described in Chapter 4. These labels are classified according to multiple dimensions that are outlined in the following. More specifically, the discussed dimensions are solution types, solution bindings, modes of use, and usage styles. While these dimensions are technically relevant in the context of this work, further dimensions could be added such as monetary cost or performance considerations, which are not the focus of this work.

5.1.1 Solution Types and Solution Bindings

Each solution is of a certain type, meaning that it provides a corresponding implementation. For example, the Chef cookbook `mysql` represents a solution that is capable of installing and running a MySQL database server. Therefore, it is classified as MySQL solution. Moving up the hierarchy, MySQL is a relational database, which itself is a specific kind of middleware. Beside middleware there are further top-level solution types such as infrastructure (covering operating systems, networking solutions, etc.) and DevOpsware (covering deployment automation and further supplementary tooling).

Beside the previously described type dimension, there is the binding dimension: some solutions are bound to be used in a specific context only. For example, Google Cloud SQL SDK can be utilized as a solution to create fully managed MySQL database instances and run them ‘as a service’. However, the solution can only be used in conjunction with Google’s Cloud platform. Therefore, the solution is provider-bound to Google. While the solution itself may perfectly fit the technical needs of a certain application environment, the provider binding may not be acceptable. In this case, alternative solutions must be considered. Another solution binding can be a specific region or location, so the solution is location-bound. In case of the Google Cloud SQL SDK example, database instances can only be created in three specific Google data centers: Asia Pacific, European Union, and North America.

---

2. [http://cloud.google.com/sql/docs/cloud-sdk](http://cloud.google.com/sql/docs/cloud-sdk)
3. [http://cloud.google.com/sql/docs/instance-locations](http://cloud.google.com/sql/docs/instance-locations)
Consequently, if a MySQL database must be hosted in another specific region or country, alternative solutions have to be used.

5.1.2 Modes of Use and Usage Styles

As another classification dimension, different modes of use are followed when utilizing diverse solutions. For example, the previously described Chef cookbook `mysql` is implemented as an executable configuration definition, in contrast to the Google Cloud SQL SDK, which is a ‘ready-to-use’ service offering. As a result, their modes of use differ. A configuration definition such as a Chef cookbook requires an existing infrastructure, for example, a virtual server to be executed on. In addition, a Chef runtime environment is required to execute the Chef cookbook to eventually install and run a MySQL database instance. Therefore, the mode of use would be executable artifact or, more specifically, configuration definition. Going for a database-as-a-service offering such as Google Cloud SQL, it is considerably simpler to manage MySQL database instances directly through a command-line interface (CLI). Thus, the mode of use would be API or CLI, to be more specific. However, the inherent dependency on Google’s infrastructure and its APIs must be accepted.

Beside the technical mode of use, additional usage styles form another classification dimension when utilizing solutions. They cover a variety of both alternative and complementary paradigms how to use solutions. A selection of such usage styles were identified by systematically analyzing commonly used solutions:

**Virtualization level:**

Different kinds of solutions can be used on different virtualization levels. Each level implies a different degree of isolation and virtualization overhead. For example, a virtual machine (VM) runs on top of a hypervisor, which provides a hardware virtualization approach. While each VM owns a dedicated guest operating system, providing a high degree of isolation, containers share the operating system’s kernel (container virtualization).
Another option is to virtualize on the middleware level [SASL13], for example, by using a multi-tenant aware application server to host applications of multiple customers. Some solutions such as virtual machines can only be used on a specific virtualization level, others such as configuration definitions (running on physical machines, VMs, or containers) can be used on multiple levels.

**Instance-driven:**

Some solutions are instance-driven, i.e., users immediately manage and interact with instances such as virtual servers, runtime environments, application instances, and database instances. This is the case for many platform-as-a-service (PaaS) solutions such as Cloud Foundry[^1] and PaaS providers. Typically, command-line interfaces, GUIs, or APIs are utilized to interact with such a platform and manage instances on it. Standardization efforts such as CAMP (Cloud Application Management for Platforms) [OAS14] aim to streamline the interaction with such platforms.

**Template-driven:**

In contrast to the instance-driven approach described previously, other solutions are template-driven. Instead of immediately using APIs to manage instances, templates are created and then processed by a deployment engine to create corresponding instances. A template could be, for example, an application topology model as described in Section 2.3 or a deployment plan. TOSCA (Topology and Orchestration Specification for Cloud Applications) [OAS13b] is a standard to specify such templates in a portable manner. A CloudFormation template[^2] is a provider-specific example for a topology model to orchestrate a set of Amazon’s Cloud services.

[^1]: [http://www.cloudfoundry.org](http://www.cloudfoundry.org)
Infrastructure-oriented:

Solutions are infrastructure-oriented in case they focus on managing and orchestrating infrastructure-level resources such as virtual servers, networks, firewalls, and block storage. APIs\(^1\) and toolkits\(^2\) are available to interact with infrastructure-as-a-service provider offerings and frameworks such as Amazon EC2 and OpenStack. These solutions provide fine-grained interfaces to manage infrastructure resources. The proper configuration and orchestration of these resources allow application instances to run based on the infrastructure.

Application-oriented:

In contrast to the previously described infrastructure-oriented approach, solutions are application-oriented in case they focus on managing application-level resources such as deploying, updating, and scaling application components. Container virtualization approaches and related cluster management frameworks such as Docker Compose\(^3\) and Kubernetes\(^4\) are prominent examples for orchestrating application-level containers. Each container hosts a separate application component. Cluster management frameworks provide a layer of abstraction, hiding the underlying infrastructure resources such as virtual machines and networks to focus on managing application resources that are, for example, packaged as containers.

Middleware-oriented:

Alternatively to the infrastructure-oriented and application-oriented approaches, solutions can be middleware-oriented [WASL14; WALS16]. This applies to solutions that manage resources on the level of middleware. For example, PaaS providers such as Heroku\(^5\) offer a broad variety

\(^{1}\) [http://developer.openstack.org/api-ref.html](http://developer.openstack.org/api-ref.html)
\(^{2}\) [http://aws.amazon.com/sdk-for-java](http://aws.amazon.com/sdk-for-java)
\(^{3}\) [http://docs.docker.com/compose](http://docs.docker.com/compose)
\(^{4}\) [http://kubernetes.io](http://kubernetes.io)
\(^{5}\) [http://elements.heroku.com/addons](http://elements.heroku.com/addons)
of middleware-level services for data processing, authentication, user management, search, and others. These middleware services can be configured and orchestrated to provide the foundation for running a specific application.

Node-centric:

Some solutions are node-centric [WBL14c; WBKL16]. For example, a Chef cookbook is a solution that provides a configuration definition to be executed on a single node such as a virtual server. It cannot be used to properly orchestrate multiple configuration definitions that are distributed across several nodes, including cross-node relations. Virtual machine images or individual containers are further examples for node-centric solutions. Further (environment-centric) solutions are required to manage the orchestration of node-centric solutions.

Environment-centric:

Environment-centric solutions [WBL14c; WBKL16] are required to manage environments consisting of multiple nodes. Juju charms\(^1\) are an example for environment-centric artifacts to manage a set of nodes as a coherent environment. Furthermore, solutions based on topology models such as CloudFormation templates or TOSCA topology templates can be utilized to manage the orchestration of multiple nodes and other resources. Therefore, these are classified as environment-centric solutions, too.

Convergence-based:

Several configuration management solutions are based on convergence [WBL14a; WBL15b]. For example, a Chef cookbook must provide an idempotent configuration definition that is executed repeatedly until the desired configuration state is reached. Repeated runs are required if an error occurs during an execution or the desired state is not reached after

\(^1\)http://jujucharms.com
the initial execution. With this approach the target resource such as a virtual server converges toward the desired state.

**Compensation-based:**

In contrast to the previously described convergence-based approach, other solutions rely on compensation mechanisms [WBL14a; WBL15b]. For example, a Docker container image is built by executing the build steps specified in a Dockerfile\(^1\). Each successfully finished build step produces a temporary intermediate container image. In case a particular build step fails, the engine goes back to the last successfully built intermediate container image. From there, the build process can be restarted without repeatedly running all build steps again.

The various classification dimensions discussed previously are the foundation of the label taxonomies presented in the following Section 5.2. The hierarchical label structure provided by the taxonomies is used to enable the matchmaking between requirements (AERs) and solutions’ capabilities as discussed in Chapter 4.

### 5.2 Label Taxonomies

Several label taxonomies are presented in this section to express both requirements and capabilities. Following the solution type classification discussed in Section 5.1.1, Figure 5.1 outlines an extract of the corresponding label taxonomy\(^2\). The taxonomy respects common categorization approaches of middleware and infrastructure entities used by established providers and vendors such as Heroku\(^3\) and New Relic\(^4\). Moreover, existing classifications of NoSQL databases proposed by Sadalage and Fowler [SF12] are incorporated into the taxonomy. The DevOpsware entity covers all supplementary

---

1. [http://docs.docker.com/engine/reference/builder](http://docs.docker.com/engine/reference/builder)
2. [http://github.com/devopsbase/devopsbase](http://github.com/devopsbase/devopsbase)
solutions that are relevant for both developing and operating applications. These include build and test tools, deployment and orchestration solutions, as well as monitoring and logging frameworks. In contrast to middleware and infrastructure solutions, these do not immediately support application functionality, but they are complementary required to develop and manage an application. Using the path representation of labels (Chapter 4), the following examples are valid labels according to the type taxonomy:

- **Type/Middleware/Database/Relational/MySQL**
- **Type/Infrastructure/Operating System/Ubuntu**
- **Type/DevOpsware/Deployment/Chef**

As discussed in Section 5.1.1 some solutions are bound to be used in a certain context only. Figure 5.2 outlines an extract of the label taxonomy to cover such solution bindings. Two common kinds of bindings are solutions
that are provider-bound or region-bound. Provider bindings can be refined to cover, for example, their individual service offerings such as Amazon EC2\(^1\) and Google App Engine\(^2\). Similarly, region bindings can be refined to cover countries or even individual cities. Using the path representation of labels, the following examples are valid labels according to the binding taxonomy:

- `Binding/Provider/Amazon`
- `Binding/Provider/Google/App Engine`
- `Binding/Region/Asia Pacific`

The actual usage of individual solutions can be quite different. For example, utilizing an API served by a Cloud provider is completely different from an executable artifact such as a deployment script running on a virtual server. These differences are covered by the modes of use classification discussed in Section 5.1.2. An extract of a corresponding label taxonomy is outlined in Figure 5.3. Different kinds of APIs exist such as endpoints and toolkits (e.g., SDKs or libraries), implying different modes of use. Endpoint APIs can either

\(^1\) [http://aws.amazon.com/ec2](http://aws.amazon.com/ec2)
\(^2\) [http://cloud.google.com/appengine](http://cloud.google.com/appengine)
be provider-hosted such as Amazon’s EC2 API[^1] or self-hosted such as an OpenStack instance with its corresponding API[^2]. Command-line interfaces (CLIs) and graphical user interfaces (GUIs) are further modes of use; examples include Amazon’s AWS CLI[^3] and management console[^4]. Another mode is an executable. As outlined by the taxonomy, different kinds of executables imply refined modes of use. Scripts and configuration definitions such as Chef cookbooks are used differently than container and virtual machine images such as Docker containers and Vagrant boxes[^5]. Further kinds of executables include bundles such as TOSCA Cloud Service Archives (CSARs) [^OAS13b] and workflow models such as BPEL plans [^OAS07]. Yet another mode of use is a guide such as patterns [^Ale77], [^FBFL15] or tutorials. Solutions that are classified under this mode typically provide human-readable resources that must be refined before using them for deployment automation purposes. For example, a technically fine-grained tutorial explaining how to run WordPress

[^1]: http://docs.aws.amazon.com/AWSEC2/latest/APIReference
[^2]: http://developer.openstack.org/api-ref.html
[^3]: http://aws.amazon.com/cli
[^4]: http://aws.amazon.com/console
[^5]: http://www.vagrantup.com/docs/boxes.html
on Amazon EC2\textsuperscript{1} can be codified as a deployment script and then utilized to automate its deployment. Using the path representation of labels, the following examples are valid labels according to the mode of use taxonomy:

- **Mode/API/Endpoint/Provider-hosted**
- **Mode/Executable/Script**
- **Mode/Guide/Tutorial**

Beside the modes of use, solutions can be classified according to different usage styles as discussed in Section 5.1.2. An extract of the corresponding taxonomy is outlined in Figure 5.4, covering the previously discussed usage styles. Using the path representation of labels, the following examples are valid labels according to the usage style taxonomy:

- **Style/Virtualization Level/Hardware**
- **Style/Template-driven**
- **Style/Infrastructure-oriented**

\textsuperscript{1}http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/hosting-wordpress.html
All of the presented taxonomies are meant to be used to label individual solutions with capabilities and finally drive the resolution of application environment requirements as described in the following Section 5.3.

5.3 Resolution of Application Environment Requirements

**Algorithm 5.1** findSolutionSets($P_{DNF}^A$)

1: $P_{DNF}^A$ \(\triangleright\) input: DNF set representation of AER predicates
2: $S_{combined} \leftarrow \{\}$ \(\triangleright\) output: set of combined solution sets
3: for all $A \in P_{DNF}^A$ do \(\triangleright\) iterate over all alternatives $A$ in $P_{DNF}^A$
4: $S_{grouped} \leftarrow \{\}$ \(\triangleright\) set of grouped solutions
5: for all $r^{t,q} \in A$ do \(\triangleright\) iterate over all inclusive AERs $r$ in $A$
6: if $t =$ inclusive and $r$ is non-negated predicate then
7: if $q$ is a URI then
8: $G \leftarrow \{KB.getSolutionByURI(q)\}$
9: else if $q$ is a label then
10: $G \leftarrow KB.getSolutionSetByLabel(q)$
11: end if
12: if $G \neq \emptyset$ then
13: $S_{grouped} \leftarrow S_{grouped} \cup \{G\}$
14: end if
15: end if
16: end for
17: $S_{filtered} \leftarrow \{\}$ \(\triangleright\) set of filtered grouped solutions
18: for all $G \in S_{grouped}$ do
19: $S_{filtered} \leftarrow S_{filtered} \cup \{filteredSolutionSet(G,A)\}$
20: end for
21: $S_{combined} \leftarrow S_{combined} \cup combinedSolutionSets(S_{filtered},A)$
22: end for
23: return $S_{combined}$

The process of systematically resolving application environment requirements (AERs) is enabled by the previously presented label taxonomies (Section 5.2) and the application environment metamodel (Chapter 4). For this purpose, the hierarchical structure of the label taxonomies is explicitly considered. The resolution consists of two major parts discussed in the
following, namely resolving immediate AERs (Section 5.3.1) and resolving solutions’ requirements (Section 5.3.2) because a solution itself may specify additional transitive requirements that must be satisfied. Algorithms are presented to properly explain the mechanisms required for the resolution of requirements. However, these algorithms are not meant to be executed in a fully automated manner, but are intended to be used as semi-automated procedures to support the systematic decision making process of selecting appropriate solutions.

5.3.1 Resolve Immediate Requirements

Application environment requirements are represented by AER predicates as described in Chapter 4. Algorithm 5.1 outlines how such AERs can be resolved by identifying and retrieving corresponding solution sets. The algorithm expects the set representation of AER predicates in disjunctive normal form (DNF) as input. Diverse algorithms and implementations exist for transforming arbitrary logical expressions to DNF. One example is the open-source implementation of the `to_dnf` function as part of SymPy’s Logic module\(^1\). In addition, Definition 5.1 describes how the set representation of a logical expression in DNF is created.

**Definition 5.1 (Set Representation of Disjunctive Normal Form)**

Let \( p_{i,j} \in \mathcal{P} \) be AER predicates or boundary predicates (Definition 4.7) with \( i, j, n, k \in \mathbb{N} \). The disjunctive normal form (DNF) of logical expressions consisting of predicates is defined as follows:

\[
(p_{1,1} \land \ldots \land p_{1,n_1}) \lor \ldots \lor (p_{k,1} \land \ldots \land p_{k,n_k})
\]

Such an expression can be represented as a set of predicate sets:

\[
P^{\text{DNF}} := \{\{p_{1,1}, \ldots, p_{1,n_1}\}, \ldots, \{p_{k,1}, \ldots, p_{k,n_k}\}\} = \{A_1, \ldots, A_k\}
\]

Each predicate set represents an alternative set \( A_i \) consisting of AERs. \( \blacksquare \)

\(^{1}\)http://docs.sympy.org/1.0/modules/logic.html
Based on the set representation of the DNF expression, the algorithm iterates over each alternative set $A_k$ of AER predicates. For each alternative $A_i$, each non-negated predicate $p_{i,j}$ of type inclusive is evaluated to retrieve potential solutions from the knowledge base as introduced in Chapter 3 in the context of the Gather’n’Deliver method. Therefore, the algorithm assumes two basic operations to be provided by the knowledge base (KB) to retrieve solutions. These are in particular:

- $\text{KB.getSolutionByURI(solutionURI)}$ returns a single solution, which is uniquely identified by its URI.

- $\text{KB.getSolutionSetByLabel(capabilityLabel)}$ returns a solution set consisting of solutions that provide the capability defined by the given label (Section 5.2).
How these operations are technically provided is an implementation detail and therefore conceptually irrelevant. For example, a database index or search engine may be used in the background to efficiently identify proper solutions. Figure 5.5 visually outlines how the algorithm works in conjunction with AERs and solutions. For retrieving solutions, given AER predicates of type inclusive are considered. Each inclusive predicate either defines a URI or a label to identify required solutions. In case of a URI, the \( \text{KB.getSolutionByURI} \) operation is used to retrieve the corresponding solution if it exists. If a label is given, the \( \text{KB.getSolutionSetByLabel} \) operation enables the retrieval of a potentially empty solution set, providing the capability defined by the label. The result of this retrieval step are solution sets. The amount of sets equals to the amount of given inclusive predicates. Each set represents a group of solutions, which satisfies one particular inclusive predicate. These sets are then filtered based on given AER predicates of type selective. The following Definition 5.2 defines the \( \text{filteredSolutionSet} \) map, which is used by the algorithm to filter solution sets based on selective predicates.

**Definition 5.2 (Map: Filtered Solution Set)**

Let \( S_1 \subseteq S \) and \( S_2 \subseteq S \) be solution sets; let \( \mathcal{P}(S) \) be the power set of all solution sets; let \( R \subseteq \mathcal{R} \) be a set of AER predicates \( r^{t,q} \in \mathcal{R} \) of type \( t \) with qualifier \( q \); let \( \mathcal{P}(\mathcal{R}) \) be the power set of all AER predicate sets. The input of the following map is a solution set and an AER predicate set; its result is a solution set:

\[
\text{filteredSolutionSet}: \mathcal{P}(S) \times \mathcal{P}(\mathcal{R}) \rightarrow \mathcal{P}(S),
\]

\( (S_1, R) \mapsto S_2 \)

The resulting solution set \( S_2 \) is:

\[
S_2 := \{ s \mid s \in S_1 \land (\forall r^{t,q} \in R: (\text{predicateApplicable}(s, r^{t,q}) = \text{true}) \implies (r^{t,q}(s) = \text{true})) \}
\]

The subset \( S_2 \subseteq S_1 \) omits solutions of the originally given solution set \( S_1 \), which are filtered out by applying all \( r^{t,q} \in R \).

5.3 | Resolution of Application Environment Requirements 97
The sets of grouped solutions are then combined to create alternative solution sets. Ideally, each solution set contains a solution that resolves at least one of the given AERs. However, if the knowledge base does not provide a solution to satisfy a particular AER, resulting solution sets may be incomplete in the sense that some requirements are still unresolved. Consequently, a solution set forms a potentially incomplete set of building blocks of an application environment that is specified by the given AERs. The **combinedSolutionSets** map, used by [Algorithm 5.1](#) for this purpose, is defined in the following **Definition 5.3**.

**Definition 5.3 (Map: Combined Solution Sets)**

Let \( S_i \subseteq S \) and \( S'_j \subseteq S \) be solution sets with \( i, j \in \mathbb{N} \); let \( \mathcal{P}(S) \) be the power set of all solution sets; let \( \{S_1, \ldots, S_m\} \subseteq \mathcal{P}(S) \) and \( \{S'_1, \ldots, S'_n\} \subseteq \mathcal{P}(S) \) be sets of solution sets with \( m, n \in \mathbb{N} \); let \( R \subseteq \mathcal{R} \) be a set of potentially unresolved AER predicates; let \( \mathcal{P}(\mathcal{R}) \) be the power set of all AER predicate sets. The input of the following map is a set of solution sets and an AER predicate set; its result is a set of pairs, each consisting of a combined solution set and the given AER predicate set:

\[
\text{combinedSolutionSets}: \mathcal{P}(\mathcal{P}(S)) \times \mathcal{P}(\mathcal{R}) \rightarrow \mathcal{P}(\mathcal{P}(S) \times \mathcal{P}(\mathcal{R})),
\]

\[
(S_1, \ldots, S_i, \ldots, S_m, R) \mapsto \{(S'_1, R), \ldots, (S'_j, R), \ldots, (S'_n, R)\}
\]

Each combined solution set contains one solution of each of the originally provided solution sets without conflicts:

\[
\forall (S'_j, R): \begin{align*}
S'_j & \neq \emptyset \\
\land (\forall S_i: |S_i \cap S'_j| = 1 \lor S_i = \emptyset) \\
\land (\forall S_i \forall s \in S_i \exists S'_j: s \in S'_j \lor S_i = \emptyset) \\
\land \text{conflictingSolutions}(S'_j) = \emptyset
\end{align*}
\]
Definition 5.3 uses the conflictingSolutions map to avoid conflicting solutions inside a particular solution set. Therefore, the following Definition 5.4 formally specifies the conflictingSolutions map.

**Definition 5.4 (Map: Conflicting Solutions)**

Let $S \subseteq S$ be a solution set consisting of solutions $s_i = (u_{s_i}, L_{s_i}, B_{s_i}, prop_{s_i})$ and $s_j = (u_{s_j}, L_{s_j}, B_{s_j}, prop_{s_j})$ with $s_i, s_j \in S$ and $i, j \in \mathbb{N}$; let $\mathcal{P}(S)$ be the power set of all solution sets. The input of the following map is a solution set $S$; its result is a potentially empty set of pairs $(s_i, s_j)$:

$$conflictingSolutions: \mathcal{P}(S) \rightarrow \mathcal{P}(S \times S),$$

$$S \mapsto \{\ldots, (s_i, s_j), \ldots\}$$

Each resulting pair $(s_i, s_j)$ consists of two conflicting solutions:

$$\forall (s_i, s_j):$$
$$\exists b_{i,q}^t \in B_{s_i}:$$

$$(t = conflicts \land predicateApplicable(s_j, b_{i,q}^t) = true) \implies (b_{i,q}^t(s_j) = true)$$
$$\lor \exists b_{j,q}^t \in B_{s_j}:$$

$$(t' = conflicts \land predicateApplicable(s_i, b_{j,q}^t) = true) \implies (b_{j,q}^t(s_i) = true)$$

Each combined solution set does not only consist of the solutions themselves, but also has the AER predicates attached, which drove the creation of the particular solution set. These requirements are reused later in the process when adding or refining solutions of a solution set. This helps to ensure that an updated solution set still complies with the originally defined AERs.

To explain more explicitly how the combinedSolutionSets is applied, a concrete example is described in the following. Let $prop_1, prop_2, prop_3$ be partial functions that map names to values to represent simplified properties of a solution as name-value pairs for demonstration purposes. More specifically,
prop₁ represents an empty set of properties:

\[
prop₁(n) := \text{undefined}
\]

prop₂ holds a single property, namely ‘version = 4’:

\[
prop₂(n) := \begin{cases} 
4 & \text{if } n = \text{version,} \\
\text{undefined} & \text{otherwise.}
\end{cases}
\]

Similarly, prop₃ also holds a single property, namely ‘version = 5’:

\[
prop₃(n) := \begin{cases} 
5 & \text{if } n = \text{version,} \\
\text{undefined} & \text{otherwise.}
\end{cases}
\]

Moreover, the following labels are defined:

- Label definition: \( l_{\text{type}} := (\text{Type}, l_{\text{root}}) \)
  Path representation: ‘Type’

- Label definition: \( l_{\text{mw}} := (\text{Middleware}, l_{\text{type}}) \)
  Path representation: ‘Type/Middleware’

- Label definition: \( l_{\text{db}} := (\text{Database}, l_{\text{mw}}) \)
  Path representation: ‘Type/Middleware/Database’

- Label definition: \( l_{\text{rel}} := (\text{Relational}, l_{\text{db}}) \)
  Path representation: ‘Type/Middleware/Database/Relational’

- Label definition: \( l_{\text{mysql}} := (\text{MySQL}, l_{\text{rel}}) \)
  Path representation: ‘Type/Middleware/Database/Relational/MySQL’

- Label definition: \( l_{\text{rt}} := (\text{Runtime}, l_{\text{mw}}) \)
  Path representation: ‘Type/Middleware/Runtime’

- Label definition: \( l_{\text{php}} := (\text{PHP}, l_{\text{rt}}) \)
  Path representation: ‘Type/Middleware/Runtime/PHP’

- Label definition: \( l_{\text{ws}} := (\text{WebServer}, l_{\text{mw}}) \)
  Path representation: ‘Type/Middleware/WebServer’
• Label definition: \( l_{\text{apache}} := (\text{ApacheHTTPD}, l_{\text{ws}}) \)
  
  Path representation: ‘Type/Middleware/WebServer/ApacheHTTPD’

Based on these labels, the following simple AER predicates are defined. The predicate \( r_1 \) is of type inclusive and requires a MySQL solution that provides a MySQL implementation version 5 or better:

\[
r_1 := r^{\text{inclusive}, l_{\text{mysql}}}_{\text{mysql}, v5+} (s) = \begin{cases} 
  \text{true} & \text{if } \text{prop}_s(\text{version}) \geq 5, \\
  \text{false} & \text{otherwise.}
\end{cases}
\]

Furthermore, the inclusive predicate \( r_2 \) requires an arbitrary solution that provides a PHP runtime environment:

\[
r_2 := r^{\text{inclusive}, l_{\text{php}}}_{\text{php}} (s) = \text{true}
\]

Similarly, the inclusive predicate \( r_3 \) requires an arbitrary solution that provides an Apache Web Server:

\[
r_3 := r^{\text{inclusive}, l_{\text{apache}}}_{\text{apache}} (s) = \text{true}
\]

In the context of this example, it is assumed that a specific application requires these AERs to be satisfied by solutions providing corresponding capabilities. Therefore, the AER predicates are bundled as a logical expression in DNF:

\[
r_{\text{all}} := r_1 \land r_2 \land r_3
\]

According to **Definition 5.1** this expression can be transformed to its set representation:

\[
P^{\text{DNF}}_{r_{\text{all}}} := \{\{r_1, r_2, r_3\} = \{A_1\}
\]

By following the previously discussed **Algorithm 5.1**, a set of solutions are retrieved from the knowledge base using the inclusive AERs specified by \( A_1 \in R \):
These solutions form a yet unfiltered solution group \( S_{\text{grouped}} \):

\[
S_{\text{grouped}} = \{ s_{\text{mysqlDocker}}, s_{\text{mysqlChef}}, \ldots \}
\]

Next, the filtered solution group \( S_{\text{filtered}} \) is derived by applying the given AER predicate set \( A_1 \):

\[
S_{\text{filtered}} = \text{filteredSolutionSet}(S_{\text{grouped}}, A_1)
\]

Compared to \( S_{\text{grouped}} \), the filtered group \( S_{\text{filtered}} \) does not contain solutions, which do not comply with the given AERs. In particular the solutions \( s_{\text{mysqlChef}} \) and \( s_{\text{mysqlInstallScript4}} \) are filtered out because their properties \( prop_1 \) and \( prop_2 \) do not comply with requirement \( r_1 \), which requires MySQL version 5 or better:

\[
S_{\text{grouped}} \setminus S_{\text{filtered}} = \{ s_{\text{mysqlChef}}, s_{\text{mysqlInstallScript4}} \}
\]

Finally, the combined solution set \( S_{\text{combined}} \) is created:

\[
S_{\text{combined}} = \text{combinedSolutionSets}(\{ S_{\text{filtered}} \}, A_1) =
\{(\{ s_{\text{mysqlDocker}}, s_{\text{apachePhpDocker}} \}, A_1), (\{ s_{\text{mysqlDocker}}, s_{\text{apachePhpAmazon}} \}, A_1), \ldots \}
\]

To continue with the process of building an application environment topology, a specific solution set \( S_{\text{selected}} \in S_{\text{combined}} \) is selected as the foundation for the
topology. The solutions \( s \in S_{selected} \) may have further requirements attached that must be resolved, too. Therefore, the following Section 5.3.2 describes how these requirements are resolved.

5.3.2 Resolve Solutions’ Requirements

**Algorithm 5.2** resolveDependencies(\( S_{selected}, R \))

1: \( S_{selected} \) ➞ input: selected solution set
2: \( R \) ➞ input: set of AER predicates
3: \( S_{combined} \leftarrow \{ \} \) ➞ output: combined set of solution sets
4: \( S_{grouped} \leftarrow \{ \} \) ➞ set of grouped solutions
5: \( B \leftarrow \{ \} \) ➞ set of boundary predicates
6: for all \( s = (u_s, I_s, B_s, prop_s) \in S_{selected} \) do

7: \( S_{grouped} \leftarrow S_{grouped} \cup \{\{s\}\} \)
8: for all \( b^{t,q} \in B_s \) do ➞ iterate over all boundary predicates

9: if \( t \in \{\text{requires, requiresHost, requiresPeer}\} \) then
10: \( B \leftarrow B \cup \{b^{t,q}\} \)
11: if \( q \) is a URI then
12: \( G \leftarrow \text{KB.getSolutionByURI}(q) \)
13: else if \( q \) is a label then
14: \( G \leftarrow \text{KB.getSolutionSetByLabel}(q) \)
15: end if
16: if \( G \setminus S_{selected} \neq \emptyset \) then ➞ if dependencies are not satisfied
17: \( S_{grouped} \leftarrow S_{grouped} \cup \{G\} \)
18: end if
19: end if
20: end for
21: \( \)
22: \( S_{filtered} \leftarrow \{ \} \) ➞ set of filtered grouped solutions
23: for all \( G \in S_{grouped} \) do
24: \( S_{filtered} \leftarrow S_{filtered} \cup \{\text{filteredSolutionSet}(G, R \cup B)\} \)
25: end for
26: \( S_{combined} \leftarrow \text{combinedSolutionSets}(S_{filtered}, R) \)
27: return \( S_{combined} \)

To resolve requirements that are attached to selected solutions, Algorithm 5.2 considers the immediate requirements of each solution of a given solution set.
Since additional solutions (added to satisfy other solutions’ requirements) may themselves have further requirements attached, the algorithm can repeatedly run to iteratively resolve all relevant requirements of a solution set. In order to evaluate whether a given solution set still has unresolved requirements, the unresolvedRequirements map as defined in the following Definition 5.5 can be utilized to decide if the algorithm has to be executed again.

**Definition 5.5 (Map: Unresolved Requirements)**

Let $S \subseteq S$ be a solution set consisting of solutions $s_i = (u_i, L_i, B_i, \text{prop}_i), i \in \mathbb{N}$; let $\mathcal{P}(S)$ be the power set of all solution sets; let $R \subseteq \mathcal{P}$ be a set of potentially unresolved AER predicates; let $\mathcal{P}(\mathcal{P})$ be the power set of all AER and boundary predicate sets. The input of the following map is a solution set and an AER predicate set; its result is a set of predicates $U \subseteq \mathcal{P}$:

$$\text{unresolvedRequirements}: \mathcal{P}(S) \times \mathcal{P}(\mathcal{P}) \rightarrow \mathcal{P}(\mathcal{P}),$$

$$(S, R) \mapsto U$$

The predicate set $U \subseteq \mathcal{P}$ contains all unresolved AER predicates and boundary predicates, i.e., they are not satisfied by any of the given solutions $s_i \in S$:

$$U := \{ u \mid u \in \bigcup B_i \cup R \land (\exists s_i \in S : \text{predicateApplicable}(s_i, u) = \text{true} \land u(s_i) = \text{true}) \}$$

If $\text{unresolvedRequirements}(S, R) \neq \emptyset$, the returned requirements (AER predicates and/or boundary predicates) are not completely satisfied by the given solution set $S$. Therefore, these requirements either must be resolved manually or [Algorithm 5.2](#) is executed (repeatedly) to try to resolve the boundary predicate requirements based on the underlying knowledge base. However, if the knowledge base does not contain an appropriate solution, which satisfies a given unresolved requirement, manual adaptation is needed. Such an adaptation could be implemented by manually creating a solution to resolve
the specific requirements and potentially feed it back to the knowledge base for reuse; alternatively, the solution owning the requirement could be modified to require another kind of solution, which actually exists. As an example, Figure 5.6 outlines a higher-level overview of the repeated execution of Algorithm 5.2 to iteratively refine solution sets. In this case two iterations are executed (i.e., two executions of Algorithm 5.2) to eventually derive a resolved solution set $S_5$ from an unresolved solution set $S_0$. The original solution set $S_0$ typically results from applying Algorithm 5.1 based on an initial set of AERs. The first execution of Algorithm 5.2 produces $S_1$, $S_2$, $S_3$, and $S_4$ as alternative solution sets. Then, $S_1$ is selected to be the input for the second iteration, which produces $S_5$, $S_6$, and $S_7$ as alternative solution sets. Finally, $S_5$ is selected as resolved solution set because it satisfies all AERs and further dependencies specified by the boundary predicates of the included solutions $s_1$ to $s_5$. 

Figure 5.6: Two iterations example for applying Algorithm 5.2
5.4 Generating Application Environment Topology Skeletons

The previous Section 5.3 focused on resolving application environment requirements in order to derive appropriate solution sets. However, such a solution set does not yet represent an application environment because the individual solutions are not explicitly linked. According to the metamodel presented in Chapter 4, an application environment topology skeleton (AET skeleton) resolves a set of application environment requirements (AERs). Therefore, the established concept of application topologies (Section 2.3) is used to specify AET skeletons as topology graphs:

**Definition 5.6 (Topology Graph)**

Let $S_g \subseteq S$ be a solution set consisting of solutions $s_i, s_j \in S_g$ with $i, j \in \mathbb{N}$; let $E_g$ be a set of typed and directed edges $e_k = (s_i, s_j, t) \in E_g$ of type $t \in \{\text{requires}, \text{requiresHost}, \text{requiresPeer}\}$ with $k \in \mathbb{N}$. A topology graph $g = (S_g, E_g)$ is defined as an acyclic and directed graph, consisting of nodes $S_g$ and edges $E_g$. Each $e_k = (s_i, s_j, t) \in E_g$ defines a directed and typed relation between two solution nodes in the graph, with $s_i$ as source node and $s_j$ as target node.

Beside the possibility to manually create topology graphs based on solution sets, Algorithm 5.3 outlines a straightforward procedure to generate an AET skeleton for a given solution set. This approach is similar to what deployment engines do when processing declarative application topology models [OAS13a]. The produced skeleton essentially represents a solution dependency graph. As outlined by the metamodel (Chapter 4), the resulting topology graph requires technical refinement in order to make a deployable AET out of it. There are various reasons why such an refinement is required. As an example, custom glue code often needs to be implemented to technically connect solutions in the topology. In some cases it may be appropriate to wrap the API of certain solutions in order to simplify their orchestration. For example, a certain solution that provides a plain command-line interface typically needs to be wrapped and enriched to provide specific functionality such as provisioning a virtual server of a certain size on a particular Cloud infrastructure. Despite the need for manual refinement, generating a topol-
Algorithm 5.3 generateSkeleton($S_g$)

1: $S_g$ $\text{▷}$ input: solution set
2: $E_g \leftarrow \{ \}$ $\text{▷}$ directed edges representing relations between solutions
3: $g \leftarrow (S_g, E_g)$ $\text{▷}$ output: topology graph
4: for all $s_{source} = (u_{source}, L_{source}, B_{source}, prop_{source}) \in S_g$ do
5:  for all $b^{t,q} \in B_{source}$ : do
6:  if $t \in \{\text{requires, requiresHost, requiresPeer}\}$ then
7:   $\text{pick } s_{target} \in \{s \in S_g \mid \text{predicateApplicable}(s, b^{t,q}) = \text{true} \land$
8:   $b^{t,q}(s) = \text{true}\}$ $\text{▷}$ pick non-deterministically
9:  if ($\nexists$ path $\in g : s_{target} \rightarrow s_{source}$) then
10: $E_g \leftarrow E_g \cup \{(s_{source}, s_{target}, t)\}$
11: end if
12: end if
13: end for
14: end for
15: return $g$

ology skeleton simplifies the process of creating an application environment topology in contrast to starting from scratch using a flat solution set; the required refinement as mentioned previously would have to be performed anyway. This is the main purpose of the algorithm: simplifying the process of building application environment topologies.

As visualized by the simplified example shown in Figure 5.7, the Algorithm 5.3 generates a topology graph by matching requirements and capabilities of the given solutions, which are represented as nodes in the graph. The technical expression of requirements and capabilities is based on established concepts such as described by the Topology and Orchestration Specification for Cloud Applications (TOSCA) [OAS13b]. TOSCA, for example, provides an XML schema definition to specify topology graphs consisting of nodes and relations. Furthermore, nodes can specify requirements and capabilities for matchmaking purposes.
5.5 Chapter Summary and Discussion

The major goal of this chapter was to present fundamental concepts for systematically building application environment topologies. As an initial step toward this goal, diverse classification dimensions of requirements and solutions were discussed. More specifically, four dimensions were considered to be technically relevant for the matchmaking between requirements and capabilities provided by solutions. These dimensions are solution types,
solution bindings, modes of use, and usage styles. Corresponding label taxonomies were outlined to refine these classification dimensions. These hierarchically structured labels are then used to express both requirements and capabilities. Furthermore, algorithms are described to resolve application environment requirements and solution dependencies by identifying and retrieving solutions with corresponding capabilities. These algorithms produce solution sets to be the foundation for eventually building application environments.

In addition, an algorithm was presented to generate topology skeletons based on such solution sets. These skeletons are then manually refined to be deployable as application environment topologies. In terms of generating application topologies, Hirmer et al. [HBBL14] present an approach to address the automated completion of TOSCA-based [OAS13b] topology models. For example, an expert focuses on modeling the application-specific parts of a topology such as the application components and database connectors. These application-specific building blocks express requirements regarding infrastructure, middleware, and platform components that are needed to run instances of the application. Such requirements are satisfied by enriching a topology model with corresponding TOSCA artifacts in order to complete the application topology. Since the approach is completely centered around TOSCA it assumes a reasonable large registry of TOSCA artifacts that provide infrastructure, middleware, and platform capabilities. However, many solutions are not immediately available in the form of TOSCA artifacts. Therefore, Chapter 7 presents an approach to utilize TOSCA as unified modeling representation for different kinds of solutions. This is technically achieved by transforming and wrapping diverse solutions as TOSCA artifacts.

Beside the label taxonomies, solutions with specific capabilities are the fundamental building blocks for resolving application environment requirements in order to build diverse application environments. Therefore, the following Chapter 6 presents approaches for the collaborative and automated gathering of solutions.
Chapter 3 introduced the Gather’n’Deliver method to continuously deliver software applications based on gathered solutions, which are organized using a knowledge base. To implement the gather phase, this chapter presents concepts and approaches [WBFL16] to enable the targeted discovery, capturing, and refinement of various kinds of solutions in a collaborative and automated manner. These provide the foundation for continuously populating and maintaining the underlying knowledge base. While the concepts presented in this chapter immediately support the Gather’n’Deliver method to implement continuous delivery, they are domain-independent and can be applied to other domains, too.

6.1 Collaborative Solution Repositories

Collaboration is a key aspect of the Gather’n’Deliver method because diverse experts such as developers and operations personnel are involved and thus need to collaborate. A broadly established approach to enable col-
laboration are shared repositories. Developers typically collaborate through code repositories based on established version control systems such as Git or Subversion. Beside the source code of an application, such repositories contain test code, build scripts, deployment plans, and further supplementary materials such as documentation. Moreover, configurations are stored in these repositories, often in the form of structured documents using markup languages such as XML, YAML, or JSON. Version-controlled repositories provide a high degree of provenance and transparency because all participants can browse and follow the history of a repository with all its changes. This does not only make collaboration more effective, but also establishes a trustful environment because even the smallest change is visible to all participants.

Experts such as developers and operations personnel are familiar with the approach of maintaining file-based structured documents inside such repositories. While experts are also familiar with the associated tools and languages such as Git, XML, JSON, etc., the approach itself is typically based on standards (XML etc.) and de-facto standards (Git etc.), so there is no strict vendor lock-in. Consequently, each participating expert uses his favorite tooling (e.g., any Git client and XML editor) to effectively participate in collaborative processes, which are centered around such repositories. Diverse participants can collaborate in a decoupled manner because each of them possesses a high degree of freedom when building his local environment. This approach turned out to be successful in the field of software development and operations, both inside and outside of organizations. GitHub\(^1\) is a prominent example for a widely adopted open-source platform and community, which is completely centered around repositories. Beside basic repositories, such platforms typically provide feature-rich collaboration techniques such as pull requests\(^2\), as well as collaborative review and discussion capabilities.

Therefore, this established approach is adapted in this work to be applied to maintain solutions and their metadata as structured documents inside version-controlled repositories. The major goal is to use collaborative solu-

---

\(^1\) [http://www.github.com](http://www.github.com)
tion & metadata repositories (in short solution repositories) as a foundation to establish and maintain a knowledge base, which is made of diverse solutions and their metadata. Instead of the conventional idea of centering repositories around specific software applications, a solution repository stores the metadata describing reusable solutions and the solutions themselves (or references to these). These solutions are used as building blocks of different application environments (Chapter 4), e.g., to implement continuous delivery pipelines. The previously discussed aspects and success factors of repositories (standards-driven, free choice of tooling, etc.) equally apply to solution repositories. As a result, diverse experts can participate in collaborative processes as outlined by the Gather’n’Deliver method (Chapter 3) to maintain and utilize corresponding solutions through a knowledge base, which is based on solution repositories.

The previously introduced metamodel (Chapter 4) prescribes how solutions and their metadata are represented. According to Definition 4.5, a solution is made of the following parts:

- A URI, which uniquely identifies the solution.
- A set of labels to specify the solution’s capabilities.
- A set of links to other solutions or labels (solution boundary) to express requirements, conflicts, and recommendations.
- Arbitrary properties (key-value pairs) to further characterize the solution and attach content such as parameters, files, references, etc.

In order to represent such solutions and their metadata as structured documents in repositories in a normalized form, established markup languages such as XML, JSON, YAML, or Markdown are utilized. The tuple representation of solutions (Definition 4.5) prescribes how these documents are structured. Schema definitions can be utilized to define a concrete serialization for specific languages such as JSON schema [Int13] or XML schema [Wor12]. Listing 6.1 outlines an example for representing a solution as structured document using JSON. However, this is just a single alternative
how to represent a solution using JSON; another representation could be based on XML[1]

Similar to application source code, these structured documents can be collaboratively maintained and shared using version-controlled repositories, i.e., collaborative solution & metadata repositories. As discussed previously, diverse experts can use their preferred tooling to create, edit, and update these documents. The example outlined in Listing 6.1 underlines the importance of labels, following the label path representation as introduced in Chapter 5. On the one hand, labels are utilized to specify characteristics (e.g., being an executable of type Chef Cookbook) and capabilities (e.g., providing an Apache HTTP Server) of a solution. On the other hand, links can refer to labels instead of solutions, for example, to express requirements that must be satisfied by solutions providing corresponding capabilities (e.g., providing a Debian operating system). These labels are organized using

http://gist.github.com/jojow/03f368aad0326273e8b5
label taxonomies as discussed in Section 5.2. The taxonomies are not static, but must be changed if, for instance, new categories of solutions appear, or existing categories of solutions are refined. Therefore, an ongoing and collaborative taxonomy evolution is happening. For this reason the label taxonomies are also maintained through structured documents inside solution repositories. Listing 6.2 shows an extract of the middleware type taxonomy, rendered using YAML. Beside the label hierarchy, the taxonomy contains alias labels (e.g., Apache HTTPD is an alias for Apache HTTP Server), which can be used alternatively to their primary labels. This is to cover and map different naming conventions and established terms used by diverse communities.

To sum up, solution repositories in conjunction with structured documents enable effective collaboration among diverse experts. The entire approach is based on established practices and tooling how software developers and operations personnel collaborate in modern environments. However, fully manually maintaining all parts of these repositories does not scale due to the potentially huge amount and variety of solutions. Especially the solutions’ metadata quickly become outdated because solutions could be developed by people different from the maintainers of solution repositories. To address this issue and to simplify the maintenance of solution repositories, the following Section 6.2 presents an approach to automate the discovery of certain solutions and store them in solution repositories. Furthermore, there is another challenge that must be addressed, namely how to query and efficiently utilize the knowledge base, which is based on the previously discussed solution repositories. While managing structured documents stored in potentially distributed solution repositories nicely works for collaboratively maintaining the knowledge base, these repositories typically do not provide fine-grained query mechanisms to utilize the knowledge base in an efficient manner. Moreover, consistency checks are not made, e.g., by verifying whether the given documents are structured properly and the utilized labels actually comply to the label taxonomy. Therefore, Section 6.3 presents an automated approach (i) to check the involved solution repositories for consistency, and (ii) to generate a consolidated variant of the knowledge
Listing 6.2: Example for YAML representation of a label taxonomy
base, providing a query interface. The idea of solution repositories was previously introduced in a different context, namely in the domain of pattern research [FBFL15]. While in this field a solution repository is used to store and manage concrete artifacts that implement abstract patterns [FBB+14, FBFL15], the presented approach also comprises knowledge artifacts such as fine-grained documentation and technical tutorials that are not immediately executable.

6.2 Automated Discovery and Capturing of Solutions

In the previous Section 6.1 collaborative solution repositories were introduced. However, it was assumed that these repositories are mostly populated and maintained manually, which does not scale due to the potentially huge amount and variety of solutions. Therefore, Figure 6.1 presents an overview of the auto-gather\textsuperscript{1} pipeline to automate the discovery and capturing of solutions. The pipeline consists of five stages: (i) during discover stage, existing solutions from different sources such as Chef Supermarket\textsuperscript{2} and Docker Hub\textsuperscript{3} are identified. (ii) The retrieve stage consumes solution references, mostly URLs, which are produced during discover stage. These references are then

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{pipeline.png}
\caption{Overview of auto-gather pipeline \cite{WBFL16}}
\end{figure}

\textsuperscript{1}Automated gathering of solutions
\textsuperscript{2}http://supermarket.chef.io
\textsuperscript{3}http://hub.docker.com
resolved by retrieving the raw solutions, i.e., the unmodified solutions as they are stored in their original source. The retrieval can happen through diverse channels such as HTTP, Git, Bazaar, Rsync, etc., depending on where a particular solution is located. (iii) During extract stage, relevant metadata are extracted and derived from previously retrieved raw solutions. (iv) The normalize stage aligns the representation of relevant metadata with the previously introduced metamodel (Chapter 4) as discussed in Section 6.1. (v) Finally, during enrich stage, metadata are refined, e.g., by applying document classification techniques for assigning labels to solutions to better characterize their capabilities. A simple classification approach would be to match keywords between the label taxonomies and the metadata of a solution. These normalized and enriched metadata are eventually stored in corresponding solution repositories. Obviously, the solution repositories may be accessed during enrich stage to retrieve the current label taxonomies, which enables the assignment of additional labels to retrieved solutions for classification purposes. Moreover, the pipeline can be optimized in various ways. For example, during retrieve stage, the solution repositories may be checked whether a particular solution is already captured. In this case the retrieval and all following stages can be skipped for this particular solution.

The auto-gather pipeline is not meant to replace the mostly manual collaborative approach based on solution repositories, which was presented in Section 6.1. The two approaches are rather complementary, so the auto-gather pipeline populates solution repositories in an automated manner, while diverse experts are still able to collaboratively maintain additional or refine existing solutions. This hybrid approach is a significant improvement over just manually maintaining solution repositories as discussed in Section 6.1: the automated gathering of solutions makes the entire approach more scalable because large amounts of solutions can be automatically discovered and captured.

Although the solution repositories, partly populated automatically and manually, provide a collaborative foundation for a comprehensive knowledge base, there are still two major issues: (i) the solutions stored inside the repositories are not checked for consistency, e.g., whether their representation
complies with a given schema or the utilized labels are actually valid regarding the label taxonomies (Section 5.2). (ii) Fine-grained query mechanisms are missing; however, these are required to utilize the knowledge base in an efficient manner. To address these issues, Section 6.3 presents an automated approach for performing consistency checks and generating consolidated variants of the knowledge base providing a proper query interface.

6.3 Continuous Delivery of Knowledge Base Variants

As discussed previously, solution repositories enable the systematic gathering and maintenance of diverse solutions and their metadata in a collaborative manner. However, solutions and metadata inside such repositories are simply represented as file-based structured documents as explained in Section 6.1. This approach is similar to managing application-specific source code files through code repositories. Consequently, neither consistency checks are performed when adding or modifying files, nor fine-grained query mechanisms are provided to find and identify appropriate solutions for a specific application environment. These deficiencies are addressed by adapting established continuous integration and continuous delivery practices [HF10], because they solve similar issues when dealing with source code repositories for specific software applications as discussed in Section 2.1. A continuous delivery pipeline runs various tests to check the correctness and consistency

Figure 6.2: Overview of deliver-knowledge-base pipeline [WBFL16]
of newly committed and modified source code. Moreover, such a pipeline continuously delivers updated instances of the corresponding application through automated build and deployment steps. Figure 6.2 outlines an overview of the deliver-knowledge-base pipeline, which makes use of these key concepts to continuously deliver updated instances of a specific knowledge base variant. With this approach, the ‘eat your own cooking’ principle is applied by utilizing and adapting the key concepts of continuous delivery pipelines to deliver knowledge base variants, which themselves are used to implement continuous delivery pipelines and corresponding application environments for specific applications.

The stages of the deliver-knowledge-base pipeline are similar to the various stages of common continuous delivery pipelines. However, they are adapted to fit the needs of implementing continuous delivery of a knowledge base instead of a specific software application. For this purpose the initial aggregate stage consolidates solutions captured in potentially multiple solution repositories. Reasons why solutions may be distributed across different solution repositories can be diverse: beside plain separation of concerns, some solutions may be private, while others are public or at least shared among several organizations. Optionally, filters could be applied in this stage, e.g., to exclude certain solutions, which should not be part of the resulting knowledge base. During optimize stage, the aggregated set of solutions can be refined in various ways. For example, duplicate solutions can be eliminated to avoid polluting the knowledge base. Furthermore, the labels describing requirements and capabilities of solutions can be normalized by replacing alias labels by their primary ones. The test stage covers the previously mentioned consistency checks. This may include schema-based validation of solutions’ metadata, as well as checking whether the utilized labels are valid regarding the label taxonomies (Section 5.2). Then, the linked set of solutions are rendered and packaged as content of the knowledge base during build stage. This is similar to building the binaries of an application, which can then be deployed in the next step, i.e., during deploy stage.

The deployment eventually results in concrete variants of the knowledge base. These variants, which are created through the deliver-knowledge-base
pipeline, provide query mechanisms to retrieve appropriate solutions. The technical foundation of such a knowledge base variant can be diverse. To make a few examples, a variant could be provided as (i) a relational database in conjunction with a REST API, (ii) a Web-based GUI with a full-text search engine as back-end, (iii) a graph database with a corresponding query API, or (iv) an RDF store with a SPARQL query interface. Multiple delivery knowledge-base pipelines can be established to deploy diverse knowledge base variants, which may be targeted to different groups of users. Similar to deploying software applications to different environments (development, test, production, etc.), knowledge base variants can be deployed to different environments, too. For example, a developer can run a minimal variant locally on his developer machine, whereas the organization may run a full-blown and scaled out variant in its private Cloud environment.

The presented approach of continuous delivery of knowledge base variants, backed by collaborative solution repositories is based on several concepts, which are successfully established to collaboratively develop and operate applications. Automated continuous delivery pipelines and source code repositories are two key building blocks in this context. However, any knowledge base is naturally associated with the risk of getting outdated, especially if the covered scope of knowledge changes quickly. The knowledge base discussed in this work aims to cover a huge variety of solutions maintained in solution repositories. These solutions provide building blocks for establishing continuous delivery pipelines, as well as implementing deployment automation. Since the communities centered around these fields are very active and disruptive, it is a challenge to keep up with constantly updating and newly emerging solutions. Consequently, the risk of an outdated knowledge base is a real problem in this context. The presented approach addresses this risk with two approaches: (i) the automated discovery and capturing of solutions [Figure 6.1] keeps at least parts of the underlying solution repositories updated. (ii) By creating knowledge base variants through automated delivery pipelines [Figure 6.2], these variants are always built and deployed based on the latest revisions of the involved solution repositories. This helps to avoid the creation and usage of variants, which provide an outdated set of
solutions. However, these two approaches do not help to keep the manually maintained parts of the solution repositories and generated knowledge base variants updated. This risk is mitigated by the significant overlap between potential producers and consumers of the knowledge base. Developers, operations personnel, and further experts utilize, i.e., consume the knowledge base, but they also produce contents and add it to the knowledge base in the form of updated and newly added solutions. This overlap aims to keep users of the knowledge base motivated to also contribute contents to the underlying repositories because they are immediately interested in keeping the quality of the resulting knowledge base high. Otherwise the knowledge base becomes less usable for them over time. Another motivation for contributing solutions may be the fact that such solutions increase their visibility to foster reuse by other experts. With this approach, the knowledge base and the underlying shared solution repositories foster collaboration between different kinds of experts, thereby implicitly supporting recently emerging software development paradigms such as DevOps [BWZ15]. As discussed previously, the presented approach utilizes established concepts and tooling such as structured documents maintained through version-controlled repositories and automated delivery pipelines. Consequently, the barrier for contributing to solution repositories and the resulting knowledge base is relatively low for potential users such as developers and operations personnel. This fact also helps to lower the actual risk of an outdated knowledge base.

6.4 GatherBase Framework

The previous sections of this chapter discussed several key concepts, which are required to achieve collaborative and automated gathering of solutions. In particular, Section 6.1 introduced the concept of collaborative solution repositories. These are maintained by experts. However, when dealing with a huge amount and variety of solutions, manually maintaining all solutions does not scale. Therefore, Section 6.2 outlined an approach to automate the discovery and capturing of solutions. Moreover, Section 6.3 discussed
Figure 6.3: Overview of GATHERBase framework [WBFL16]
how knowledge base variants can be established by applying continuous delivery principles. The presented concepts are abstract and generic enough to be applied and implemented in various ways. This section aims to present an integrated framework, namely the GatherBase framework, which covers all previously discussed concepts from gathering solutions in solution repositories to creating diverse variants of the knowledge base. A corresponding prototype implementation of this framework is presented later in Section 8.2.1.

Figure 6.3 shows an overview of the GatherBase framework, which is fully plugin-based to be modular and extensible. The upper part (above solution repositories) covers the auto-gather pipeline described in Section 6.2 for the automated gathering of solutions. This part of the framework essentially provides a crawler to discover and capture existing solutions. Three kinds of job processing components (i.e., job processors) connected by corresponding job queues provide a loosely coupled system, covering the stages of the auto-gather pipeline. Specialized discovery job processors cover the discover stage. As an example, the Chef cookbook discoverer performs a targeted discovery of Chef cookbooks, which are provided through the Chef Supermarket. The discovery may be triggered in various ways. A straightforward approach would be to run each discoverer periodically at certain time intervals to check for updated and newly added solutions. Alternatively, an event-based approach may be followed, e.g., by a discoverer subscribing to a specific source, which notifies the discoverer about changes. Consequently, the corresponding discoverer can react to incoming change notifications instead of periodically polling for updates. For example, GitHub webhooks\(^1\) can be utilized to react on changes that are pushed to a Git repository.

Each discoverer produces a separate retrieval job for each discovered artifact or solution and puts the job into the retrieval job queue. Jobs in this queue are asynchronously consumed by retrieval job processors, which cover the retrieve stage of the auto-gather pipeline. Depending on where a specific solution is stored, a corresponding retriever is invoked. If, for instance, a

\(^1\)http://developer.github.com/webhooks
particular Chef cookbook is stored in a Git repository, the Git retriever is used to fetch the cookbook.

For each successfully retrieved solution, a separate handle job is put into the handle job queue. This queue is consumed by diverse handle job processors, in short handlers, which cover the stages extract, normalize, and enrich (Figure 6.2). Some handlers are run for all solutions, others are only run for specific kinds of solutions. As an example, a ZIP file handler is utilized to extract the contents of a solution during extract stage. Then, a Chef metadata handler may be used (specifically for Chef cookbooks only) to transform Chef-specific metadata into a normalized representation during normalize stage. The classifier uses the label taxonomies to categorize a given solution by adding labels to the solution to specify its capabilities. This happens during enrich stage. Finally, the writer puts the normalized and enriched solution metadata – bundled with the solution itself or a reference to it – into a solution repository. To determine which handlers are run in which order for a specific handle job, a set of rules are specified as user-defined configuration to be evaluated at runtime. As a result, an individual chain of handlers is dynamically identified at runtime for each handle job. For instance, a simplified handler chain for a Chef cookbook may be as follows: ZIP file handler $\rightarrow$ Chef metadata handler $\rightarrow$ classifier $\rightarrow$ writer.

The middle part of Figure 6.3 positions the solution repositories as link between auto-gather pipeline building blocks (upper part) and deliver-knowledge-base pipeline building blocks (lower part). Beside solutions and their metadata, these repositories contain the label taxonomies, which are, for example, used by the classifier to categorize solutions according to the taxonomies. In addition to the auto-gather pipeline, solutions in repositories can be maintained and refined by experts manually.

The lower part (below solution repositories) covers the deliver-knowledge-base pipeline described in Section 6.3 for the creation of concrete knowledge base variants. For each stage, a set of modules is provided, which are used to establish different variants of the pipeline, depending on the kind of knowledge base variant that should be created. Figure 6.3 outlines two example pipelines that can be implemented by combining different sets of
modules across the pipeline stages to eventually create diverse knowledge base variants. **Pipeline 1** uses reader and filter during aggregate stage to fetch selected solutions from the solution repositories. The deduplicator is utilized during optimize stage to remove duplicate solutions. Moreover, the label normalizer replaces alias labels by primary labels according to the label taxonomies. Next, during test stage, the schema validator checks whether the solutions’ metadata are represented properly according to given schema definitions such as an XML schema definition for solution metadata documents, which are expressed using XML. Of course, this can only happen if a corresponding schema is defined. The label validator checks the utilized labels (for expressing requirements and capabilities of solutions) against the label taxonomies (Section 5.2) to ensure whether they are valid labels. Once the test stage finished successfully, the knowledge base variant is built during build stage. In case of pipeline 1, an Elasticsearch instance\(^1\) is populated with the contents of the knowledge base, providing the back-end of the knowledge base variant. Moreover, a Web UI is packaged as front-end together with the populated Elasticsearch back-end to ease the interaction with the knowledge base variant. Finally, the packaged variant is deployed using Docker on a developer’s machine during deploy stage.

**Pipeline 2** outlines a second example: the stages aggregate, optimize, and test are pretty similar to pipeline 1 in terms of the utilized modules. However, the filter is omitted in this case, i.e., all solutions stored in the repositories are considered. The build and deploy stages are completely different: in this case a MySQL database\(^2\) is populated with the contents of the knowledge base. A REST API is packaged together with the MySQL back-end to provide a Web-based query interface for the underlying knowledge base. This knowledge base variant is then deployed to Amazon EC2\(^2\) to run as a Cloud-based service. Many other variants of the pipeline could be established using this framework. Such a pipeline can then either be triggered on each commit to one of the solution repositories, or it can be triggered on demand or periodically at certain time intervals. The capability and flexibility of

\(^{1}\)http://www.elastic.co/products/elasticsearch  
\(^{2}\)http://aws.amazon.com/ec2
maintaining different pipelines to deliver diverse knowledge base variants shows the broad applicability of the GatherBase framework. A specific knowledge base variant is not meant to be specialized on a certain kind of solutions, but aims to provide a useful and efficient interface (for a certain user group or usage scenario) to query and access solutions. This helps to decouple solution repositories from the generated knowledge base and enables the implementation of a broad diversity of alternative knowledge base variants.

6.5 Chapter Summary and Discussion

The previously presented GatherBase framework provides a comprehensive approach to implement the collaborative and automated gathering of solutions as discussed in this chapter. It supports several steps of the Gather’n’Deliver method described in Chapter 3 to systematically gather, maintain, and identify building blocks for implementing continuous delivery pipelines for specific applications. GatherBase builds upon three fundamental concepts that were introduced in this chapter: (i) collaborative solution repositories, (ii) the automated discovery and capturing of solutions, and (iii) the continuous delivery of diverse knowledge base variants. More specifically, collaborative solution repositories are based on several established and proven concepts known from code repository platforms to enable experts to reuse their expertise while collaboratively working with solution repositories. By automating parts of the solution discovery and capturing, the entire approach scales better because the amount of covered solutions can be significantly increased. In order to utilize the knowledge base that is composed of the captured solutions through an efficient query interface, concrete knowledge base variants providing different kinds of query mechanisms can be created using corresponding continuous delivery pipelines for these variants. Further technical details of the GatherBase framework are discussed in Chapter 8 in the context of its prototype implementation.

The next step in the Gather’n’Deliver method is the orchestration of
diverse solutions, which have been identified using the knowledge base described in this chapter. This is required to eventually establish application environments, which are the foundation to operate concrete continuous delivery pipelines for specific applications. Therefore, the following Chapter 7 discusses how to streamline the orchestration of diverse solutions.
STREAMLINED ORCHESTRATION OF DIVERSE SOLUTIONS

The Gather’n’Deliver method (Chapter 3) provides the foundation to continuously deliver software applications based on gathered solutions. Chapter 6 mostly focused on the gather phase, describing required concepts for collaboratively gathering solutions and providing diverse knowledge base variants with systematic and fine-grained query mechanisms. To refine the method’s deliver phase, this chapter presents concepts and approaches to streamline the orchestration of diverse solutions, which potentially have been selected from the knowledge base. The combined usage of various kinds of solutions is often required because they are differently specialized \[BBK+13, WBL14a, BBK+15, WBL15b\]. This is due to various reasons such as diverse technologies and differently focused communities that provide corresponding solutions. In particular, a major goal and focus of this chapter is to provide proper APIs for executable solutions such as virtual machine images, container images, deployment scripts, and configuration definitions. As a result, the orchestration of solutions is streamlined and simplified by orchestrating proper APIs instead of integrating highly heterogeneous lower-level artifacts.
The purpose of orchestrating solutions is to foster their combined usage at runtime and thus forms the technical foundation for creating application environments as part of continuous delivery pipelines for specific applications. While the concepts presented in this chapter immediately support the GATHER’n’DELIVER method to implement continuous delivery, they are domain-independent and can be applied to other domains, too.

7.1 TOSCAfy Framework – Standards-driven Transformation

The initial step toward streamlining the orchestration of diverse solutions is to homogenize their representation. Solutions tend to be heterogeneous due to diverse maintainers, communities, and toolchains. Consequently, a significant orchestration challenge appears at modeling time already because each kind of solution specifies its interface, dependencies, and boundaries differently. The following sections present the concepts of TOSCAfy as an automated and standards-driven transformation framework. Its goal is to provide extensible transformation capabilities to utilize the Topology and Orchestration Specification for Cloud Applications (TOSCA) [OAS13b] as standardized modeling representation of diverse solutions. Therefore, the following Section 7.1.1 describes the key concepts of TOSCA, which is a standard to model and serialize application topologies and their potentially reusable building blocks, i.e., the individual solutions of which topologies are made. As introduced in Section 2.3 topology models are used to specify the structure and bundle all building blocks of individual application environments (development, test, production, etc.), which implement the stages of a continuous delivery pipeline. The presented concepts are not bound to TOSCA, so other approaches could be utilized to provide a unified modeling representation of diverse solutions. However, in the following sections TOSCA is used as a standardized approach to avoid inventing another modeling language and to make the presented concepts more concrete.
Figure 7.1: TOSCA type definitions and topology templates [WBL14c]

7.1.1 TOSCA as Standardized Modeling Representation of Solutions

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [BBL12; OAS13b; BBL14b] is a standard supported by several industrial companies and used in academic research projects [WBB+13; BBH+13; KBBL13; HGG13; BBK+14; CCP14; WBB+14b; MCE15]. Its main goal is to enhance the portability and management of Cloud applications. Technically, the TOSCA standard provides an XML-based language to ensure a unified serialization of models and individual modeling constructs. In addition, a ‘simple profile’ of TOSCA is defined based on YAML [OAS16]. TOSCA is centered around the concept of application topology models and their individual building blocks as outlined in Section 2.3. Topology templates are defined as graphs consisting of nodes and relationships to specify the

http://www.oasis-open.org/committees/tosca
topological structure of an application. As a foundation for defining such templates, *node types* and *relationship types* are defined as shown in Figure 7.1. These are used to create and wire corresponding *node templates* and *relationship templates* as part of a topology template. A comprehensive type system can be introduced because types may be derived from other existing types in the sense of inheritance as it is used, for instance, in object-oriented programming. As an example, an abstract *Java Servlet Container* node type can be defined, which has a child node type *Apache Tomcat*. There could be further node types derived from the latter, such as *Apache Tomcat 6.0* and *Apache Tomcat 7.0*.

Types consist of further sub-elements: *operations* are attached to nodes and relationship types, for instance, to cover their lifecycle: *install*, *start*, *stop*, etc. [OAS13a]. In addition, further management operations may be defined such as *backup-database* and *restore-database*. These operations are implemented by *implementation artifacts* (IAs), which could be any kind of executables such as Chef cookbooks or Unix shell scripts. An IA is executed when the corresponding operation is invoked, e.g., by the TOSCA runtime environment. Operations belonging to relationships are distinguished in *source operations* and *target operations* because a relationship links a source node with a target node. Source operations are executed on the source node, target operations on the target node. To ease the wiring of nodes and relationships, they specify *requirements* and *capabilities* that are used for matchmaking purposes. For instance, a Java application node type may specify a *Java Servlet Runtime* requirement, whereas the *Apache Tomcat* node type provides a matching *Java Servlet Runtime* capability. A relationship specifying the *Java Servlet Runtime* requirement as valid source and the *Java Servlet Runtime* capability as valid target can be used to wire the two nodes. *Properties* can be specified as arbitrary data structures, e.g., using XML schema definitions to make nodes and relationships configurable. As an example, the *Apache Tomcat* node type may provide a property to specify the directory where the log files are stored at runtime. Properties are typically exposed to the operations and their IAs, so they can be considered during execution. Finally, TOSCA specifies the structure of *Cloud Service Archives*
(CSAR) as a portable, self-contained packaging format for application topology models. Beside the schema definitions of types and templates, a CSAR contains all artifacts that are used, e.g., IAs such as scripts, executables, and configuration definitions. Due to the fact that a CSAR is self-contained, a TOSCA runtime environment can process it by traversing the topology template to create instances of the application. Each node typically provides lifecycle operations such as install that are invoked when the topology template gets traversed to create instances of the nodes and relationships on a specific infrastructure.

TOSCA supports the deployment and management of applications modeled as topology templates by two different flavors: (i) imperative processing and (ii) declarative processing. The imperative approach employs management plans to orchestrate the management operations provided by node types and relationship types. These plans execute the implementation artifacts that are attached to the corresponding operation. The execution of such plans can be performed automatically and is typically implemented using workflow languages such as BPEL, BPMN, or the BPMN extension BPMN4TOSCA. Management plans are packaged with the corresponding CSAR to make CSARs self-contained. Thus, imperative TOSCA runtime engines run these plans to consistently execute management tasks such as deploying instances of an application. In contrast to the imperative approach, the declarative approach does not require any plans: a declarative TOSCA runtime engine derives the corresponding logic automatically by interpreting the topology template. However, the declarative approach often does not work for complex management tasks such as modifying an application topology or migrating parts of it. Therefore, imperative management plans need to be created and maintained.

With all these concepts in place, TOSCA aims to be used as standardized modeling representation of diverse solutions, i.e., different kinds of artifacts can be wrapped and represented as node types and relationship types. Manual approaches exist to create TOSCA modeling artifacts based on such solutions. However, this is time-consuming and error-prone, so it does not scale for a large number
of artifacts. Therefore, the following Section 7.1.2 outlines the architecture of the extensible and modular TOSCAfy framework. Its main goal is the automated transformation of diverse artifacts toward TOSCA modeling artifacts.

7.1.2 Framework Architecture

The automated transformation of heterogeneous artifacts toward TOSCA is enabled by the modular TOSCAfy framework. Its architecture is outlined in Figure 7.2. The core of the transformation logic lives in the prepare & transform tier: specialized modules implement the highly domain-specific transformation applied to certain kinds of artifacts. For example, the Chef-specific transformer and dependency resolver modules can process Chef cookbooks by
analyzing their content and metadata to retrieve their dependencies and to eventually generate corresponding self-contained TOSCA node types and relationship types. Retrieving the original artifacts and their dependencies happens through the retrieve tier, which provides modules to enable the retrieval through different channels such as an HTTP endpoint, a Git repository, or a locally stored file. Such artifacts may be solutions retrieved from a knowledge base variant as discussed in Section 6.4. A domain-specific transformer module has to be implemented for each kind of artifact that is processed through the TOSCAfy framework. The implementation of a complementary domain-specific dependency resolver is optional due to different reasons: some artifacts are bound to specific tooling (e.g., Dockerfile bound to the Docker engine), which is capable of resolving dependencies at runtime. Therefore, dependencies do not need to be resolved and retrieved when generating TOSCA artifacts. Another reason for skipping the implementation of a dependency resolver is the case when artifacts are bundled together with their dependencies in a self-contained manner. This applies, for example, to exported Docker container images. Originally, TOSCA was technically based on an XML schema definition to serialize TOSCA modeling artifacts. Later, the YAML-based ‘TOSCA simple profile’ was created to provide an alternative serialization of TOSCA models. These alternatives are considered as part of the serialize tier by providing corresponding serializer modules to export transformed artifacts in different ways. Finally, the package tier provides modules to bundle transformed and serialized TOSCA artifacts. TOSCA specifies Cloud Service Archives (CSAR) as a portable packaging format, so a CSAR packager module can produce CSARs based on previously transformed and serialized TOSCA artifacts. A CSAR either holds a complete topology model, e.g., representing a specific application environment, or it contains a set of modeling artifacts such as node types and relationship types. In the context of the TOSCAfy framework, the latter approach is followed because the goal is to homogenize reusable
solutions, i.e., existing artifacts available as building blocks to create application topologies for diverse application environments. Generated CSARs are then stored in a CSAR repository [KBBL13], which can be technically implemented in various ways such as a simple file system-based solution, using a database, or utilizing a version-controlled repository.

The presented framework is fully extensible at all tiers, so both domain-specific and generic modules can be added to make the framework more powerful. Although the transformation process and framework is centered around TOSCA to provide a standardized modeling representation of diverse solutions, additional **serializer** and **packager** modules can be plugged into the framework to produce non-TOSCA artifacts. As an example, corresponding modules can generate CloudML artifacts [BRF+15] instead of TOSCA node types. However, the previously discussed TOSCA metamodel (Section 7.1.1) is still being used internally by the framework as a unified modeling representation of heterogeneous artifacts. Consequently, each **transformer** module produces TOSCA node types and relationship types, and each **serializer** module consumes such TOSCA types, but may also produce non-TOSCA artifacts.

7.1.3 Example: Chef to TOSCA Transformation

Previously, Section 7.1.2 presented an extensible architecture of the TOSCAfy framework to homogenize the modeling representation of diverse kinds of artifacts. The core of the automated transformation toward TOSCA as a standard is driven by domain-specific **transformer** and **dependency resolver** modules. This section aims to describe a detailed example for the concrete transformation mechanisms to generate TOSCA node types and relationship types based on existing Chef cookbooks. These mechanisms can finally be implemented as modules and plugged into the previously discussed framework. In addition to the TOSCA basics discussed in Section 7.1.1, Chef’s key concepts and building blocks are outlined in the following to explain the transformation mechanisms subsequently.

Chef [Nel13; SW14] is a configuration management and infrastructure
automation framework, which provides a domain-specific language (Chef DSL) based on Ruby. The Chef DSL is used to define configurations of resources such as virtual machines. These configuration definitions are called recipes. Multiple recipes are bundled in cookbooks. For instance, a MySQL cookbook may provide the recipes install_server, install_client, and install_all to deploy the corresponding components of MySQL. In terms of dependencies, a particular recipe can depend on other recipes, either bundled in the same or other cookbooks. This helps to extract reusable parts of configuration definitions such as the configuration of common operating system management tools and include these into different recipes. The definition of declarative expressions such as ensure that MySQL server is installed is the recommended way to define portable configurations in recipes. However,
imperative expressions such as operating system command statements (e.g., “apt-get install mysql-server”) can be used, too. Figure 7.3 provides an overview of the core concepts of Chef: the Chef server acts as a central management instance for recipes, run lists, and attributes. Each node has a run list and a set of attributes assigned. The run list of a particular node specifies the set of recipes to be executed on this node. Many recipes have variability points such as the version number of the mysql package as shown in the sample recipe in Figure 7.3. To resolve these variability points at runtime, attribute definitions such as mysql_ver='5.5’ can be assigned to a node. These attributes can be read during execution time. Finally, a Chef workstation runs knife\footnote{http://docs.chef.io/knife.html} to control the Chef server, as well as all nodes and data that are registered with the Chef server.

The transformation mechanisms for generating TOSCA node types out of Chef cookbooks can be based on different levels of granularity. Figure 7.4 outlines a fine-grained transformation approach: a separate node type is created per cookbook. Each recipe is attached to the generated node type as a TOSCA implementation artifact that implements a particular operation as discussed by Wettunger et al. [WBB+13]. A single node type capability is generated for each cookbook based on the cookbook name and cookbook URL. Each cookbook possesses a metadata.rb file, containing information such as name, version, and maintainer of the cookbook. Moreover, all dependencies on other cookbooks are defined inside the metadata.rb file. For each dependency a corresponding requirement is generated. Relationship types are not required to be generated because Chef does not specify any relationship-specific wiring logic outside recipes. Therefore, generic depends on relationships are used to wire nodes derived from node types, which were generated from cookbooks. Finally, attribute definitions (*.rb files) stored in the attributes sub-directory of a cookbook are used to derive node type properties. These are used to configure node templates derived from the node type, parameterizing the operations execution that are attached to the node type.
Alternatively, a more coarse-grained transformation approach of Chef cookbooks is shown in Figure 7.5: node types are generated only for cookbooks that are used immediately. Cookbook dependencies are bundled inline as implementation artifacts. Consequently, dependencies do not appear as separate node types, so a single node type may wrap multiple cookbooks. The main motivation for this approach is the fact that cookbooks and their dependencies may be very fine-grained. For instance, the Apache HTTP server cookbook depends on the cookbooks logrotate, iptables, and pacman. These are operating system-level software packages that are not supposed to appear in the application topology as individual nodes, i.e., TOSCA node

\[\text{http://supermarket.chef.io/cookbooks/apache2/versions/2.0.0}\]
templates. This is because an application topology typically models more coarse-grained middleware and application components such as Web servers and databases. The coarse-grained transformation skips the creation of lower-level node types, thereby avoiding the pollution of topologies. A more advanced approach would be to make the transformation configurable, e.g., by explicitly specifying which cookbooks are too low-level to be provided as separate node types. Consequently, these specifically defined cookbooks are transparently bundled as implementation artifacts as outlined in Figure 7.5; for the other cookbooks, individual node types are generated as shown in Figure 7.4.

The concrete example for Chef-specific transformation mechanisms can
be transferred to other technologies such as Puppet \cite{Loo11} and CFEngine \cite{Zam12}, because their key concepts are very similar to Chef. Consequently, the transformation approach discussed in this section can be adapted to be applied to these artifacts, too. By enriching the TOSCAfy framework with corresponding modules that implement such transformation mechanisms, it becomes a powerful approach to automate the transformation of various kinds of artifacts toward TOSCA. The result is a unified modeling representation of diverse artifacts. However, this is the initial step only toward the major goal discussed in this chapter: streamlining the orchestration of diverse solutions such as executable artifacts. While unifying the modeling representation of artifacts eases their handling for modeling and development, it does not help a lot at runtime. Figure 7.5 demonstrates how Chef-specific concepts are mapped to TOSCA concepts such as Chef recipes represented as operations on the TOSCA side with Chef implementation artifacts attached. As a result, the orchestration layer (TOSCA engine, overarching deployment plan, etc.), which runs an operation of a node type that was previously derived from a Chef cookbook, has to consider all technical details of invoking a Chef recipe, passing parameters, collecting invocation results, etc. The unified modeling representation does not significantly reduce complexity at runtime, so the orchestration layer has to deal with a broad variety of implementation artifacts and their huge technical diversity at runtime. In order to avoid polluting the orchestration layer with lower-level technical details to keep overarching deployment plans and engines maintainable, the PLUGINVOKE framework is presented in the following Section 7.2. Its major goal is to improve the handling and invocation of diverse artifacts at runtime and thus to complement the modeling-centric transformation approach discussed in this section.

7.2 PLUGINVOKE Framework – Pluggable Invocation

Despite a unified modeling representation of diverse artifacts as discussed in Section 7.1, their invocation at runtime is still a major challenge. The
orchestration layer such as overarching deployment plans have to deal with lots of lower-level technical details to make the invocation happen. Beside the individual invocation mechanisms of certain kinds of artifacts, the invocation needs to be prepared, e.g., by setting corresponding parameters and copying files to the target environment. Moreover, results of the invocation such as logs and further output must be collected in different ways for different kinds of artifacts.

Therefore, PlugInvoke is presented as a pluggable invocation framework, which aims to provide higher-level APIs to be used for the orchestration of diverse artifacts. This complements the TOSCAfy framework and enables the orchestration layer to utilize a common interface to transparently invoke artifacts of very different kinds, abstracting from artifact-level implementation-specific details. The goal is to make the orchestration layer such as an overarching deployment plan completely focus on the flow logic, e.g., by simply invoking operations attached to TOSCA node types and relationship types, without polluting the plan with lower-level actions such as copying files through SSH and setting command line parameters. Technically, architecture concepts of enterprise service buses (ESBs) [Jos07; Cha04] are considered and reused.

7.2.1 Framework Architecture

The PlugInvoke framework is based on a modular architecture as outlined in Figure 7.6. This is to avoid a monolithic architecture, which would be hard to maintain and extend. Since extension points are vital for the framework due to the large variety of artifacts, the architecture is pluggable at multiple tiers to be extended by specialized modules. The framework’s external-facing uniform interface, which abstracts from lower-level invocation APIs, is provided by the invocation & management API. Technically, it could be, for example, implemented as HTTP-based RESTful API [RAR13]. The API receives different kinds of requests. Corresponding request handlers process these incoming requests and send back responses once the required actions have been carried out. The artifact request handler deals with all requests
that are relevant to managing artifacts such as receiving a deployment script, storing it in the artifact database through the data access API, or fetching it again from there to respond to another request that asked for the particular artifact. Actual invocation requests are managed by the invocation request handler, which retrieves the corresponding artifact from the artifact database (if it is not contained in the request) and triggers its invocation through the corresponding invoker, for example, the Chef invoker if the given artifact is a Chef cookbook. In addition, a new invocation instance is created inside the instance database, containing various instance data such as the parameters and the current status of the invocation. In case there is no invoker that can deal with the given artifact, the invocation request handler responds with an
‘invoker missing’ error. The instance request handler reads from the instance database to serve requests, which ask for specific instance data of a particular invocation.

Beside the request handling tier, only the invoke tier needs to access the data tier through the data access API. At the invoke tier, specialized invoker modules can be plugged into the framework. Figure 7.6 outlines two examples, namely the Chef invoker and the Shell script invoker, which encapsulate all the required technical details how to invoke and run corresponding artifacts. By plugging invokers into the invoke tier, the invocation of a huge diversity of artifacts can be supported by the framework. Each invoker implements a predefined interface, which is, for example, used by the invocation request handler through the invoker API. An invoker itself utilizes the resource access API to access different kinds of resources, which are located in diverse environments such as an on-premise data center or a Cloud environment. Behind the resource access API, several pluggable access modules such as SSH access can provide a broad variety of access capabilities for different kinds of resources such as virtual machines (VMs) and containers. Similar to invokers, each access module implements a predefined interface, which is exposed to invokers through the resource access API.

7.2.2 Example: Invocation of Chef Cookbooks from BPEL Plans

A major goal of the previously presented PlugInvoke framework is to avoid polluting the orchestration layer. Consequently, the framework helps to avoid putting lower-level technical details into overarching deployment plans. Figure 7.7 outlines a concrete example, how a BPEL-based deployment plan utilizes the invocation & management API. In case of using BPEL plans, it is assumed that the invocation & management API is provided as WSDL/SOAP-based Web service because BPEL is specialized to deal with this kind of APIs. The ‘install Apache’ activity in the deployment plan technically uses the Chef cookbook apache2[1]. However, the plan only refers to the cookbook and defines the invocation parameters. The cookbook itself can either be

referenced by URL, for example, pointing to the Chef Supermarket\(^1\) or in case of TOSCA it can be bundled inside a self-contained CSAR. In the first case, the PLUGINSVOKE framework is in charge of retrieving the artifact from the given URL; in the second case, the TOSCA engine would initially load the artifacts of a particular CSAR into the framework’s artifact database, so the corresponding artifact such as a cookbook is available to the framework on its invocation. Lower-level technical details such as Chef-specific invocation mechanisms are not part of the orchestration layer. A BPEL plan contains activities such as the ‘install Apache’ activity to invoke Web services. In this case, the invocation & management API is the Web service to be invoked. For example, an invocation request sent to the invocation & management API to run the Chef cookbook on a specific virtual machine through SSH is processed as follows: the invocation request handler receives the request and utilizes the invoker API to trigger the Chef invoker. Then, the Chef invoker uses the resource access API, which utilizes the SSH access module for this

\(^1\)http://supermarket.chef.io
request to connect to the corresponding machine. This SSH connection is used to place the Chef cookbook on the machine, install the Chef runtime, and finally execute the Chef cookbook with the parameters given in the request. After finishing the invocation successfully or in case of an error, the Chef invoker informs the invocation request handler through the invoker API. Based on this feedback, the request handler finally responds to the original invocation request.

This example shows how the PlugInvoke framework addresses the previously identified challenges, namely complementing the TOSCAfy approach (Section 7.1) with runtime support and enabling the orchestration layer entirely focus on the composition of proper APIs abstracting from lower-level artifact-specific technical details. However, the presented approach still has two major issues: (i) the orchestration layer such as a deployment plan inherently depends on the framework. In the context of TOSCA, this means an inherent dependency of CSARs to the PlugInvoke framework, so CSARs are not self-contained anymore. The portability of CSARs is also constrained because it is assumed that each TOSCA engine must provide the framework and its external-facing API to run corresponding deployment plans. (ii) Moreover, the external-facing uniform interface cannot be ideal for all orchestration techniques and scenarios. While the previously mentioned WSDL/SOAP-based Web service API may be ideal for BPEL-based deployment plans, an HTTP-based RESTful API may be the better option when utilizing scripting languages such as Python or Ruby for orchestration purposes. Essentially, there is not the optimal ‘one size fits all’ kind of API because it highly depends on the concrete use case and the utilized orchestration techniques, in particular, which kind of API is the best choice in a certain context. These open issues are addressed by the automated APIfication approach discussed in the following Section 7.3.
7.3 Any2API Framework – Automated APIfication

As discussed previously, the unified modeling representation of diverse artifacts (Section 7.1) does not significantly help to streamline their invocation and integration at runtime. Overarching orchestration mechanisms such as deployment plans have to deal with lots of lower-level technical details. The PlugInvoke framework presented in Section 7.2 addresses these challenges by making an extensible ecosystem of specialized runtime modules available through a uniform interface. While this interface helps to keep the orchestration layer such as a deployment plan focused on the flow of activities, there is an inherent dependency on the framework and its interface. This impacts portability because such a deployment plan assumes the framework with a certain set of modules to be available at runtime. Consequently, a specific engine such as a TOSCA engine must provide the invocation framework and its interface to successfully run the plan. Another limitation of the PlugInvoke framework is its uniform interface. While the interface is a powerful means in the first place to abstract from lower-level and artifact-specific technical details, the specific kind of interface may not be ideal for all orchestration techniques and scenarios. For example, exposing a SOAP API as interface makes a lot of sense if BPEL workflows are used for orchestration. However, when using Web technologies or scripting languages for orchestration, a REST API may the better choice. Therefore, choosing the ‘best’ kind of interface highly depends on the utilized orchestration approach.

7.3.1 Framework Architecture

To address these issues, an automated APIfication framework is presented in this section to provide an alternative approach to the PlugInvoke framework. The core idea of the framework is to generate and package self-contained API implementations for any given executable artifact, i.e., the APIfication of artifacts. Therefore, the framework is named Any2API. The generating and packaging process is highly modular and customizable, so different kinds of APIs can be produced to fit the individual needs of diverse orchestration
Figure 7.8: Any2API framework architecture

Techniques. Figure 7.8 outlines the framework architecture and how the different parts of the framework are utilized to generate corresponding APIs. Users interact with the framework through the user interface tier, for example, using a command-line interface or GUI. The process of generating APIs is strongly driven by an artifact’s metadata, describing its dependencies, parameters, etc. Thus, the analyze tier allows for providing domain-specific modules to analyze certain kinds of artifacts. For example, a Chef cookbook analyzer scans Chef cookbooks to extract their metadata. Implementing such analyzer modules can be quite straightforward for some kinds of artifacts, which already specify their metadata more or less explicitly; however, it can also get arbitrarily complex, for example, when source code of general-
purpose scripting languages such as Ruby or Python must be analyzed to identify the input parameters of a script. Consequently, it is always a trade-off to decide whether the effort to implement a specialized analyzer module is justified.

Section 7.1.3 discussed by example, how Chef cookbooks are composed of multiple building blocks such as recipes, attributes, and dependencies. These building blocks are systematically analyzed to extract the artifact metadata of a cookbook in an automated manner by the Chef cookbook analyzer.
Listing 7.3: List of files included in Chef cookbook `mysql`

Listing 7.1 outlines the content of the `metadata.rb` file that is included in the Chef cookbook `mysql`. Complementing these metadata, Listing 7.2 shows the `attributes/default.rb` file contained in the same cookbook. Moreover, Listing 7.3 outlines the list of files that are part of the cookbook. These parts of the cookbook are analyzed to generate the artifact metadata shown in Listing 7.4. Initially, general information such as description, license, and dependencies on other cookbooks such as `yum-mysql-community` can be derived from the `metadata.rb` file. The set of supported operating systems of the cookbook (line 9–19) can also be extracted. Next, the cookbook attributes can be derived from the `attributes/default.rb` file, which aims to provide a meaningful default value for each attribute. Cookbook attributes serve as input and configuration parameters when invoking the cookbook. Finally, the list of cookbook files specifically includes the list of recipes (inside the `recipes` sub-directory). This information is utilized to identify the recipes that are provided by the cookbook.

Similarly, automated artifact analyzers can be implemented for other kinds of artifacts. Listing 7.5 outlines the content of a Dockerfile that installs and runs a PHP runtime environment in conjunction with an Apache Web server inside a Docker container. The first line of the Dockerfile includes the `FROM`
Listing 7.4: Artifact metadata extracted by the Chef cookbook analyzer
statement, pointing to the base container. This is the initial dependency of the Dockerfile. Furthermore, each ENV statement inside the Dockerfile defines an environment variable, which serves as configuration parameter when building and running the container. The EXPOSE statement defines on which TCP ports the container listens at runtime. By analyzing this information in an automated manner, corresponding artifact metadata can be extracted from any Dockerfile.

In any case, experts are always capable of manually specifying an artifact's metadata if the effort for developing a corresponding analyzer does not make sense in a certain context. Hybrid approaches can be followed, too. For instance, a simplified analyzer module can be implemented to extract a significant portion of an artifact's metadata, which is then refined and completed manually by an expert.

Independently of the applied approach to make an artifact's metadata available, the generate tier requires the metadata to generate a proper API for the underlying artifact. Each generator module is highly specialized
on producing a certain kind of API. For example, the SOAP API generator wraps a given artifact and makes it usable through a SOAP-based Web service API. Based on the kind of artifact, the generator fetches the corresponding invoker module from the invoker module registry. In case of a Chef cookbook, for example, the Chef invoker is fetched and bundled with the generated API. Similarly, access modules are fetched from the access module registry and wired with the invoker and the generated API. These access modules enable the remote execution of the artifact. For instance, the Chef invoker may utilize the SSH access module to run the wrapped Chef cookbook on a virtual machine through SSH. However, the entire complexity of managing SSH connections, copying and collecting files, etc. is hidden behind the generated higher-level API. Access and invoker modules are very similar to the access and invoker modules presented in the context of the PlugInvoke framework (Section 7.2). However, they are not invoked in the scope of the Any2API framework itself but they are packaged and executed as part of the generated APIs. Consequently, access and invoker modules can be created in a way to be utilized by both frameworks.

To achieve a higher degree of portability, the package tier provides diverse modules to bundle the generated APIs together with the actual artifact, as well as the invoker and access modules in a self-contained manner. Each packager module is specialized to provide a specific packaging mechanism. For example, the Docker container packager provides the mechanisms to bundle the entire API implementation as a self-contained Docker container, whereas the Vagrant VM packager does the same for Vagrant-based virtual machines. Finally, the resulting generated and packaged API implementation can be instantiated to deploy and manage various kinds of resources in diverse environments such as a local development environment, an on-premise data center, or a Cloud-based environment. The Any2API framework itself is not required at runtime anymore, but only the API implementations that it produces. These generated API implementations can also be described as tailored, minimal, and embeddable invocation frameworks for a particular artifact: tailored and minimal because it only contains the individual logic required to run and expose the functionality of a specific artifact; embed-
because of the self-contained bundling of the required logic. Due to their portability and self-contained packaging, API implementations can be embedded into higher-level bundles such as a Cloud Service Archive (CSAR) in the context of TOSCA; in case of TOSCA each API implementation is packaged as implementation artifact (IA) and attached to a CSAR as discussed in Section 7.1.1. This perfectly complements the previously presented TOSCAfy approach (Section 7.1) and enables the creation of self-contained CSARs, so the TOSCA engine does not have to provide complex processing logic for a plethora of lower-level artifacts. Moreover, the kind of generated APIs is highly flexible, meaning different kinds of APIs (REST, SOAP, etc.) can be generated for certain artifacts to fit the needs of diverse orchestration techniques.

Although the Any2API framework focuses on exposing the functionality of certain artifacts through APIs, artifacts are not inherently required to generate useful API implementations. This applies in case the invoker itself provides capabilities to be utilized and served through generated APIs. For example, an Amazon EC2 invoker may wrap Amazon’s Web service API to manage virtual servers in the Cloud. The invoker's functionality itself can be provided by generating different kinds of APIs. Consequently, another usage scenario of the Any2API framework is to wrap existing APIs and make them available as another kind of API.

7.3.2 Example: Generated SOAP APIs for Chef Cookbooks

In order to show how generated and packaged APIs can be used, Figure 7.9 outlines the concrete example of a BPEL-based deployment plan invoking generated SOAP APIs. BPEL is specialized on the orchestration of Web services that are specified using WSDL. Thus, the Any2API framework is initially utilized (at buildtime) to generate SOAP APIs for artifacts such as Chef cookbooks that are supposed to be invoked by the overarching deployment plan. One activity of the plan installs the Apache HTTP server. Therefore, the Chef cookbook analyzer scans the attributes file of the Apache HTTP server

| Streamlined Orchestration of Diverse Solutions | 154 | 7 |
Figure 7.9: Invocation of generated APIs from BPEL plan

cookbook\textsuperscript{1} to derive the artifact metadata such as input parameters. These metadata are then used by the SOAP API generator and the Docker container packager to produce a bundled and containerized API implementation, including the cookbook itself and required runtime modules, i.e., the Chef invoker and the SSH access modules. The Any2API framework finishes with completing the process of generating and packaging the APIs. Consequently, the Any2API framework is not required at runtime because the generated APIs are self-contained. This is a major difference compared to the previously discussed example for the PlugInvoke framework in Section 7.2.2 because the PlugInvoke framework is inherently required at runtime, while the Any2API framework itself does not appear at runtime at all. The actual invocation of the generated SOAP Web service APIs at runtime happens through a BPEL engine, which is in charge of executing the plan. Web service invocations are triggered by activities in the plan such as the ‘install

\textsuperscript{1}http://gist.github.com/jojow/8e6ba9065522e2906377412bb1d1a5c7
Apache' activity. The packaged API implementation with its invoker and access module performs the execution of the corresponding artifact, including all required preparations: setting parameters, collecting results by accessing remote resources such as virtual machines, containers, etc.

7.4 Chapter Summary and Discussion

TOSCAfy as standards-driven transformation framework (Section 7.1) was the initial step toward streamlining the orchestration of diverse executable artifacts in order to homogenize their representation. At the core of the transformation framework, the Topology and Orchestration Specification for Cloud Applications (TOSCA) was utilized as standardized modeling representation for different kinds of artifacts. The extensible architecture of the framework enables the creation of a rich ecosystem of specialized modules, which implement domain-specific transformation mechanisms. Implementing modules to be used in conjunction with the TOSCAfy framework requires development effort. Consequently, the question appears whether it is worth the effort to implement a particular module in contrast to manually transforming certain artifacts toward TOSCA. Since the generic modules such as the HTTP retriever, the TOSCA XML serializer, and the CSAR packager are reusable for doing the automated transformation of various kinds of artifacts, the occurring development effort can be justified more quickly than for domain-specific modules. For these kinds of modules such as a Chef-specific transformer and dependency resolver, it depends on various aspects whether it is worth to spend costs and effort to implement domain-specific modules. If a large number of reusable artifacts of a particular kind is provided, which are also continuously updated, it may be a strong indicator that it makes sense to create modules for their automated transformation toward TOSCA. Prominent examples for sources of large and public collections of certain kinds of artifacts are Docker Hub\(^1\) and Chef Supermarket\(^2\). In addition to such publicly shared collections, certain organizations may maintain reposi-

---
\(^1\)http://hub.docker.com
\(^2\)http://supermarket.chef.io
tories, which hold internally reusable deployment scripts that are created and annotated by following well-defined conventions and templates. If the number of these artifacts is large enough, it may also be worth the effort to implement specific transformer modules instead of manually transforming them to TOSCA artifacts.

A uniform modeling representation such as the previously discussed TOSCA-based transformation approach does not address several challenges that appear at runtime. For example, the orchestration layer (e.g., an overarching deployment plan) gets polluted and bloated by lower-level technical details such as diverse invocation mechanisms, parameter passing, collecting results, etc. Consequently, subsequent steps toward streamlining the orchestration of different kinds of artifacts are required to explicitly target runtime aspects. Therefore, the PlugInvoke framework was presented in Section 7.2 as a complement to provide a uniform interface in the form of an invocation & management API. This API hides much of the lower-level technical complexity by utilizing a plethora of specialized modules underneath as part of the highly extensible PlugInvoke framework. While this approach already provides significant improvements over putting all the lower-level invocation and management logic in the orchestration layer, there is an inherent runtime dependency on the framework. Consequently, a deployment plan utilizing the interface of the framework can only run on an engine that is somehow connected or bundled with the framework. Moreover, the uniform interface provided by the framework may not be the ideal fit for all kinds of orchestration scenarios and techniques. To address these issues, the Any2API framework was introduced in Section 7.3 to provide an automated APIfication approach. This framework provides the capabilities to generate self-contained API implementations for different kinds of executable artifacts, i.e., enabling their APIfication. Its extensible architecture enables the creation and usage of specialized modules to generate different types of APIs (REST, SOAP, etc.) for diverse artifacts. Therefore, the kind of API to be generated can be chosen to fit the utilized orchestration technique: while BPEL-based deployment plans can perfectly orchestrate SOAP APIs, REST APIs may be the better choice when using other orchestration mechanisms.
The Any2API framework can be used alternatively to the PlugInvoke framework, although both frameworks can technically share certain parts such as the invoker and access modules. A comparison of the two frameworks makes clear that Any2API is heavily buildtime-oriented to completely eliminate any runtime dependency to the framework itself. This is achieved by generating self-contained APIs and packaging them with the corresponding artifacts. By bundling these generated APIs together with orchestration logic such as a deployment plan, fully self-contained TOSCA CSARs can be produced. In this case neither the TOSCA engine nor the orchestration layer gets bloated because the lower-level technical details are hidden as part of the generated API implementations. However, this approach has its costs: (i) significant efforts appear at buildtime because APIs must be generated and tested for all artifacts that are potentially invoked at runtime. Although most of these activities can be fully automated, at least it takes additional time and requires additional processing steps to be performed. (ii) Moreover, the resulting bundles such as CSARs are larger because not only the plans, topology models, and artifacts are bundled, but also the generated API implementations.

The decision whether to prefer Any2API or the PlugInvoke framework highly depends on the concrete usage scenario. If, for example, only a fixed set of artifact types are used in a certain context and CSARs are only used in a limited scope such as a single small-scale organization in conjunction with a fixed orchestration technique, it may be perfectly fine to rely on the PlugInvoke framework at runtime. This helps to keep CSARs small and to avoid additional buildtime costs. However, in case of a more dynamic scenario, for example, when using or switching between different orchestration mechanisms, or frequently using new kinds of artifacts, Any2API would be the preferable option because it provides more flexibility but still reduces runtime complexity. Another option is to combine both frameworks: the PlugInvoke framework may cover a small and relatively fixed set of artifact types that are often used in a certain context. Any2API is only utilized to wrap more exotic artifacts that are not covered by the PlugInvoke framework. For example, if the usage of new kinds of artifacts is evaluated,
the PlugInvoke framework itself does not have to be extended but APIs for these new artifact types can be generated at buildtime. This helps to keep most CSARs small but still gaining from the benefits of the Any2API framework.
The previous chapters presented several concepts and approaches, which help to achieve the major goal of this work: enabling continuous delivery based on gathered solutions. This goal was initially expressed as part of the Gather’n’Deliver method in Chapter 3. The presented approaches provide key building blocks to implement this method. However, these building blocks must be connected and integrated to effectively support the Gather’n’Deliver method. Therefore, an integrated architecture overview is given in the following. Furthermore, several prototype implementations that have been developed to show the feasibility of the presented approaches are described in this chapter. Finally, the presented approaches are evaluated based on multiple validation scenarios. In addition to the scenarios that are positioned in the scope of continuous delivery, further validation scenarios are presented beyond the scope continuous delivery. This is to emphasize that some of the presented approaches are not domain-specific to continuous delivery, but can be applied and used in other domains, too.
8.1 Architecture Overview

Figure 8.1 provides an overview of the integrated architecture [WBKL16], which outlines the interconnections between the previously presented building blocks. The main purpose of the integrated architecture is to provide the technical foundation of the Gather’n’Deliver method. The upper part outlines the collaborative and automated gathering of solutions as discussed in Chapter 6. While the collaborative part is enabled by solution repositories (Section 6.1), the automated discovery and capturing of existing solutions is enabled by the auto-gather pipeline (Section 6.2). Solution repositories serve as input for the deliver-knowledge-base pipeline (Section 6.3), which produces a knowledge base variant that provides fine-grained query mechanisms to find appropriate solutions for a specific application environment. TOSCAfy as standards-driven transformation framework (Section 7.1) represents the initial building block toward streamlining the orchestration of diverse solutions as discussed in Chapter 7. A set of potentially diverse solutions such as executable artifacts is exported from the knowledge base to act as input for the TOSCAfy framework. The transformation framework utilizes TOSCA as standardized modeling representation of different kinds of artifacts: TOSCA artifacts such as node types are produced and packaged as Cloud Service Archives (CSARs) to be stored in a CSAR repository. Such a repository could be, for example, provided by Winery [KBBL13]. The solutions wrapped as TOSCA artifacts serve as input for the utilized build environment for application topologies, which could be provided by Winery as well. As discussed in Section 7.3, the Any2API framework can be used at buildtime to generate self-contained APIs, which help to simplify the artifacts’ orchestration at runtime. A build environment for continuous delivery pipelines such as GitLab CI¹ or Jenkins Pipeline² is then used to create a pipeline based on different stages and application environments by utilizing and wiring the previously built application topologies. Moving from buildtime to runtime, a runtime environment for continuous delivery pipelines that could be, for

¹ [http://about.gitlab.com/gitlab-ci](http://about.gitlab.com/gitlab-ci)
Figure 8.1: Integrated architecture overview
example, also provided by GitLab CI or Jenkins Pipeline, enables instances of the pipeline to be created. The pipeline’s underlying application environments are managed by a runtime environment for application topologies e.g., OpenTOSCA [BBH+13]. This runtime environment can be paired with the \texttt{PlugInvoke} framework (Section 7.2) to enable the execution of different kinds of executable artifacts such as deployment scripts that are attached to application topologies to create instances of those.

8.2 Prototype Implementations

The following sections present several prototype implementations, covering all parts of the integrated architecture that have been newly added to it in the context of this work. These prototypes have two major purposes, namely (i) to show that the proposed approaches are technically feasible, and (ii) to enable their evaluation based on the validation scenarios described afterward.

8.2.1 GatherBase & DevOpsBase Prototype

Following the GatherBase architecture presented in Section 6.4 a prototype\footnote{http://github.com/gatherbase/gatherbase} has been implemented as shown in Figure 8.2 in order to provide a crawler to discover and capture existing solutions. The upper part covers the \textit{auto-gather pipeline}, which is technically based on the Kue job processing framework\footnote{http://automattic.github.io/kue} running inside a Node.js runtime environment\footnote{http://nodejs.org}. Both job queues are managed by a Redis server\footnote{http://redis.io}, which is natively integrated with the Kue framework. Since the Kue framework provides a layer of abstraction to the underlying queueing mechanism, Redis could be replaced by another queueing system. The domain-specific job processing modules such as the Chef cookbook discoverer and the HTTP retriever are plugged into the Kue framework and wired using the job queues. The writer module exports the
Figure 8.2: GATHERBASE & DEVOPSBASE prototype
retrieved and processed solutions and pushes them to a solution repository. Solution repositories as part of this prototype implementation are Git repositories, which are hosted on GitHub\(^1\).

The \textit{deliver-knowledge-base} pipeline runs in a separate Node.js environment and consumes solutions from the previously described Git repositories. As discussed in Section \textbf{6.4}, diverse flavors of the \textit{deliver-knowledge-base} pipeline can be implemented based on a set of reusable and pluggable modules. The pipeline outlined in Figure \textbf{8.2} as part of the prototype implementation is based on a subset of these previously discussed modules. This particular pipeline produces a specific knowledge base variant\(^2\) which is called \textsc{DevOpsBase} because the knowledge base provides diverse solutions to implement DevOps-oriented processes such as continuous delivery pipelines. Technically, the \textit{DevOpsBase builder} creates \textsc{DevOpsBase} as a knowledge base variant (KB variant) that consists of an Elasticsearch back-end\(^3\) and

---

\(1\) \url{http://github.com}
\(2\) \url{http://github.com/devopsbase/devopsbase}
\(3\) \url{http://www.elastic.co/products/elasticsearch}
a Web-based graphical user interface. The KB variant is packaged as a Docker container, so it can run on different infrastructures such as a local developer machine or a Cloud environment. DEVOPSBASE can either be queried by users through the Web UI or by utilizing the Elasticsearch query DSL \[^1\], a JSON-based query language. The conceptual foundation for such queries are AER predicates (Definition 4.6) as discussed in the context of the application environment metamodel (Chapter 4) to express application environment requirements (AERs). Let \( l_{mysql} = (MySQL, l_{parent}) \) be a label and \( s_{my} = (u_{s_{my}}, L_{s_{my}}, B_{s_{my}}, prop_{s_{my}}) \) a solution. The following AER predicate can be defined to retrieve solutions from the knowledge base that provide an implementation of MySQL version 5.7:

\[
r^{\text{inclusive}, l_{mysql}}(s) = \begin{cases} 
    \text{true} & \text{if } \text{prop}_{s}(\text{version}) = 5.7, \\
    \text{false} & \text{otherwise.}
\end{cases}
\]

This AER predicate can be translated into an Elasticsearch query as outlined in Listing 8.1. A logical expression consisting of multiple AER predicates can also be translated into such queries, for example, using Boolean query compositions, which is a specific feature of the Elasticsearch query DSL. The evaluation of AER predicates corresponds to the execution of the translated queries. The result are sets of solutions as outlined by Algorithm 5.1 in Section 5.3. Retrieved solutions may themselves have dependencies that must be resolved. Therefore, the procedure sketched by Algorithm 5.2 can be applied as discussed in Section 5.3.2 to derive additional queries. These queries are executed to identify further solutions in order to satisfy existing unresolved requirements. A selected set of retrieved solutions can then be transformed toward a unified modeling representation by using the TOSCAfy prototype as described in the following.
8.2.2 TOSCAfy Prototype

Following the architecture of TOSCAfy as standards-driven transformation framework presented in Section 7.1.2, the prototype outlined in Figure 8.3 implements its core functionality. Technically, the TOSCAfy prototype is implemented as a set of modules, running in a Node.js runtime environment. Input to the framework can be provided through different channels such as a file fetched from an HTTP endpoint using the HTTP retriever module or reading a file from the filesystem. In terms of transformation, the PluginInvoke prototype covers two different kinds of artifacts: Chef cookbooks and Docker containers. Domain-specific transformer modules are implemented for this purpose. In addition, the Chef dependency resolver module covers the resolution and retrieval of Chef cookbook dependencies. The transformer modules utilize the TOSCA metamodel to wrap incoming artifacts as TOSCA node types. TOSCA’s XML schema definition [OAS13b] is the foundation of the TOSCA XML serializer module to render the node types in XML and

---

1[^1]


finally package them together with the original artifacts as Cloud Service Archives (CSARs) using the CSAR packager. The resulting CSARs are stored in a CSAR repository, which is implemented as a Git repository.

8.2.3 PlugInvoke Prototype

Following the PlugInvoke architecture presented in Section 7.2.1, its prototype implementation is outlined in Figure 8.4. The framework’s core runs as a set of modules in a Node.js runtime environment. All state information and persistent data are stored in databases: the artifact database and the instance database on a MongoDB server\(^1\). The request handler modules are implemented to serve incoming requests from the invocation & management API, which is provided as HTTP-based RESTful API. Three invoker modules

---

\(^1\) [http://www.mongodb.com](http://www.mongodb.com)
are implemented as part of the framework to cover the invocation of Chef cookbooks, Python scripts, and Unix shell scripts. Moreover, three access modules are available to run an invocation in local and remote environments. The *local access* module performs an invocation locally in the same environment in which the framework is running. Both *SSH access* modules (for Unix and Windows) enable remote invocations as long as the target environment such as Cloud-based virtual machines can be accessed through SSH. The PlugInvoke framework is integrated with the OpenTOSCA engine [BBH+13], which provides a Java OSGi runtime environment\(^1\). The *operation invoker* [WBB+14b] is one of several components that run in this OSGi environment and compose the functionality of the OpenTOSCA engine. Its main goal is to provide a unified interface to deployment and management plans to simplify the invocation of operations that are attached to node types and relationship types used in application topologies. Consequently, plans do not have to consider how to invoke implementation artifacts that implement certain operations, so they are not polluted with lower-level technology-specific details and can focus on the orchestration of operations. By utilizing the framework’s REST API, OpenTOSCA’s *operation invoker* transparently hooks the functionality of the framework into the OpenTOSCA engine. Technically, a plan invokes an operation through the *operation invoker* without considering any technical details of the underlying implementation artifacts. The *operation invoker* then collects the required implementation artifacts to use them in conjunction with the original invocation parameters from the plan for making a request against the framework’s REST API. The PlugInvoke framework performs the invocation and reports the results back to the *operation invoker*, which itself responds to the plan that originally invoked the operation. All interactions between plans, operation invoker, and the PlugInvoke framework are asynchronous to be robust against connection timeouts and other potential issues of synchronous communication. To sum up, the presented approach enables separation of concerns, so that (i) deployment and management plans focus on the logic of

\(^1\)http://www.osgi.org
orchestrating operations properly, (ii) CSARs bundle artifacts and plans only related to the application it specifies, and (iii) different kinds of artifacts can be used and combined to implement a particular operation without changing higher-level plans.

### 8.2.4 Any2API Prototype

The prototype implementation of the Any2API framework is outlined in Figure 8.5 and follows the originally described Any2API architecture that was presented in Section 7.3.1. More specifically, the modules depicted in Figure 8.5 are implemented as part of the prototype. A basic user interface is provided by the command-line interface (CLI) to serve the functionality of the underlying set of modules to generate self-contained API implementations for existing executable artifacts such as deployment scripts and

---

[Figure 8.5: Any2API prototype]

---

configuration definitions. The CLI, as well as the analyzer, generator, and packager modules are running in a Node.js runtime environment. These parts of the framework are used at buildtime to generate and package corresponding APIs. As depicted in Figure 8.5, the prototype can generate REST APIs and SOAP APIs. Each generated API implementation includes a selection of reusable runtime modules, which encapsulate specialized access and invocation logic for certain artifacts and resources. For example, the SSH access module provides access functionality (copying files, executing commands, setting environment variables, etc.) for resources such as virtual servers that are accessible through an SSH interface; the Chef invoker module encapsulates the logic to run Chef cookbooks and can be paired with any access module to perform the invocation. These modules are referred to as packageable modules because they are packaged together with generated API implementations to do their work at runtime. As part of the prototype these modules are technically provided as NPM modules through an NPM registry. To sum up, the Any2API prototype can be used to perform the process of generating self-contained REST or SOAP APIs and orchestrating them at runtime as discussed in Section 7.3.

8.2.5 Integration with OpenTOSCA Toolchain

Section 8.2.3 already outlined the integration of the PlugInvoke framework with the OpenTOSCA engine. This section delves into this integration approach by showing how the other prototypes presented in this work can be combined with the OpenTOSCA toolchain [WBB+14b; WBKL16]. A higher-level overview of the integration is shown in Figure 8.6: the upper part (first row including the TOSCAfy framework) summarizes the previously discussed toolchain that has been created as part of this work. TOSCA modeling artifacts are produced by the TOSCAfy prototype to be used as input for Winery [KBBL13], which is the modeling tooling of the OpenTOSCA toolchain. Alternatively, Any2API can be utilized in between to wrap the functionality of an executable artifact as API to ease the orchestration at run-
time. For example, in the scope of the OpenTOSCA toolchain, BPEL [OAS07] is typically used to create deployment and management plans. Consequently, Any2API is used in this setting to generate SOAP APIs for executable artifacts to simplify their orchestration using BPEL. Different application environments can then be modeled as application environment topologies using Winery. The previously produced TOSCA modeling artifacts can be combined with individually developed ones to create corresponding topology models. Instances of an application topology can be deployed and managed using the OpenTOSCA engine. Furthermore, a set of topologies may represent the application environments (development, test, production, etc.) that form the foundation of the different stages of a particular continuous delivery pipeline. In this case a continuous delivery tooling such as GitLab CI[1] is used in between to build and run the pipeline. Instead of directly using the OpenTOSCA engine to create instances of application topologies, GitLab CI uses the OpenTOSCA engine to manage these instances to provide the

application environments required by a particular pipeline. Regardless of whether the OpenTOSCA engine is used directly or through a continuous delivery tooling such as GitLab CI, the PlugInvoke prototype is integrated with the engine as discussed in Section 8.2.3. This enables OpenTOSCA to run a broad variety of executable artifacts, which are attached to application topology models to automate the deployment and management of corresponding application instances and application environments.

8.3 Validation in the Scope of Continuous Delivery

The previously discussed prototype implementations (Section 8.2) including GatherBase, DevOpsBase, TOSCAfy, PlugInvoke, ANY2API, and the OpenTOSCA integration provide an initial validation of the concepts and frameworks that were presented as part of this work. These prototypes are the foundation for further validation scenarios to confirm the actual applicability of the different approaches. Therefore, the prototypes themselves are the initial validation beside the validation scenarios that are described in the following sections. A specific goal of this work is to support the implementation of continuous delivery pipelines and their underlying application environments. Therefore, the following sections discuss several validation scenarios in the scope of continuous delivery to evaluate the presented approaches and prototypes.

8.3.1 Application Environments for WordPress Delivery Pipeline

According to various statistics such as W3Techs[1] or BuiltWith[2] WordPress[3] is not only the most widely used Web application to provide blogs, but it also serves as content management system (CMS) for many popular websites[4]. Therefore, the current stable version 4.5 of WordPress was chosen to be the foundation of the initial validation scenario of this work to build

---

1. http://w3techs.com/technologies/overview/content_management/all
Figure 8.7: WordPress’ application environment requirements
a continuous delivery pipeline. The upper part of Figure 8.7 outlines the application environment requirements (AERs) for WordPress that must be satisfied to provide the technical foundation for the pipeline stages. As discussed in Section 8.2.5, the integrated toolchain can be utilized to build a corresponding pipeline. Initially, the DevOpsBase prototype (Section 8.2.1) is used to find and fetch appropriate solutions from the knowledge base to satisfy these requirements. Previously fetched solutions, for example, in the form of executable artifacts can then be transformed into TOSCA modeling artifacts (Section 8.2.2) to be used as building blocks for creating application topologies using Winery [KBBL13]. These topology models specify the different application environments. Next, GitLab CI can be used as a continuous delivery pipeline tooling to create the pipeline and wire the underlying application environments based on the topology models. This is technically enabled by the OpenTOSCA engine [BBH+13] in conjunction with the Plug-Invoke framework (Section 8.2.3), taking care of maintaining application topology instances. Optionally, Any2API (Section 8.2.4) is used before in order to generate APIs for diverse executable artifacts such as deployment scripts. This makes application topology models more self-contained and simplifies the creation of orchestration logic such as deployment plans as discussed previously.

The lower part of Figure 8.7 shows two topologies: the one of the JavaScript test environment and another one of the production environment of WordPress. These topologies are significantly different due to the different application environment requirements. Moreover, selecting the ‘best’ solutions depends on certain environment-specific constraints, too. For example, a development environment typically aims to be lightweight by running a minimal set of components in containers, whereas a production environment must be highly available by using cluster-based infrastructures. On the other hand, the JavaScript test environment can be hosted on a single virtual machine without requiring an entire cluster. The two outlined topology models show how different kinds of solutions can be combined to maintain corresponding application environments. More specifically, the Chef cook-
book nodejs is used to install Node.js and NPM as part of the JavaScript test environment; NPM is then utilized to install Grunt, which is also required in the JavaScript test environment. The WordPress application and test files are placed inside the JavaScript test environment using an application-specific Shell script. On the production side, the Chef cookbook kubernetes is used to maintain a Kubernetes cluster to run php/apache Docker containers on top of it. The WordPress database is hosted on a MySQL cluster, which is managed through the Juju charm mysql.

The previously listed solutions resolve the corresponding AERs. However, they only represent a certain selection of solutions fetched from the knowledge base. Consequently, there is a multitude of alternative solution sets available. For example, the Node.js environment required at JavaScript test stage can be provided by a Docker container. The PHP runtime and Apache Web server can be provided by a corresponding Chef cookbook and Juju charm. Further alternatives include the use of scalable and elastic platform-as-a-service offerings such as Google App Engine, Amazon Elastic Beanstalk, and Google Cloud SQL instead of managing clusters running on basic infrastructure offerings such as Amazon Elastic Compute Cloud as shown in Figure 8.7. The other application environments (build, PHP test, etc.) can be implemented by selecting and fetching further building blocks from the knowledge base such as the Chef cookbooks phpunit and subversion. Depending on different additional constraints (tooling exper-

---

1. http://supermarket.chef.io/cookbooks/nodejs
tise, community ratings, etc.), certain solutions may be preferred to others to serve as building blocks of specific application environments.

For the sake of completeness, it is worth mentioning that some of the requirements outlined in Figure 8.7 can be substituted by satisfying alternative requirements. For example, instead of using MySQL, the community-driven alternative MariaDB\(^1\) can be used to run WordPress. Similarly, NGINX\(^2\) can substitute the Apache Web server. In terms of optimizing the continuous delivery process, the pipeline could be streamlined by sharing certain application environments across stages. This makes particular sense in case (i) the environments have similar requirements and (ii) they do not have to be strictly isolated from each other. Merging a test environment with a production environment is typically not recommended because tests may significantly influence the production instance of the application. However, running the build stage and the JavaScript test stage of the WordPress pipeline in the same environment would make a lot of sense because their AERs are very similar and the production environment is not affected. Using a single application environment for running the build and test stage of a pipeline is common practice and therefore implemented by popular continuous delivery tools and services such as Travis CI\(^3\).

8.3.2 Delivery Pipelines for Netflix RSS Reader Recipes Application

The Netflix company is known for its proactive open-source strategy: they constantly share most of their code publicly as open-source libraries and tools\(^4\). To show how a selection of these open-source components can be used and combined, they published the RSS reader recipes application\(^5\) (in short RSS application) as a simple reference application. It consists of three parts: (i) the rss-edge component that provides the user-facing edge service, including an RSS reader UI; (ii) the rss-middletier component is implemented

\(^1\) http://mariadb.org
\(^2\) http://nginx.org
\(^3\) http://travis-ci.org
\(^4\) http://netflix.github.io
\(^5\) http://github.com/Netflix/recipes-rss
as an internal service, providing the application logic for fetching RSS feeds and persisting user subscriptions. Apache Cassandra\(^1\) serves as persistent storage for the application and is connected to the `rss-middletier` service. (iii) The `rss-core` provides shared Java classes used by both `rss-edge` and `rss-middletier`.

In the context of the DevOps Performance Working Group\(^2\), which is part of the Standard Performance Evaluation Corporation (SPEC), efforts are made to establish a reusable continuous delivery pipeline for a specific sample application. Such a pipeline can then be used by the working group, for example, to conduct different kinds of performance measurements in the context of DevOps-oriented software delivery processes. The previously described Netflix RSS application is one candidate to be used as foundation for creating a corresponding pipeline. Figure 8.8 outlines the pipeline stages that are derived from the application architecture, its build and test procedure, and the usage of containers as a portable packaging mechanism. Container virtualization solutions such as Docker aim to simplify the deployment by running and reusing portable containers in diverse environments (integration test environment, staging environment, production environment, etc.). Moreover, reproducibility is improved because the very same

---

1. [http://cassandra.apache.org](http://cassandra.apache.org)
2. [http://research.spec.org/working-groups](http://research.spec.org/working-groups)
containers are reused to create application instances. The deployment itself is not part of the pipeline as shown in Figure 8.8 because the published containers are the resulting artifacts to be deployed in diverse environments. This helps to make the pipeline reusable by keeping it independent from concrete deployment targets that are specific to certain usage scenarios and research groups. For example, Kubernetes recipes\(^1\) exist to be used to individually extend the pipeline by adding a corresponding deployment stage.

The simple reuse of the pipeline for different scenarios is one of its most important goals. Therefore, the DevOpsBase prototype (Section 8.2.1) is used to identify fully integrated continuous delivery solutions that cover all application environment requirements instead of searching for individual building blocks for each application environment. Maintaining a pipeline using such an integrated solution is typically easier because less additional tooling such as OpenTOSCA or Any2API is required to maintain the underlying application environments. However, migrating a pipeline to another continuous delivery solution is usually harder because application environments are not specified and bundled in a portable manner, but rely on tooling-specific APIs. In case of the Netflix RSS application the required tooling for the different stages is outlined in Figure 8.8: Git, Java, Gradle, Docker, and Docker Compose. These requirements are used to query the knowledge base for integrated continuous delivery solutions. As a result, two alternative pipelines\(^2\) were implemented. The initial setup is based on Travis CI\(^3\), a continuous delivery service that seamlessly integrates with GitHub\(^4\) and Docker Hub\(^5\). Each commit to the GitHub code repository triggers a new run of the pipeline, which eventually pushes updated container images to Docker Hub. Since this setup satisfies all requirements, all stages of the pipeline are covered. Another option is to replace Travis CI with Snap CI\(^6\). While Travis

---

5. http://hub.docker.com
Figure 8.9: Pipeline stages requirements to deploy crash simulation APIs

CI uses a shared environment across all stages, Snap CI provides an isolated environment per stage. This helps to better isolate the different stages, but tends to make the pipeline slower because a separate environment must be established when entering a new stage. Moreover, the Snap CI-based pipeline is configured through a Web UI, whereas Travis CI allows to store a YAML configuration file inside the GitHub repository. This configuration file is automatically read and processed by Travis CI to dynamically build the pipeline. Travis CI and Snap CI are just two options that were used to implement integrated pipelines for the Netflix RSS application. Further alternative solutions include CircleCI\(^1\) and Codeship\(^2\), the latter is additionally integrated with Bitbucket\(^3\) to be used alternatively to GitHub.

8.3.3 Continuous Delivery of Containerized APIs for Matlab Scripts

Continuous delivery pipelines are typically maintained for applications as, for example, discussed previously in Section 8.3.1 and Section 8.3.2. However, pipelines can also be built and utilized to automate the build, test, integra-

\(\text{\footnotesize \textsuperscript{1}}\) http://circleci.com
\(\text{\footnotesize \textsuperscript{2}}\) http://codeship.com
\(\text{\footnotesize \textsuperscript{3}}\) http://bitbucket.org
tion, and deployment process of other kinds of software. As an example, Section 6.3 discussed how continuous delivery of diverse knowledge base variants can be implemented. This section presents another usage scenario, namely a continuous delivery pipeline that produces containerized APIs that provide the functionality of plain Matlab\(^1\) scripts. In the field of e-science, Matlab is used as a tool to implement and run complex scientific calculations and simulations. For instance, Nowakowski [Now15] proposes the use of MatMorembs\(^2\) as “a Matlab-based software package (...) which serves as preprocessor for EMBS [Elastic Multibody Systems] simulations. Apart from import and export functions (...), the main issue is the model order reduction of elastic bodies”\(^3\). An essential use case for this approach are crash simulations that perform destructive crash tests. Corresponding experiments typically aim to investigate the safety of a car and the human beings sitting inside the car during a crash. In order to distribute and parallelize the execution of scientific simulations, Weiß and Karastoyanova [WK16] propose the use of choreographies based on workflow technology. More specifically, BPEL4Chor [DKLW07] is utilized as a language to specify choreography models that can distribute and parallelize certain parts of a simulation. Since these models are not immediately executable, they are transformed into BPEL workflows [OAS07] to be executed on a BPEL engine such as Apache ODE\(^4\). In this context, Any2API is used to generate containerized APIs for existing Matlab scripts; these scripts use MatMoremb to run crash simulations. This is different from the previously described uses of Any2API that were focusing on the APIfication of executables such as deployment scripts that help to build application environments. Therefore, this validation scenario confirms the fact that Any2API is domain-independent and can be applied to very different kinds of executables. Technically, the Any2API framework is used the same way as for generating APIs for deployment executables (Section 7.3 and Section 8.2.4). In addition to the individual Matlab scripts.

\(^1\) http://mathworks.com/products/Matlab
\(^2\) Matlab + Moremb (Model Order Reduction of Elastic Multibody Systems)
\(^3\) http://www.itm.uni-stuttgart.de/research/model_reduction/MOREMBS_en.php
\(^4\) http://ode.apache.org
specific artifact metadata is provided for each script to define its dependencies, input parameters, and produced output. These metadata are in this case processed by the SOAP API generator of the framework to produce a SOAP API that exactly corresponds to the interface of the underlying Matlab script. Generated SOAP APIs can then be combined and invoked through a BPEL4Chor model, which is eventually transformed into an executable BPEL workflow.

*ANY2API* could be used immediately to generate corresponding SOAP APIs based on the artifact metadata and the original scripts. However, the underlying Matlab scripts are continuously changed and refined similarly to an actively developed application or microservice. Therefore, a continuous delivery pipeline helps to automate the process of building, testing, and deploying the generated and containerized APIs. Figure 8.9 outlines the requirements that are relevant for the different pipeline stages. As discussed in Section 8.3.1 and Section 8.3.2 as part of the previous validation scenarios, *DEVOPSBASE* can be used to identify corresponding solutions. For example, the Chef cookbook *docker*[^1] is utilized to run the Docker engine. Subversion is required because the Matlab scripts and their artifact metadata are managed through a version-controlled subversion repository. To enable interactions with the repository such as the checkout performed in the *API build* and *container build* stages, the Docker container *svn-client*[^2] provides a subversion client. At API test stage a Node.js runtime environment is required beside the Matlab environment. This is because the produced API implementations and generated ‘smoke tests’ require Node.js to run. The Matlab dependency at the different stages is a special case in this scenario. For reasons of licensing the utilized Matlab environment is running on a dedicated server. SSH access is provided to this server to run the Matlab scripts. Therefore, the *SSH access* module mentioned in Section 8.2.4 is bundled with the generated APIs to enable remote invocations of Matlab scripts. As a result, the generated and containerized APIs can be hosted on a different server as long as it can reach the Matlab environment via SSH. This is inherently required because the

[^1]: [http://supermarket.chef.io/cookbooks/docker](http://supermarket.chef.io/cookbooks/docker)
corresponding permissions are missing to install and run the containerized APIs directly on the server hosting the Matlab environment.

8.3.4 Tailored Deployment Engines

As outlined previously in Section 8.2.5, general-purpose deployment engines such as OpenTOSCA \([\text{BBH}+13]\) are used to manage diverse application environments as part of continuous delivery pipelines based on application topology models and their required technical artifacts such as deployment scripts and configuration definitions. Alternatively, individually tailored deployment engines \([\text{WBL15c}]\) can be produced, which cover exactly the deployment logic required by a set of related application topologies. These are, for example, topology models specifying application environments that belong to the stages of a particular continuous delivery pipeline. Technically, a tailored deployment engine (in short *tailored engine*) is a *portable and executable package of deployment logic that provides at least one API endpoint to deploy instances of at least one application topology* \([\text{WBL15c}]\).

Figure 8.10 outlines a simplified overview of how tailored engines are built and used. Moreover, it compares the concept of tailored engines with general-purpose engines by example of OpenTOSCA. Application topology models including all required artifacts such as deployment scripts and plans serve as input on both sides. If TOSCA is used, topology models and artifacts can be provided as Cloud Service Archives (CSARs). In case of producing a tailored engine these artifacts are processed by Any2API (Section 7.3 and Section 8.2.4) to generate API implementations and package them as Docker containers to provide the artifacts’ functionality through proper APIs. These containers are then bundled using Docker Compose\(^1\), so the container bundle can be used as a tailored deployment engine. Overarching deployment plans to create instances of certain topologies can either be created manually and bundled with the topology models, or a plan generator \([\text{BBK}+14]\) can be utilized to generate such plans. These plans are artifacts, too; their functionality is provided through corresponding APIs. However, depending on which

---

\(^1\) [http://docs.docker.com/compose](http://docs.docker.com/compose)
Figure 8.10: Tailored vs. general-purpose deployment engines

kind of APIs are generated for the underlying artifacts, the manually created plans may have to adapted to pair them with the generated APIs. Finally, instances of such a tailored engine can be created to manage application environments that are specified by the topology models. The Docker-based portable packaging enables instances of tailored engines to run on different infrastructures such as a developer laptop or a Cloud-based infrastructure.

General-purpose deployment engines such as OpenTOSCA follow a dif-
ferent approach as outlined in a simplified way by the right-hand part of Figure 8.10. Topology models and their artifacts are loaded into the engine, which is typically existing already to deploy and manage multiple applications. In case of OpenTOSCA, a self-contained CSAR is imported into the engine. In contrast to a tailored engine, OpenTOSCA provides general-purpose APIs such as the operation invoker API, which is utilized to invoke artifacts attached to an application topology. The operation invoker utilizes the PlugInvoke framework (Section 7.2 and Section 8.2.3) in the background to carry out the execution of artifacts such as installation scripts that are centrally stored inside the implementation artifact database. Deployment plans running inside the plan engine use the operation invoker through its API to run the required artifacts. The plan engine itself provides an API to trigger and control plan executions. By running plans and their underlying artifacts through OpenTOSCA’s APIs, corresponding application environments can be managed through the engine. Comparing tailored engines with general-purpose deployment engines, several technical implications with certain advantages and disadvantages result from following the one or the other approach [WBL15c]:

- A tailored engine can be minimal by only including deployment executables implementing the deployment actions required by a certain application topology. Consequently, it provides an optimized performance due to minimal resource consumption and minimal setup efforts.

- A tailored engine can be optimized for the deployment of a specific set of application topologies in terms of which kind of API is provided (REST, SOAP/WSDL, JSON-RPC, etc.), how it is packaged (Docker container, VM image, etc.), and further aspects.

- While a general-purpose engine does not provide such a high degree of flexibility and customizability, it provides unified APIs to be used for many different deployment scenarios. This makes it easier for experts to switch between projects and applications instead of getting familiar with individual APIs provided by different tailored engines.
• Tailored engines are independent of each other because they package all required deployment executables for deploying a particular application in a self-contained manner. Consequently, they do not rely on centralized, self-hosted middleware components such as a service bus. As a result, these engines tend to be more robust by avoiding a single point of failure as it would be implied by a centralized middleware component.

• Instances of tailored engines have to be created before they can be used to create application instances. General-purpose engines are typically already available, e.g., as a service, publicly or privately. Consequently, application instances can be created immediately by using the engine without managing the engine itself.

• In case of tailored engines, glue code that is required for exposing the functionality of deployment executables through APIs does not have to be developed and maintained manually, but it is generated at buildtime through the Any2API framework.

• However, this requires specific actions to be carried out at buildtime to generate and bundle APIs that are part of a tailored engine. Moreover, when application topologies or their artifacts are updated, these buildtime actions need to rerun. These additional steps can be skipped completely when using general-purpose engines.

The concept of initially provisioning a deployment engine and then using it to deploy application instances is similar to the “bootware” approach [VKL13] used in the context of modeling and running scientific workflows. It follows a two-step bootstrapping process, which initially provisions a deployment engine. In the second step, the deployment engine is used to deploy the target environment including all required middleware and application components. However, in contrast to tailored engines, these deployment engines are not dynamically generated. They are general-purpose deployment engines, i.e., non-specialized complex systems, which are not specifically tailored for certain application topologies [WBL15c].
To sum up, no general recommendation can be derived which variant of deployment engine to prefer. It highly depends on the concrete usage scenario and the application environments to be deployed. For example, when managing a large number of small applications such as microservices, having a tailored engine for each of them can add a remarkable amount of management complexity. By relying on a shared general-purpose engine this complexity can be reduced. However, since microservices are typically based on diverse technologies, the utilized deployment technologies differ as well. Therefore, if a general-purpose engine is used, it must be extensible by providing appropriate mechanisms such as provided by the PlugInvoke framework (Section 7.2).

8.4 Validation beyond Continuous Delivery

The previous sections discussed various validation scenarios to show how the approaches and prototypes presented in this work can be applied in the scope of continuous delivery. Although this work focuses on supporting the implementation of continuous delivery pipelines and their underlying application environments, some of the presented approaches and prototypes are domain-independent and thus can be applied in other contexts, too. Therefore, the following sections discuss further validation scenarios beyond the scope of continuous delivery.

8.4.1 Any2API as Development Kit for TOSCA Implementation Artifacts

As discussed in Section 7.1.1, the Topology and Orchestration Specification for Cloud Applications (TOSCA) [OAS13b] is a standard to enhance the portability and management of Cloud applications. TOSCA is centered around the concept of application topology models. Topology templates are defined as graphs consisting of nodes and relationships to specify the topological structure of an application. As a foundation for defining such templates, node types and relationship types are defined as shown in Figure 7.1 (Section 7.1.1). These are used to create and wire corresponding
node templates and relationship templates as part of a topology template. Beside its properties, requirements, and capabilities, a node type specifies operations to be invoked by the overarching orchestration layer such as a deployment plan. These operations define their input and output, but they are abstract. To make them executable, implementation artifacts are attached to operations. Different kinds of implementation artifacts (IAs) exist such as scripts or Java WAR files providing a Web service API that can be invoked by a deployment plan. Consequently, IAs are key building blocks of an application topology because they implement and provide the logic of operations. TOSCA allows for attaching multiple IAs to a particular operation. For example, one IA provides a REST API to the orchestration layer, another IA provides a SOAP API. Both IAs must implement the same functionality, but serve it through different kinds of interfaces. Therefore, the TOSCA engine or its operation orchestration mechanism can select one of them, depending on which kind of interface is preferred. Providing multiple alternative IAs per operation significantly improves the portability of node types because of the increased chance that at least one IA can be processed by the employed TOSCA engine.

Manually developing IAs from scratch is time-consuming, especially for non-trivial IAs that provide functionality through multiple alternative APIs as discussed before. This is because not only the actual logic of the IA has to be implemented, but additional code has to be maintained to serve proper APIs. The Any2API framework was presented (Section 7.3) and implemented as a prototype (Section 8.2.4) to generate APIs for arbitrary executables. Although its major usage was discussed in the scope of generating APIs for existing artifacts such as deployment scripts retrieved from a knowledge base, Any2API can also be utilized to wrap individually developed executables as APIs. The generated APIs can then be used as IAs to be attached, for example, to TOSCA node type operations. Therefore, Any2API serves as a development kit for creating and maintaining TOSCA implementation artifacts. IA developers can completely focus on implementing the actual logic of the IA as an executable and then generate multiple APIs to be attached to a certain node type. This approach simplifies the development
process of IAs significantly. Several provisioning scripts were developed for evaluation purposes \cite{WBL15} to manage resources such as virtual servers and database instances in the Amazon Cloud. Based on additionally specified artifact metadata for each script, both REST APIs and SOAP APIs were generated to enable their usage as IAs.

### 8.4.2 APIfication of Executables and their Orchestration in e-Science

Many simulations and complex calculations performed in the field of e-science are implemented as a set of executables in the form of scripts or compiled programs. A broad variety of scripting and programming languages such as Python, Perl, C, or Fortran are used, depending on the preferences and existing expertise of participating scientists. Sharing and integrating such executables can be a powerful way to combine results of different scientific areas. However, the orchestration of such diverse executables is a major technical challenge due to the large diversity of the underlying technologies. Moreover, some scientists refuse to share these executables because of non-disclosure. For example, included algorithms are the core of their current research and thus strongly confidential. By wrapping scientific executables using Any2API, the generated API implementation can be hosted on a trusted server, so only the API endpoint is shared with peer scientists. Furthermore, the generated API for a particular executable hides the technical complexity to use it. For instance, interactions with a confidential and proprietary Fortran executable can happen through a REST API or SOAP API. Such APIs enable the usage of higher-level orchestration techniques such as scientific workflows \cite{SHK+11,SK13} to integrate and combine their underlying executables.

Section 8.3.3 already outlined an e-science scenario that utilizes Any2API to provide the functionality of plain scripts through generated APIs, i.e., the APIfication of scripts. The main focus of that scenario was on creating a continuous delivery pipeline that produces containerized APIs for the given Matlab scripts. This validation scenario covers another e-science use case, namely the physical simulation of how the material behavior of solid
bodies changes during their aging process at high temperatures [SHK+11]. The original implementation of this simulation is named OPAL (Ostwald-Ripening of Precipitates on an Atomic Lattice) and consists of a set of Fortran programs and scripts, which are manually executed in sequence. Any2API is used to generate a containerized API for each Fortran executable or script as individual building blocks of the OPAL simulation. Technically, a self-contained and portable Docker container of the API implementation and the executable is generated for each Fortran executable. Because the generated and containerized APIs are self-contained, there are no specific dependencies on specialized e-science platforms or frameworks [ASV13; SA14] to serve the functionality of the underlying executables through Web service APIs.

Figure 8.11: OPAL workflow orchestrating generated SOAP APIs
Moreover, the use of container virtualization for this scenario aims to improve reproducibility in research as discussed in related works [CIS14; Boe15; CG16]. The lower part of Figure 8.11 outlines the containerized APIs. The prepare.sh script receives the initial input and performs some domain-specific transformation of the input data to eventually store them inside the shared data volume. This shared volume is accessible by all containers, so the large chunks of input and output data can be efficiently shared through this volume. The Fortran executable opal-mc runs the main part of the simulation. Next, the opal-clus executable analyzes the output data of opal-mc to identify atom clusters. These clusters are then processed by opal-xyzr to determine their size distributions and mean radii. Finally, the simulation results are visualized as a plot and a video by running the Python scripts vis-plot.py and vis-video.py. All generated APIs are SOAP APIs to enable their orchestration by the OPAL workflow as outlined in Figure 8.11. Both the artifact metadata and the generated WSDL file for the opal-clus executable are provided as an example. The OPAL workflow [SHK+11] is implemented in BPEL [OAS07] and runs on Apache ODE. However, the generated SOAP APIs are just one single option out of various alternatives. By utilizing ANY2API to generate the APIs, the functionality of all Fortran executables and scripts of the OPAL simulation can be provided through different kinds of APIs such as REST APIs to fit other usage scenarios. The generated REST API description for the opal-clus executable are provided as an example.

8.4.3 Resource-oriented Informal Processes

Traditionally, the field of business process management is driven by business process languages such as BPEL [OAS07] and BPMN [OMG11a]. These languages enable the modeling and execution of clearly defined processes such as sequences of tasks and activities that potentially run in parallel.

1. [http://docs.docker.com/engine/userguide/containers/dockervolumes](http://docs.docker.com/engine/userguide/containers/dockervolumes)
2. [http://gist.github.com/jojow/0e5f6cea3d052999b662b60ad37f6d2f](http://gist.github.com/jojow/0e5f6cea3d052999b662b60ad37f6d2f)
5. [http://gist.github.com/jojow/27d1f2043baee807e04579abb907b60](http://gist.github.com/jojow/27d1f2043baee807e04579abb907b60)
However, some processes in more agile environments are considerably less predefined and therefore more informal. For example, an informal process in the context of a software development project is typically centered around a set of different kinds of resources such as human resources (software developers, operations personnel, etc.) and IT resources (software artifacts, code repositories, issue trackers, etc.). Instead of a fixed sequence of activities to be performed, an informal process is driven by a set of goals. All resources act toward these process goals or they are utilized by other resources to get closer to the goals. The concept of Informal Process Essentials (IPE) [SBLW15] provides such a goal-driven and resource-centric approach for specifying and executing informal processes.

When dealing with IT resources for DevOps-oriented informal processes, DevOpsBase can be utilized as knowledge base (Section 8.2.1) to identify and retrieve resources such as deployment scripts or virtual machine images that are required in the scope of a certain informal process. All selected resources in conjunction with corresponding process goals are then part of the IPE model as shown in Figure 8.12. To create a deployable IPE model out of it, some resources must be packaged and wired to enable an environment for running the informal process toward its goals. Any2API can be utilized as one of multiple building blocks in this context to generate APIs.
for lower-level resources such as deployment scripts. These APIs are then utilized by corresponding orchestration techniques to ease the integration and combination of certain resources. The resulting deployable IPE model is used to create instances of it. Depending on the packaging mechanism (Docker, TOSCA, etc.) of the deployable IPE model, a corresponding engine is required for the instance creation. The IPE model instance provides an environment with all resources to run the informal process and work toward the specified process goals. Since informal processes are agile by their nature, not only certain goals but also involved resources change over time. Therefore, the feedback & refinement loop helps to iteratively improve the IPE model, so updated instances of it can be created.

8.4.4 Migration of Existing Applications

There are various reasons for existing applications that are running on a particular infrastructure to be migrated into another environment. As a prominent example, the advent of Cloud computing pushed many organizations toward migrating at least parts of their applications to Cloud-based infrastructures. Moreover, applications are moved between Cloud infrastructures, for example, in case a specific provider offers a better
cost model. The pay-per-use and self-service capabilities of various Cloud providers attract organizations to run their applications at reduced overall costs and gaining from the flexibility of rapidly provisioning and decommissioning infrastructure resources such as virtual servers. Technically, the migration of existing applications is usually a non-trivial process. Binz [Bin15] proposes ‘AROMA’ as an automated migration approach [BBKL14a]. A case study of AROMA considers the Moodle application[^1] which provides a popular learning platform. The initial step is to discover application topologies of existing applications as part of so called ‘enterprise topology graphs’. In case of Moodle, Figure 8.13 outlines the discovered topology, representing the currently running instance of the application. This instance is hosted on an Ubuntu 12.04 VM provisioned by OpenStack.

As part of this validation scenario the goal is to host the migrated Moodle application instance on DigitalOcean[^2]. To execute the application migration, the originally discovered topology is not sufficient because it misses the required artifacts such as DigitalOcean-specific provisioning logic for managing virtual servers. Moreover, the deployment and wiring logic of the middleware and application components (Apache, PHP, MySQL) is not part of the application instance topology. Since the AROMA approach is technically based on TOSCA, it assumes a broad variety of TOSCA modeling artifacts such as node types to be available. The AROMA migration assistant [Bin15] helps create a deployable topology model that is used to create application instances in the target environment. Existing TOSCA modeling artifacts are utilized to create such deployable topology models. DevOpsBase can be utilized as knowledge base (Section 8.2.1) in conjunction with the TOSCAfy framework (Section 8.2.2) to produce required node types that are missing in the underlying collection of node types used by the AROMA migration assistant. For example, if a node type for provisioning virtual servers on DigitalOcean is not available, a corresponding provisioning script may be provided through DevOpsBase to generate a node type out of it using the TOSCAfy framework. Similarly, the discovered components of the instance

[^1]: http://moodle.org
[^2]: http://www.digitalocean.com
topology such as MySQL 5.5 and PHP 5.3 can be used as queries against the knowledge base to find solutions to be used for creating an instance of the application in the target environment. Two concrete examples include the Juju charm `mysql`\(^1\) and the Docker container `php/apache`\(^2\) as solutions retrieved from the knowledge base to deploy the key components of Moodle. Optionally, Any2API can be utilized in between to provide the functionality of certain artifacts such as scripts through proper APIs, such as REST APIs or SOAP APIs. This helps to utilize established orchestration mechanisms such as overarching deployment plans implemented in BPEL. Finally, the migration is finished by using OpenTOSCA to create an application instance in the target environment, i.e., the DigitalOcean infrastructure. As a result of the discussed integration, the AROMA approach does not exclusively rely on existing TOSCA modeling artifacts, but can be combined with the approaches and prototypes presented in this work. Consequently, a potentially larger collection of TOSCA modeling artifacts is available for migration scenarios.

8.5 Chapter Summary and Discussion

Initially, this chapter presented an integrated architecture to outline how the different building blocks presented in this work fit together in order to effectively support the GATHER’n’DELIVER method. Next, several prototypes were described as a foundation for the validation scenarios that are performed to evaluate the presented approaches. More specifically, the GATHERBase & DevOpsBase prototype does not only cover the knowledge base itself, but also the tooling to populate and maintain it, as well as creating diverse knowledge base variants. The prototype implementation of the TOSCAfy framework includes a basic set of modules to enable the transformation of certain artifacts such as Chef cookbooks into TOSCA modeling artifacts. Moving toward runtime, the PLUGINVOKE framework was implemented and integrated with OpenTOSCA’s operation invoker to simplify the execution of diverse artifacts. The Any2API prototype can be used alternatively or

\(^{1}\) [http://jujucharms.com/mysql/trusty](http://jujucharms.com/mysql/trusty)  
complementary to the PlugInvoke framework. Beside various analyzer, packager, invoker, and access modules, it provides two generators to produce REST APIs and SOAP APIs. In addition to the previously discussed integration of the PlugInvoke framework with OpenTOSCA’s operation invoker, further integration points with the OpenTOSCA toolchain were shown in this chapter.

Beside enabling the validation scenarios, the presented prototype implementations confirm the actual feasibility of the concepts and approaches presented in this work. Therefore, the prototypes themselves are an initial validation. However, the discussed validation scenarios are key to further evaluate the approaches, frameworks, and prototypes in different contexts.
The scenarios are divided in two major categories: (i) since continuous delivery pipelines and their underlying application environments are a major focus of this work, the first category contains several validation scenarios in this scope. (ii) However, some of the presented approaches and prototypes are domain-independent; therefore the second category contains a number of validation scenarios beyond the scope of continuous delivery. Figure 8.14 provides an overview of which frameworks and prototypes were implemented and utilized as part of which validation scenario. The case study annotation inside the table means that the corresponding framework such as PlugInvoke or Any2API were utilized conceptually, but the associated prototype was not used and executed for the particular validation scenario. To sum up, both the presented prototype implementations and validation scenarios show that the previously proposed approaches are technically feasible. Moreover, diverse use cases are provided by the validation scenarios. These can be used as a source of inspiration for further usage scenarios.
Continuous delivery is a rapidly emerging practice to significantly accelerate software release cycles. Implementing continuous delivery is not only technically important to build software more rapidly and with higher quality, but also possesses essential economic implications. As discussed in Chapter 1, since many of today’s users and customers of software products and services in various fields expect fast and immediate feedback loops, applying continuous delivery provides a critical competitive advantage. Chapter 1 outlined several research challenges in this context. These challenges were addressed by specific research contributions as part of this dissertation. The required background for this work was discussed in Chapter 2 in order to describe key fundamentals and to provide an overview of relevant related works. Chapter 3 to Chapter 8 presented each contribution in detail. Although the individual contributions were clearly driven by the intent of supporting the systematic implementation of continuous delivery, several are generic enough to be applied to other domains beyond continuous delivery. The following Section 9.1 provides a summary of the research contributions, additionally indicating their scope and limitations. Finally, Section 9.2 elaborates on potential research opportunities to be examined in future works.
9.1 Summary of Research Contributions and Limitations

The research contributions presented in this work focused on technical aspects of implementing continuous delivery. As discussed in Section 2.1, continuous delivery pipelines are technically utilized as a foundation of highly automated software delivery processes. Several application-specific stages (build, test, production, etc.) typically compose such a pipeline. Each stage relies on a specific application environment such as either a build or test environment to run stage-specific activities such as building or testing an application. While stages can share application environments in the case of their environment requirements being similar, building such application environments was a major focus of this work.

The initial research contribution of this work was presented as a method to implement continuous delivery pipelines and their underlying application environments using gathered solutions. As the name of the Gather’n’Deliver method described in Chapter 3 implies, it consists of the gather phase and the deliver phase to comprise both the gathering of solutions and the process of continuously delivering software based on these solutions. Since the method involves activities carried out by diverse experts, efficient collaboration was an important consideration. This does not only include collaboration among differently specialized experts such as architects, developers, and operations personnel, but also how organizations collaborate based, for example, on a shared knowledge base. Gathered solutions are organized and maintained through such a knowledge base, i.e., the knowledge base connects the gather and deliver phases of the method. While the method is domain-specific and limited to the scope of continuous delivery, it is meant to be a conceptual framework and thus remains on a generic and abstract level to permit its implementation in various manners. A dedicated discussion followed, focusing on a number of tangible possibilities regarding the means by which the method could be technically implemented.

The application environment metamodel established the approach of maintaining continuous delivery pipelines based on application environments. This is the second research contribution described in Chapter 4. Although
the stages and application environments of a continuous delivery pipeline are highly application-specific, a set of common application environments were discussed as part of that chapter. As a technical foundation of the knowledge base, the metamodel comprises key entities and their relations including solutions, application environment requirements (AERs), and application environment topologies. AERs aim to drive the selection of specific solutions that eventually compose application environment topologies. Such topologies are then used as templates to create instances of application environments. Furthermore, the metamodel includes the formal representation of AERs and solutions, as well as their matchmaking using labels. More specifically, hierarchically structured labels are used to express both requirements and capabilities. Since capabilities are attached to solutions, AERs can utilize corresponding labels to express dependencies on solutions that provide certain capabilities. In addition, relations between solutions can be specified using the previously introduced labeling and matchmaking mechanism. The presented metamodel is domain-specific to application environments and limited to three kinds of relations between solutions: dependencies, recommendations, and conflicts. However, additional relations can be added by extending the metamodel accordingly.

To refine the application environment metamodel and its labeling mechanism, [Chapter 5](#) covered the third research contribution by presenting a systematic classification approach for solutions that serve as building blocks of application environments. Multiple classification dimensions were identified to serve as a foundation of several label taxonomies in order to hierarchically structure labels for the purpose of matching solutions’ capabilities with application environment requirements. These dimensions consist of the type of solution, its binding (region binding, provider binding, etc.), its mode of use (executable, CLI, API, etc.), and its usage style (virtualization-level, infrastructure-centric, etc.). The process of resolving AERs was formally described based on the metamodel using pseudo-code algorithms. Generating skeletons for application environment topologies was another key aspect discussed as part of this contribution. Both the classification dimensions and the resulting label taxonomies are currently limited to various techni-
cal aspects for fine-grained decision support for solution selection. Thus, additional aspects that may drive the decision process such as operational costs and service level agreements are not considered. This limitation could be eliminated by introducing additional label taxonomies comprising these dimensions or by attaching corresponding metadata to individual solutions.

Populating the knowledge base with diverse solutions is essential for making it a useful source of building blocks of application environments. Chapter 6 addressed this concern by presenting the fourth research contribution: the collaborative and automated gathering of solutions. More specifically, the concept of collaborative solution repositories was introduced by adopting established concepts from collaborative software development approaches. These include shared and version-controlled repositories, as well as storing and maintaining structured documents inside them. In contrast to traditional code repositories centered around application-specific source code artifacts, solution repositories aim to foster the collaboration among different kinds of experts to manage solutions and their metadata. Since manually maintaining a significant number of solutions through the knowledge base does not scale, an automated approach was presented to discover and capture certain solutions. Because solution repositories are not designed to provide fine-grained query mechanisms to efficiently find solutions that fit specific application environment requirements, knowledge base variants are built for this purpose. Therefore, the ‘eat your own cooking’ principle was followed by adopting the concept of continuous delivery pipelines to create diverse knowledge base variants, which themselves are used to implement continuous delivery pipelines and corresponding application environments for specific applications. Finally, the GATHERBASE framework was presented to integrate and combine the previously described concepts in order to achieve the overarching goal of the collaborative and automated gathering of solutions. Although the solutions gathering approach is exclusively applied in the scope of continuous delivery and application environments, the underlying concepts are not domain-specific and may thus be transferred and applied to other domains.

Gathered solutions inside the knowledge base aim to provide a large-scale
foundation for selecting diverse solutions as building blocks of application environments. However, their combination, integration, and orchestration at runtime is another major challenge addressed by the fifth research contribution described in Chapter 7. To streamline the orchestration of different kinds of solutions, each solution must provide its functionality through proper APIs. Some solutions such as Cloud services typically do this natively, others such as lower-level executable artifacts (deployment scripts, configuration definitions, etc.) are missing APIs completely. Therefore, several approaches were presented to address this issue by providing additional APIs, for example, by generating individual APIs for executable artifacts. First, TOSCAfy was presented as standards-driven transformation framework to utilize TOSCA [OAS13b] as a standard and unified modeling representation of diverse artifacts. Second, the PlugInvoke framework aims to simplify the invocation of very different kinds of artifacts through a uniform interface. Third, the Any2API framework enables the automated APIfication of artifacts, i.e., different kinds of API implementations can be generated for different kinds of artifacts. In terms of limitations an additional overhead appears at buildtime (wrapping artifacts using TOSCA, generating APIs, etc.) and runtime (artifacts called through PlugInvoke framework or generated API). The overhead represents the cost for streamlining and thus significantly simplifying the orchestration layer when different kinds of solutions have to be combined, for example, to deploy instances of an application environment. This contribution is also generic enough to be applied outside the context of application environments and deployment automation. Indeed, any kind of executable artifact can potentially be wrapped, so its functionality is provided through APIs using this approach.

The sixth and final contribution of this work, described in Chapter 8, provided an integrated architecture, implementation, and validation, considering all of the previously described research contributions. While the integrated architecture connects the presented approaches, the prototype implementations aim to confirm their feasibility. Several validation scenarios that were built upon these prototypes evaluated the approaches in the scope of continuous delivery, application environments, and beyond. The valida-
tion performed beyond continuous delivery and application environments is key to demonstrate how the approaches presented in this work can be additionally applied to other domains.

9.2 Research Opportunities

Taking this work as a starting point for further research opportunities, several candidates can be identified as potential future lines of investigation. To extend and refine the solution classification approach presented in Section 5.2, further classification dimensions and label taxonomies could be established. The existing taxonomies focus on technical aspects to drive the decision making and solution selection process. Further taxonomies could cover non-technical dimensions such as categories of service level agreements and payment options to provide additional decision support when choosing between certain solutions.

Furthermore, a recommendation engine could be connected to the knowledge base tasked with automatically suggesting both similar and related solutions for a given solution set. This would assist in exploring and evaluating further solution candidates in addition to those identified based on application environment requirements that were perhaps overly constrained. For example, a specific application environment requirement may express the need for a MySQL database server\(^1\) so MySQL solutions are considered only. However, MariaDB\(^2\) is an open-source fork of the MySQL database server, so MariaDB solutions could be suggested by the recommendation engine even though the original requirement focused on MySQL. Another aspect would be to incorporate solution-specific runtime data into the knowledge base. If, for instance, a specific deployment script regularly fails on a particular operating system, this information could be added to the knowledge base to be considered when evaluating the selection of this specific solution.

On a higher level, the knowledge base could be analyzed, for example, using data mining and pattern mining techniques to identify potentially

\(^1\)http://mysql.com
\(^2\)http://mariadb.org
reusable patterns, as well as anti-patterns to avoid. More specifically, such a systematic analysis could help to establish technical patterns that capture complementary knowledge, such as which solutions are typically used in combination and for which kinds of application environments they are commonly utilized. As proposed previously, technical anti-patterns could potentially be extracted by considering runtime data: if a certain combination of solutions fails on a regular basis, this could be a strong indicator that the corresponding solution set might not be a proven path to follow. These patterns could then also be suggested by the previously proposed recommendation engine, connected to the knowledge base.

The concepts and approaches presented in this work focused on technically implementing individual stages and their application environments as key building blocks of continuous delivery pipelines. Future work could complement these by providing decision and design support on how to structure such pipelines, i.e., how to arrange its stages, which parts to run in parallel, and when it makes sense for multiple stages to share application environments. Such decisions could be driven by diverse aspects such as performance and isolation considerations. For example, sharing and reusing application environments usually renders a pipeline faster and more efficient in terms of resource consumption. However, the isolation of the affected stages is limited because of a shared environment, which might be an issue in some cases.

Another research opportunity appears in the context of the automated capture of solutions, as discussed in Section 6.2. Solutions are not only automatically stored inside solution repositories, but are also classified according to the label taxonomies in an automated manner. Optimizing the automated classification of diverse solutions is a research challenge on its own. Contiguous to straightforward approaches such as matching label names with keywords that are part of a solution, considerably more sophisticated strategies can be established and evaluated. Machine learning techniques and other artificial intelligence approaches can be utilized to enhance the results of automated solution classification. For example, solutions that were manually added and classified by experts can be used as training data to
make the system learn how to properly classify new solutions.

The validation scenarios beyond the scope of continuous delivery as discussed in Section 8.4 confirmed that the Any2API framework presented in Section 7.3 and its automated APIfication approach can be applied in additional contexts. With microservice architectures on the rise [New15; Fam15; Ric15; BHJ16] as a ‘modernized flavor’ of service-oriented architectures, this approach could be extended to provide the foundation for a service development framework. Clear separation of concerns could be achieved by maintaining the code for the required application and business logic without focusing on which kind of API is harnessed in order to make the functionality available. Any2API could then be utilized to generate APIs such as Web service APIs for each service. Consequently, deciding on a specific kind of API does not have to happen immediately when implementing a service. The decision can even be changed later by generating another kind of API, for instance, replacing a SOAP/WSDL API by an HTTP-based RESTful API without altering the code implementing the application logic. Moreover, multiple APIs could be generated to serve diverse requirements. For example, an efficient messaging-based API may be preferred for internal communication among services, whereas an external-facing interface may better provide an HTTP-based RESTful API because customers are more familiar with this kind of API. In addition to the technical implementation of APIs, evaluating and improving their usability [MS16] becomes increasingly important. Therefore, future research efforts could focus on investigating on the usability of generated APIs and potentially enhance the process of producing APIs correspondingly.

Although “the idea of virtual reality (VR) has been around for more than 50 years, and with successive improvements” [PFS+16], a tipping point seems to have been reached: “modern graphics cards provide sufficient compute power to render detailed, realistic scenes in high resolutions, and at the same time (...) headsets are affordable, they have high-resolution displays, and they eliminate perceivable motion-tracking lag, which was causing issues such as headaches and nausea before. (...) They will open many possibilities for VR beyond gaming” [PFS+16]. Consequently, VR applications could be
used in a professional context as an alternative to traditional graphical user interfaces or command-line interfaces. In the scope of this research, it could be investigated how VR-based knowledge base variants (Section 6.3) can be provided to explore and identify solution candidates, as well as to draft continuous delivery pipelines and application environments based on a set of chosen solutions. Another alternative kind of user interface could be provided by intelligent chatbots in the context of messaging platforms of different providers such as Slack\(^1\) and Facebook\(^2\). The actual goal of these approaches is to allow users to interact with such bots using natural language. Technically, existing open-source software such as Hubot\(^3\) can be utilized to implement a chatbot. Therefore, another research opportunity would be to evaluate if and how a chatbot-based interface could be employed to interact in natural language with the knowledge base discussed in this work, as well as collaboratively creating and maintaining application environments as part of continuous delivery pipelines.

\(^1\) http://api.slack.com/bot-users
\(^2\) http://developers.facebook.com/docs/messenger-platform
\(^3\) http://hubot.github.com
Bibliography


through Repositories.” In: Computer Science – Research and Development (2016) (cit. on pp. [20][21][111][117][119][123]).


All links were last followed on March 19, 2017.

Bibliography
List of Figures

1.1 Overview of addressed research challenges .......................... 16
1.2 Overview of research contributions ................................. 18

2.1 Continuous delivery pipeline example ............................... 29
2.2 Application topology example for production environment ...... 33

3.1 Gather’n’Deliver method ............................................. 51

4.1 Application environment metamodel ................................. 64
4.2 Positioning of application environment requirements ............ 65
4.3 Example instance of application environment metamodel ...... 67
4.4 Refined metamodel ..................................................... 72
4.5 Example instance of the refined part of the metamodel ...... 73

5.1 Solution type label taxonomy .......................................... 90
5.2 Solution binding label taxonomy ..................................... 91
5.3 Mode of use label taxonomy .......................................... 92
5.4 Usage style label taxonomy ........................................... 93
5.5 Overview of findSolutionSets algorithm ............................ 96
5.6 Two iterations example for applying Algorithm 5.2 algorithm . 105
<table>
<thead>
<tr>
<th>Section</th>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Continuous Delivery Pipeline – informal</td>
<td>29</td>
</tr>
<tr>
<td>2.2</td>
<td>Application Topology – informal</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Label</td>
<td>74</td>
</tr>
<tr>
<td>4.2</td>
<td>Map: Label Set</td>
<td>75</td>
</tr>
<tr>
<td>4.3</td>
<td>Solution Boundary</td>
<td>75</td>
</tr>
<tr>
<td>4.4</td>
<td>Solution Properties</td>
<td>76</td>
</tr>
<tr>
<td>4.5</td>
<td>Solution</td>
<td>76</td>
</tr>
<tr>
<td>4.6</td>
<td>AER Predicate</td>
<td>78</td>
</tr>
<tr>
<td>4.7</td>
<td>Union of AER Predicates and Boundary Predicates</td>
<td>79</td>
</tr>
<tr>
<td>4.8</td>
<td>Map: Predicate Applicable</td>
<td>79</td>
</tr>
<tr>
<td>5.1</td>
<td>Set Representation of Disjunctive Normal Form</td>
<td>95</td>
</tr>
<tr>
<td>5.2</td>
<td>Map: Filtered Solution Set</td>
<td>97</td>
</tr>
<tr>
<td>5.3</td>
<td>Map: Combined Solution Sets</td>
<td>98</td>
</tr>
<tr>
<td>5.4</td>
<td>Map: Conflicting Solutions</td>
<td>99</td>
</tr>
<tr>
<td>5.5</td>
<td>Map: Unresolved Requirements</td>
<td>104</td>
</tr>
<tr>
<td>5.6</td>
<td>Topology Graph</td>
<td>106</td>
</tr>
</tbody>
</table>
## List of Symbols

<table>
<thead>
<tr>
<th>Symbol or Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{L} )</td>
<td>Set of all labels (Definition 4.1).</td>
</tr>
<tr>
<td>( l )</td>
<td>A specific label ( l \in \mathcal{L} ) (Definition 4.1).</td>
</tr>
<tr>
<td>labelSet</td>
<td>Maps a label ( l \in \mathcal{L} ) to a set of labels ( L \subseteq \mathcal{L} ) consisting of ( l ) and all of its parent labels (Definition 4.2).</td>
</tr>
<tr>
<td>( S )</td>
<td>Set of all solutions (Definition 4.5).</td>
</tr>
<tr>
<td>( s )</td>
<td>A specific solution ( s \in S ) (Definition 4.5).</td>
</tr>
<tr>
<td>( u_s )</td>
<td>The URI(^1) that uniquely identifies a specific solution ( s \in S ) (Definition 4.5).</td>
</tr>
<tr>
<td>( L_s )</td>
<td>The set of labels ( L_s \subseteq \mathcal{L} ) specifying the capabilities that are provided by a specific solution ( s \in S ) (Definition 4.5).</td>
</tr>
<tr>
<td>( B_s )</td>
<td>The set of boundary predicates associated with a specific solution ( s \in S ) (Definition 4.3).</td>
</tr>
<tr>
<td>( b_{t,q}^d )</td>
<td>A specific boundary predicate ( b \in B_s ) with an associated type ( t ), qualifier ( q ), and optional description ( d ); the predicate is associated with a specific solution ( s \in S ) (Definition 4.3).</td>
</tr>
<tr>
<td>( \mathcal{N}_{\text{properties}} )</td>
<td>Set of all property names (Definition 4.4).</td>
</tr>
<tr>
<td>( \mathcal{V}_{\text{properties}} )</td>
<td>Set of all property values (Definition 4.4).</td>
</tr>
</tbody>
</table>

\(^1\)http://tools.ietf.org/html/rfc3986
<table>
<thead>
<tr>
<th>Symbol or Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{prop}_s )</td>
<td>Maps property names ( n \in \mathcal{N}<em>{\text{properties}} ) to property values ( v \in \mathcal{V}</em>{\text{properties}} ) for a specific solution ( s \in S ) (Definition 4.4).</td>
</tr>
<tr>
<td>( \mathcal{R} )</td>
<td>Set of all AER predicates (Definition 4.6).</td>
</tr>
<tr>
<td>( r_{t,q}^d )</td>
<td>A specific AER predicate ( r \in \mathcal{R} ) with an associated type ( t ), qualifier ( q ), and optional description ( d ) (Definition 4.6).</td>
</tr>
<tr>
<td>( \mathcal{B} )</td>
<td>Set of all boundary predicates (Definition 4.7).</td>
</tr>
<tr>
<td>( \mathcal{P} )</td>
<td>Union of all boundary predicates ( \mathcal{B} ) and all AER predicates ( \mathcal{R} ) (Definition 4.7).</td>
</tr>
<tr>
<td>( p_{t,q}^d )</td>
<td>A specific boundary predicate or AER predicate ( p \in \mathcal{P} ) with an associated type ( t ), qualifier ( q ), and optional description ( d ) (Definition 4.7).</td>
</tr>
<tr>
<td>( \text{predicateApplicable} )</td>
<td>Maps a pair consisting of a solution ( s \in S ) and a predicate ( p \in \mathcal{P} ) to a Boolean value (Definition 4.8).</td>
</tr>
<tr>
<td>( \mathcal{P}^{\text{DNF}} )</td>
<td>A set of predicate sets representing a logical expression in disjunctive normal form (DNF) consisting of AER predicates and boundary predicates ( p \in \mathcal{P} ) (Definition 5.1).</td>
</tr>
<tr>
<td>( \text{filteredSolutionSet} )</td>
<td>Maps a pair consisting of a solution set ( S_1 \subseteq S ) and an AER predicate set ( R \subseteq \mathcal{R} ) to a filtered solution set ( S_2 \subseteq S_1 ) (Definition 5.2).</td>
</tr>
<tr>
<td>( \text{combinedSolutionSets} )</td>
<td>Maps a pair consisting of a set of solution sets ( {S_1, S_2, \ldots, S_n} ) and an AER predicate set ( R \subseteq \mathcal{R} ) to a set of combined solution sets (Definition 5.3).</td>
</tr>
<tr>
<td>( \text{conflictingSolutions} )</td>
<td>Maps a solution set ( S \subseteq S ) to a set of solution pairs, each consisting of two conflicting solutions (Definition 5.4).</td>
</tr>
<tr>
<td>Symbol or Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>unresolvedRequirements</td>
<td>Maps a pair consisting of a solution set $S \subseteq S$ and an AER predicate set $R \subseteq R$ to a set of unresolved AER predicates and boundary predicates $p \in \mathcal{P}$ (Definition 5.5).</td>
</tr>
<tr>
<td>$U$</td>
<td>A set of unresolved AER predicates and boundary predicates $p \in \mathcal{P}$ (Definition 5.5).</td>
</tr>
<tr>
<td>$g$</td>
<td>A topology graph consisting of a solution set $S_g \subseteq S$ and a set of edges $E_g$ (Definition 5.6).</td>
</tr>
</tbody>
</table>