Master Thesis

April 4, 2016

# Bifrost Toolkit: Data-Driven Release Strategies

## Formalize and automate real-time, data-driven live-testing methods using a DSL

**Dominik Schöni**

of Sumiswald, Switzerland (10-738-607)

**supervised by**

Prof. Dr. Harald C. Gall

Gerald Schermann

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Master Thesis

# Bifrost Toolkit: Data-Driven Release Strategies

## Formalize and automate real-time, data-driven live-testing methods using a DSL

**Dominik Schöni**

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

**Master Thesis**

**Author:** Dominik Schöni, dominik.schoeni@uzh.ch

**Project period:** 07.10.2015 - 07.04.2016

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

# Acknowledgements

I would like to thank Prof. Dr. Harald C. Gall for giving me the opportunity to write this thesis at the software evolution and architecture lab at the University of Zurich. I would also like to thank Dr. Philipp Leitner for introducing me to the topic of this thesis and Gerald Schermann for his inputs, guidance and great effort. Moreover, I would like to thank Joel Scheuner and Genc Mazlami for the great discussions in the s.e.a.l lab. My sincere thanks also goes to Lukas Merz for proofreading this thesis. Finally, special recognition goes out to my parents, Tiziana and Roland, for always encouraging and supporting me over the course of my studies, as well as my brother Pascal for his advice and knowledge from which I was able to benefit tremendously. Last but not least, I would like to thank my dearest girlfriend Melissa for motivating and inspiring me.

# Abstract

The pace of software development has steadily increased, being transformed by the notion of agile and iterative development. In the meantime, the process of turning source code into releasable software artifacts is similarly transforming using techniques such as continuous integration, delivery and deployment. By automating and streamlining the process of releasing software, developers receive a new set of methods to leverage real-time customer feedback in their development processes. Fueled by scalable and flexible approaches to software such as microservices-based architecture, companies start to integrate live-testing methods into their applications to benefit from real-time user data *e.g.*, by conducting A/B tests, silently launching new functionality or gradually testing new features in production. However, these kind of data-driven release strategies exist solely in large-scale corporations and few research has been conducted in this area. This thesis formalizes a general model of data-driven release strategies to allow complex and multi-staged software releases. Furthermore, a prototype based on the developed model is implemented to showcase its potential and to discover its pitfalls. In a qualitative evaluation, it is demonstrated how the prototype and thus the model allows for more complex scenarios in comparison to existing tools. Furthermore, a quantitative evaluation of the prototype shows that the chosen approach can be integrated into existing web applications with minimal performance constraints.

# Zusammenfassung

Die Entwicklung von Software wurde zunehmend schneller, transfomiert und beeinflusst von Techniken wie agiler und iterativer Entwicklung. Zugleich passt sich auch der Prozess an, welcher aus Code auslieferungsfähige Artefakte erstellt, unter anderem durch Techniken wie Continuous Integration, Delivery und Deployment. Durch die zunehmende Automatisierung und Vereinfachung des Auslieferungsprozesses erhalten Entwickler die Möglichkeit Kundenfeedback in Echtzeit zu erhalten, indem Metriken ausgewertet werden. Diese Metriken können Feedbackprozesse der Organisation beschleunigen. Angetrieben durch neuartige und flexible Architekturansätze wie Microservices integrieren Organisationen zunehmend live-testing in ihre Applikationen, um in Echtzeit durch Metriken profitieren zu können. Beispiele dafür sind A/B Tests oder das versteckte Ausrollen neuer Funktionalitäten. Diese Daten-getriebenen Prozesse existieren allerdings hauptsächlich in grossen Firmen und bis anhin wurde wenig Forschungsarbeit in diesem Bereich geleistet. Diese Arbeit präsentiert wie Daten-getriebene Auslieferungsprozesse in ein generalisiertes Modell formalisiert werden können, welches komplexe und mehrstufige Prozesse erlaubt. Basierend darauf wird ein Prototyp implementiert, um das Potenzial sowie mögliche Schwachstellen des Modells aufzuzeigen. Eine qualitative und eine quantitative Evaluation des Prototypen zeigen, dass das vorgestellte Modell komplexere Szenarios erlaubt als existierende Lösungen und dass der gewählte Ansatz es erlaubt das entwickelte Toolkit mit minimalem Performanceverlust in bestehende Webapplikationen zu integrieren.

# Contents


**1 Introduction** **1**
- 1.1 Contribution and Research Questions . . . 2
- 1.2 Thesis Outline . . . 2

**2 Background** **5**
- 2.1 Building and Releasing Software . . . 5
  - 2.1.1 Continuous Integration . . . 5
  - 2.1.2 Continuous Delivery . . . 6
  - 2.1.3 Continuous Deployment . . . 7
  - 2.1.4 DevOps . . . 8
  - 2.1.5 Deployment Pipeline . . . 9
- 2.2 Data-Driven Software Release . . . 9
  - 2.2.1 Methods of Live-Testing . . . 9
  - 2.2.2 Implementation Techniques . . . 11
- 2.3 Microservices Architecture . . . 13
  - 2.3.1 Definition . . . 13
  - 2.3.2 Advantages of Microservices . . . 14
  - 2.3.3 Communication in Microservices . . . 14
  - 2.3.4 Metrics in Microservices . . . 14
- 2.4 Summary . . . 16

**3 Related Work** **17**
- 3.1 Releasing Software . . . 17
- 3.2 Dynamic Release Management . . . 18
- 3.3 Existing Tools . . . 18
  - 3.3.1 Research Prototypes . . . 19
  - 3.3.2 Industry Tools . . . 20

**4 Data-Driven Release Strategies using Bifrost** **21**
- 4.1 Problem Analysis . . . 21
  - 4.1.1 Usage Scenario . . . 21
- 4.2 A general model for releasing software . . . 22
  - 4.2.1 Characteristics . . . 22

4.2.2 Formal Definition . . . 23
4.3 Bifrost Toolkit Prototype . . . 25
4.3.1 Requirements . . . 25
4.3.2 Approach . . . 26
4.3.3 Technologies . . . 27
4.4 Bifrost DSL . . . 28
4.4.1 Converting the Release Model into a DSL . . . 28
4.4.2 Deployment . . . 29
4.4.3 Strategies . . . 30
4.4.4 Actions . . . 30
4.5 Bifrost Engine . . . 33
4.5.1 Overview . . . 33
4.5.2 Interpreter . . . 34
4.5.3 Deployment . . . 34
4.5.4 Model . . . 34
4.6 Bifrost Proxy . . . 37
4.6.1 Configuration . . . 37
4.6.2 Sticky Sessions . . . 37
4.6.3 Filters . . . 37
4.7 Bifrost CLI . . . 38
4.7.1 Overview . . . 38
4.8 Bifrost UI . . . 39
4.8.1 Overview . . . 39

**5 Evaluation** **41**
5.1 Qualitative Evaluation . . . 41
5.1.1 Tools . . . 41
5.1.2 Dimensions . . . 43
5.1.3 Analysis and Discussion . . . 44
5.1.4 Summary . . . 47
5.2 Quantitative Evaluation . . . 48
5.2.1 Method . . . 48
5.2.2 Request Performance . . . 51
5.2.3 Filter Performance . . . 53
5.2.4 Release Performance . . . 55

**6 Final Remarks** **59**
6.1 Conclusion . . . 59
6.1.1 Threats to Validity . . . 60
6.2 Future Work . . . 61
6.2.1 Formal Verification . . . 61
6.2.2 Extend Bifrost Toolkit . . . 61
6.2.3 Integration in Deployment Pipeline . . . 61
6.2.4 Feature Toggles . . . 62

**Glossary** 69

**Bifrost Sample Release Strategy** 73

**Performance Evaluation Filter Configurations** 77

**Bifrost Engine Installation Guide** 83

**Bifrost Toolkit Integration Guide** 85

**Bifrost Microservices Sample Application Guide** 89

# List of Figures

2.1 From agile development to DevOps . . . 8
2.2 Visualization of A/B testing and canary launches . . . 10
2.3 Visualization of shadow launches and blue/green deployments . . . 11
2.4 Dynamic request routing . . . 13

4.1 Architecture diagram of the Bifrost Toolkit, embedded in a sample application . . . 26
4.2 The Bifrost CLI provides real-time status information while a release is running . . . 38
4.3 Ongoing release in Bifrost UI, updating its progress in real-time . . . 39

5.1 Bifrost microservices sample application as used in the performance evaluation . . . 49
5.2 Average response time per tested endpoint . . . 51
5.3 Histogram of response times, depicting both proxied and non-proxied . . . 53
5.4 Achieved response time while scaling from 0 to 120 threads . . . 54
5.5 Average response time during the release test in milliseconds . . . 56

# List of Tables

5.1 Feature-comparison of analyzed tools . . . 42
5.2 Comparison of live-testing tools . . . 45
5.3 Results of request performance test in milliseconds . . . 52
5.4 Average response time in milliseconds grouped by active threads . . . 55
5.5 Average response time during a phase in milliseconds and delta to the baseline . . . 57

# List of Listings

2.1 Example of feature toggles using fflip . . . 12
4.1 EBNF representation of the Bifrost DSL . . . 29
4.2 Example of a deployment specification in Bifrost . . . 29
4.3 Example of multiple strategies that make use of the *next* property . . . 30
4.4 Route action that redirects 50% of the traffic from service A to service B . . . 31
4.5 Request action that checks http://www.google.ch once in 5 seconds for 10 times . . . 31
4.6 Pause action that stops the release after finishing the running strategy . . . 32
4.7 Metric action that compares CPU-Load using Prometheus as provider . . . 32
4.8 AND action that tests the reachability of two services . . . 33
4.9 Action-class implementation in Bifrost Engine . . . 36
5.1 Example of how to use Scientist! [jba14] . . . 43

# Chapter 1

# Introduction

The process of releasing software has been undergoing significant changes in recent years [Dea07]. Software companies tend to release not yearly or quarterly anymore but weekly, daily or even per-commit. To support faster processes in a resilient and stable way, automation becomes key. Whereas software development has adapted to iterative and agile processes for years, operations has begun to change rapidly as well, influenced by techniques such as continuous integration [DMG07], delivery [Che15] and deployment [Pul13]. In addition, cloud providers bring ubiquitous computing to the masses and promise unlimited scalability and flexibility for everyone. This shift creates challenges and opportunities. The notion of resilient and auto-scalable services in the cloud means that software architecture has to adapt accordingly. Software developers adapted, increasingly following an architectural approach called microservices. Additionally, the deployment of software becomes increasingly agile and more versatile. A shift in paradigm, commonly referred to as DevOps [FS14], where the separation between the operations and development teams blurs, embraces *infrastructure as code* and advocates faster release cycles. Developers have begun to embrace the fact that software, being a non-physical good, has properties such as intangibility that allow for interesting and sophisticated delivery mechanisms. Large-scale companies such as Facebook, Netflix or Etsy started to use these properties to gain insights about their products using data-driven release methods while collecting customer feedback. The transformation from agile development to a feedback driven process is even labeled as the "stairway to heaven" [OAB12], where the final step is described as follows:

> "The entire R&D system responds and acts based on instant customer feedback and where actual deployment of software functionality is seen as a way of experimenting and testing what the customer needs." [OAB12]

An example of such a data-driven method are A/B tests, conducted between different user groups, to find out if a new feature serves its intended purpose. Another popular approach is to *dogfeed* applications to internal testing groups in order to find bugs that are hard to detect using automated testing. This adds a new level of complexity to their release configuration, with which they deal with using custom-made configuration management tools.

## 1.1 Contribution and Research Questions

As of today no general model of releasing software exists that incorporates live-testing strategies, *e.g.*, A/B testing. This thesis aims to propose a general model on how to structure software releases in a way that complex release procedures can become a part of automated rollouts using continuous integration, delivery and deployment. The first posed research question therefore is:

> **RQ1: How can we formalize a (generic) model for data-driven release and deployment strategies?**

The model is later validated by building a prototype implementation called Bifrost, named after the burning rainbow bridge between the world and the realm of gods in Norse mythology. The prototype serves as a first implementation of the developed release model. It explores how techniques like dynamic request routing can be incorporated into existing applications following a microservices-based architecture to allow complex, data-driven release methods when deploying new versions of existing services. Therefore, the second derived research question is:

> **RQ2: How can we build a tool that supports and automates data-driven release and deployment strategies for microservices-based architectures in a non-intrusive way?**

The prototype is evaluated in a qualitative comparison to existing approaches, as well as in a quantitative evaluation concerning its performance impact on applications. Existing approaches to deliver customer-specific software versions mostly use feature toggles to change the behavior of software at runtime, depending on the current user. This approach is prone to errors and can become tedious to manage depending on the application scale. Applications built using the microservices-based architecture approach allow alternatives such as dynamic routing to serve customer requests to specifically built versions of services.

## 1.2 Thesis Outline

The thesis is structured as follows:

- Chapter 2 introduces relevant topics and terms. It focuses on how software deployment has evolved during the recent years, considering how agile practices influence the way software gets developed and delivered. Special attention is given to the evolution of the deployment pipeline and the concepts of continuous integration, delivery and deployment. It also touches briefly on the notion of DevOps in the context of this thesis. It introduces and defines a selection of relevant live-testing methods such as canary launches, shadow launches, A/B testing or blue/green deployments. Furthermore, the approach of microservices architecture is introduced to give an understanding on how modern web applications are currently developed.
- In Chapter 3, a selection of related work that is close to the chosen topic of this thesis is presented, containing research papers and showcasing existing tools and approaches from both research and industry.

- Chapter 4 shows the development of a general model to formalize release strategies and applies the developed model in a prototype implementation called Bifrost. The Bifrost Toolkit is presented in detail in the second part of this chapter, focusing on its architecture and selected implementation details.
- The evaluation in Chapter 5 is split into two parts. The first part is a qualitative evaluation that compares the Bifrost Toolkit to existing research prototypes and industry tools within a set of defined dimensions. The second part focuses on a quantitative evaluation considering the performance impact of the Bifrost Toolkit upon existing applications.
- Finally, Chapter 6 summarizes the conclusions of this thesis and discusses potential future work regarding the presented model and the implemented prototype.

# Chapter 2

# Background

The rise of the cloud and ubiquitous computing [BYV+09] has given developers new possibilities in terms of resource availability and ease of provisioning. This allows for faster releases and new methods of dynamic deployments. The first part of this chapter focuses on the process of building and releasing software, whereas the second part concentrates on data-driven software releases considering its methods and techniques. Lastly, the concept of microservices-based architecture is introduced and discussed in terms of inter-service communication and metrics.

## 2.1 Building and Releasing Software

While many methods were developed that covered software design, development, and testing, the process of delivering software was often overlooked in comparison [HF10]. This changed through the introduction of key concepts that brought the notion of agile development [BBvB+01] into the software build and release process, beginning with continuous integration.

### 2.1.1 Continuous Integration

> "Continuous Integration (CI) is the process of building software with every change committed to a project's version control repository." [DMG07]

Software projects often suffer from lengthy integration periods, commonly referred to as "integration hell" [Cun09]. As software gets developed by different teams, the phase of bringing the software together is complex and error-prone, let alone the fact that incompatibilities between parts and components are often discovered during the integration period [HF10]. These difficulties result in long integration periods and unclear timelines as a result thereof [FF05]. Continuous integration was first proposed by Booch [Boo91], although the pace of integration is comparatively low. Beck [Bec00] explicitly stated that no code "sits unintegrated for more than a couple of hours", defining the notion of CI.

CI describes a set of techniques and processes that aim to prevent integration problems. It is able to alleviate these difficulties by "eliminating the blind spot" of knowing where you are

and what bugs are left to be fixed. CI automates tasks such as compiling, static code analysis, running unit or acceptance tests, generating coverage reports or building deployable artifacts [FS14, Aga11, Mar07].

According to Fowler [FF05] practices to implement CI include, but are not limited to:

- Maintaining a single source repository
- Build automation
- Self-testing builds, automatically triggered builds upon commits to the main branch of development
- Keeping the build process fast and fixing broken builds immediately

The process of continuous integration knows no standardized method [SB14]. It merely describes a set of goals that can be achieved using different tools. There exist a lot of continuous build systems that are suitable to implement continuous integration, such as Buildbot[1], GitLab[2] or Jenkins[3].

### 2.1.2 Continuous Delivery

The industry is quickly embracing the notion of value- and business-driven software development. Agile processes allow developers to deliver changes and new features in software to users quickly, either for use in production or for testing and verification of requirements [Abr08]. Software is not evolving slowly in month over month releases, but the process is being streamlined into weekly, daily or even hourly release schedules. This makes it necessary to continuously have software available for release, which introduces the concept of continuous delivery. This concept helped customers and developers alike to follow the spirit of the agile manifesto [BBvB+01], which states that developers should "deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale". Humble and Farley define continuous delivery as:

> "A software development discipline where you build software in such a way that the software can be released to production at any time" [HF10]

It includes the process used to build software that can be deployed at any given time. In the industry, the practice of continuous delivery has been increasingly adopted. A practical example of how continuous delivery can be applied, was given by Chen for Paddy Power [Che15], describing their process in implementing it. The company also introduced manual steps at the end of the automated chain, primarily for user acceptance tests and exploratory testing purposes [Che15]. As reasons for adopting continuous delivery, they cite advantages such as accelerated time to market, building the right product due to a shorter feedback-cycle or generally improved production and efficiency through automation.

[1] Buildbot: `http://buildbot.net/`
[2] Gitlab - Code, test and deploy together: `https://about.gitlab.com/`
[3] Jenkins - Build great things at any scale: `https://jenkins.io/index.html`

### 2.1.3 Continuous Deployment

While the process of actually deploying the release is often automated [Che15], the decision whether the release should be deployed is still manually triggered. While a release-capable build exists, it still only gets delivered to production on scheduled releases. Releasing software is for many development teams a tense and stressful process. Humble *et al.* [HF10] describe manual deployments as an anti-pattern which should be avoided in favor of fully automated ones. This facilitates a repeatable, reliable and error-resilient deployment-process that is not susceptible to human judgment and decision-making. Whereas manual deployments need proper documentation, the scripts that enable automated deployment already serve as up-to-date and complete documentation of the process. Beck and Andres [BA04] introduced the concept of daily deployments in 2004. In contrast to the first steps of continuous deployment, nowadays every commit gets automatically built, tested and if, continuous deployment is practiced, even released directly into production [HF10]. Continuous deployment is therefore described as:

> "A software process that releases software changes automatically to end-users after they pass the required tests". [HF10]

In reality, the actual implementation of these techniques often differ from the theoretical definition. A study conducted by Rahman *et al.* [RHWP15] that aimed to map continuous deployment practices used in bigger companies, refined the definition of continuous deployment as:

> "A software engineering process where incremental software changes are automatically tested, and frequently deployed to production environments". [RHWP15]

Implementing continuous deployment is "harder as it seems" [NS13]. Challenges that developers face when adopting continuous deployment can be categorized in organizational, technical, and social.

#### Organizational Challenges

Adopting continuous deployment is a company-wide effort [CSA15] and cannot be implemented by one development team itself, as it touches many parts of a product and thus requires effort from the top-level management to successfully implement these practices [OAB12]. In addition, software often has dependencies to existing tools or components delivered through suppliers. These suppliers have to adapt their process as well to make sure that continuous deployment can deliver its advantages [OAB12].

#### Technical Challenges

From a technical perspective, introducing continuous deployment also means to introduce new tools, which sometimes do not meet the required quality or maturity standards [OAB12]. Apart from an automated testing per se, the fast-paced release schedule also requires fast testing to make sure that the testing process does not limit the release frequency [NS13]. This problem creates difficult decisions where companies even drop part of their test suite to "cut the fat and focus on

those test areas which are prone to failure" [MKA+13] in order to adapt rapid-release schedules. A direct consequence of pushing code directly into production is that bugs slipping through quality control are now discovered by customers [CSA15], highlighting the need for effective monitoring and feedback mechanisms.

### Social Challenges

Continuous deployment basically removes the notion of versions. What formerly consisted of one release, now gets split and gradually rolled out. Highlighting changes and new features to customers therefore requires an adaption in product marketing [CSA15, NS13]. The transition from continuous delivery to continuous deployment also benefits from having pro-active customers, willing to explore such new concepts [OAB12]. Another challenge is that the responsibility for code quality shifts, adding more pressure to software developers as each commit that passes quality control also reaches customers directly [CSA15].

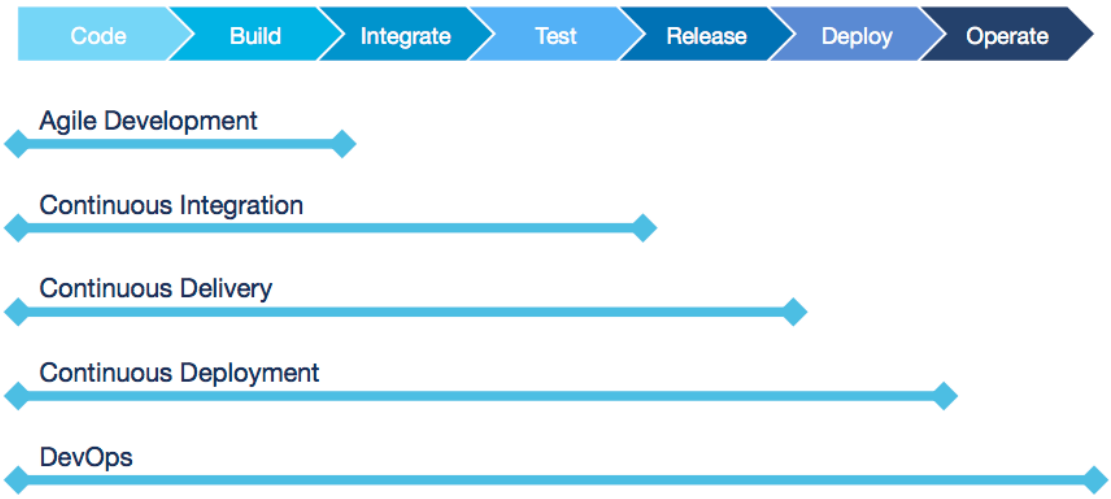



Figure 2.1: From agile development to DevOps [Rig14]

## 2.1.4 DevOps

The concepts and techniques mentioned beforehand, are increasingly part of a wider concept that glues development and operations together. As the release frequency increased, it became clear that the historically separated teams of development and operations needed to adapt to the faster cycle. Emerging techniques such as *Infrastructure as Code* helped shaping a new collaboration between development and operations teams, commonly known as DevOps [Dev09].

Hüttermann defines DevOps as follows:

> "DevOps describes practices that streamline the software delivery process, emphasizing the learning by streaming feedback from production to development and improving the cycle time." [Hüt12]

The notion of DevOps is not purely technical but it includes several aspects also summarized under the acronym CAMS: Culture, Automation, Measurement and Sharing [Joh10]. It revolves around practicing the right culture that allows automated release management, integrated feedback-loops using measuring tools, and creating an environment where problems and mistakes are shared and discussed. From a technical perspective it is another step in automating the process from commit to production, as writing out infrastructure and environment configuration in code allows for simpler scalability of services (spinning up 5 or 50 machines becomes an easy matter if the process of provisioning machines is completely automated), better encapsulation, faster disaster recovery and new abilities to respond elastically to demand [EHN14]. It also helps to remove the gap between development environment and production, which is essential for having a functional deployment pipeline [HF10].

### 2.1.5 Deployment Pipeline

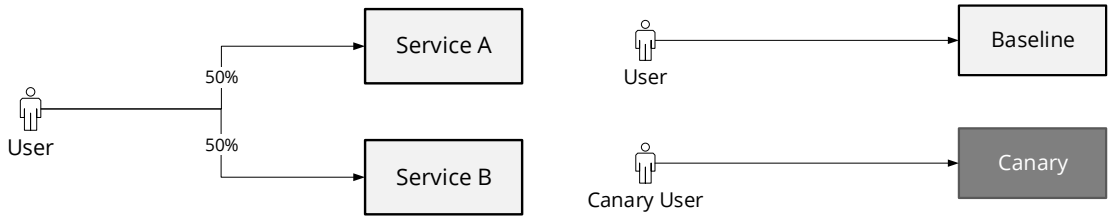
The process that gets triggered by every commit and turns version controlled software into executable software artifacts is generally called the deployment pipeline [HF10]. Code that gets committed into version control automatically steps through the pipeline, with the goal of bringing software into the hands of its users. Each task the code has to fulfill is harder and the environment becomes more and more production-like. How far the code is able to get in the deployment pipeline therefore determines its production readiness. This can be visualized using a multi-step process as shown in Figure 2.1.

## 2.2 Data-Driven Software Release

The deployment pipeline covers building, testing, and releasing software from commit to production. As previously mentioned, the testing part of the deployment pipeline can suffer from cuts to allow faster release cycles [MKA+13]. A consequence of this shift is that software features often get tested in practice by real users [FFB13] using a range of tools and techniques.

### 2.2.1 Methods of Live-Testing

To gain an understanding of how the process of live-testing individual features or service versions works, the following sections discuss and present the most common methods used. Afterwards, a set of techniques is presented, showing how the given methods can be implemented from a technical perspective.




(a) A/B testing (b) Canary launches

Figure 2.2: Visualization of A/B testing and canary launches

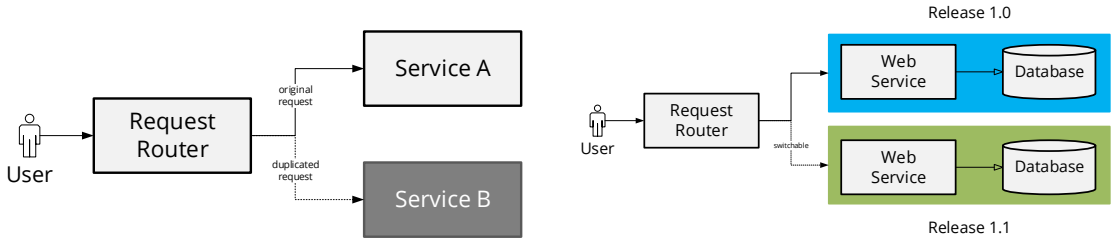
## Partial Rollouts

A set of methods is generally combined under the term *partial rollouts* as they all are used to expose new functionality on a fixed or random set of users.

**A/B Testing**: A/B testing (see Figure 2.2a) is a way to compare two versions of software with each other, often only differentiated in one tested aspect, to determine the effect of a certain change [KLSH09]. It is a form of statistical hypothesis testing. Typically, A/B tests only involve a small subset of users. Given a big enough sample size, this enables to determine what users want instead of generating elicit requirements in advance [FFB13]. The general use case is to determine whether users use new features in practice, but it can also be used to identify situations in which users use new features in unexpected or harmful ways [FFB13].

**Canary Launches**: Similar to A/B testing, canary launches (see Figure 2.2b) compare two versions of software with each other. In comparison however, canary launches mostly addresses the problem of introducing a new version into a stable environment [TSM+15]. The new version (potentially unstable, called *canary*) is compared to the existing version (stable, called *baseline*) in terms of a set of criteria such as stability, performance, or correctness. The goal of a canary launch is to make sure that the canary does not perform worse than the baseline with respect to the selected criteria.

**Shadow Launches**: New features often suffer from performance and reliability issues when facing production-like load levels. To mitigate such problems, shadow launches (see Figure 2.3a) allow to deploy features on production but not visible to users. An example is Facebook [FFB13], which rolled out its new chat service to users without enabling the user interface component allowing them to stress-test the service with realistic usage behaviors and at scale. Through monitoring, development teams were able to fix the remaining bugs and scalability concerns before enabling the feature for all users entirely. This kind of deployment is sometimes also referred to as *dark launch*.




(a) Shadow launches (b) Blue/green deployments

Figure 2.3: Visualization of shadow launches and blue/green deployments

### Phased Rollouts

Phased rollouts introduce the possibility to expose more and more users to a new or an alternate version of a service. This guarantees a smooth transition from for example canary to baseline, as unexpected issues such as scalability concerns due to increasing workload are discovered during the process [AM16]. If issues are uncovered, the service can quickly be restored to its previous version to avoid further trouble for users. The process of restoring previous versions is often referred to as rollback.

### Blue/Green Deployments

The idea of blue/green deployments (see Figure 2.3b) is to deploy two identical versions of your production environment. While one of the two versions (*e.g.*, blue) is currently productive, the other version (*e.g.*, green) is just deployed but does not serve any users. This setup allows running tests of a new version in an environment that is identical to production and releasing a new version with a simple change of the routing configuration [HF10]. Difficulties like handling database migrations have to be handled though and introduce non-trivial challenges. While there exists advice on how to manage database migration and synchronization issues [Fow10a], there is no general consent whether one actually should. A guide published by CloudNative even advices to avoid any synchronization scenarios [San15].

## 2.2.2 Implementation Techniques

The previously introduced methods of live-testing rely on the ability to dynamically alter the behavior of applications at runtime. There exist two prominent approaches on how to implement this from a technical point of view, namely feature toggles and dynamic routing.

### Feature Toggles

One method to dynamically enable and disable code segments are feature toggles [Fow10b,Hod16] (also referred to as switches, flippers or feature flags [Fow10b]). A feature toggle can be as simple as an if-else branch checking a variable in the applications code (see Listing 2.1 for an example). A better implementation uses external configuration providers or files to facilitate dynamic

testing of different features [Hod16]. Feature toggles allow incomplete features to be shipped into production as the code in question never gets executed given the configuration is set properly [Fow10b]. There exist various tools that help introducing feature toggles in applications such as LaunchDarkly[4] (a multi-platform Software as a Service solution) or individual language-based frameworks such as Togglz[5] for Java or fflip[6] for JavaScript. Each of these frameworks and tools have in common that they require developers to modify existing application code using the provided framework.

```
1 // Get all of a user's enabled features
2 var Features = fflip.userFeatures(someFreeUser);
3 if(Features.closedBeta) {
4   console.log('Welcome to the Closed Beta!');
5 }
6 // Or, just get a single one
7 if (fflip.userHasFeature(someFreeUser, 'closedBeta')) {
8   console.log('Welcome to the Closed Beta!');
9 }
```

Listing 2.1: Example of feature toggles using fflip

They are a useful and cheap addition at first, but tend to introduce complexity over time. Some developers consider feature toggles as a prime example of technical debt [Bir11], defined as a situation in which long-term code quality is traded for short-term gain [BCG+10]. A famous example of how feature toggles can introduce serious bugs is an example from financial services firm *Knight Capital Group*. Due to reusing an old feature toggle, their trading software managed to generate a 460 million U.S. dollar loss in 45 minutes [Sev14].

### Dynamic Routing

Web-based applications are able to leverage a different approach called dynamic routing. Instead of embedding code into the service, the basic premise of dynamic routing is to use a request-router which determines what service (and thus which version) should receive a specific request. A patent describing a similar mechanism to accomplish phased rollouts of version upgrades was filed in 2009 [LBR09]. In comparison to feature toggles, no modification of existing application code is necessary. A simple example of dynamic routing are blue/green deployments (see Section 2.2.1) which are generally implemented using reverse-proxies[7] that offer dynamic routing functionalities such as NGINX[8] or HAProxy[9].

The method is also capable of more fine-granular routing. Depending on the protocol used, the request-routers can use header fields (*e.g.*, in HTTP[10]) to determine which service should receive a specific request. An example can be seen in Figure 2.4, depicting a HTTP-based request-router that distinguishes between two user groups and the corresponding servers that serve the request.

[4] LaunchDarkly: `https://launchdarkly.com/`
[5] Togglz: `http://www.togglz.org/`
[6] fflip: `https://github.com/FredKSchott/fflip`
[7] A proxy server that serves resources of one or more servers to clients, as if it itself was the origin [Apa13]
[8] NGINX - High Performance Load Balancer, Web Server & Reverse Proxy: `https://www.nginx.com/`
[9] HAProxy -The Reliable, High Performance TCP/HTTP Load Balancer: `http://www.haproxy.org/`
[10] Hypertext Transfer Protocol [FGM+99a]

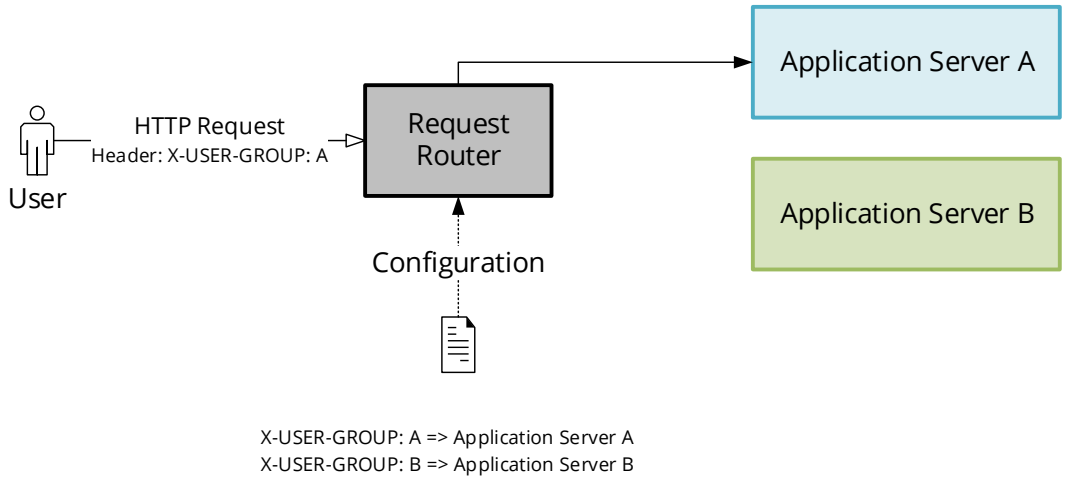



Figure 2.4: Example of dynamic request routing using HTTP-headers

Routing based on request properties has previously been used especially to implement dynamic load balancing of web applications [CCP99, BKK00].

## 2.3 Microservices Architecture

As the following thesis focuses on web applications built using a microservices architecture approach, the following section will define the term in the scope of this thesis to guarantee a consistent understanding should the term or its meaning change in the future.

### 2.3.1 Definition

The term microservices is getting a lot of attention in the software development industry. Although a lot of blogs, companies, and people are talking about microservices there exists no unified view on what microservices architecture consists of. It is highly discussed, either dismissed as being yet another buzzword or presented as what Service-Oriented-Architecture should have been in the first place [McK15]. Fowler and Lewis [FL14] define microservices as follows:

> "The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies." [FL14]

This definition matches Newman's from his book "Building Microservices", where microservices are described as small and autonomous services that work together [New15]. A microservice should do one thing well and therefore adhere to the Single Responsibility Principle introduced by Robert C. Martin [Mar03].

### 2.3.2 Advantages of Microservices

The whole notion of splitting applications into multiple services comes from the realization that monolithic architectures tend to end as what is described as "big ball of mud" [FY00]. Microservices architecture reduces the risk by advocating small and lightweight components, splitting up code in terms of business boundaries to create explicit services for a given piece of functionality [New15]. The fact that components are now autonomous also means that it is possible to adapt new technologies into the application stack over time. A single microservice does not have to share the technology stack with other services as long as the communication protocol is clearly defined and easily accessible [New15]. Another key advantage is the inherent scalability of microservices in comparison to monolithic applications. Services can be scaled on a fine-granular level instead of running multiple copies of a monolithic application [New15, ND14]. A common approach is to use load balancers in order to distribute workload to multiple service instances [New15].

### 2.3.3 Communication in Microservices

Microservices architecture focuses on creating small, self-contained and decoupled services. These services use lightweight mechanisms such as HTTP [FL14] for communication. Albeit other protocols can be used as well, 95% of participants of a survey stated that they use HTTP in combination with RESTful APIs for inter-service communication [SCL15].

Thönes [Tho15] proposes that complexity is being moved from the inside of monolithic applications to the networking layer and thus into the infrastructure. This change has been made possible by programmable infrastructure and the general approach of automation that is mandated by the DevOps movement [Tho15]. One of the challenges in adapting microservices architecture lies in getting services to know each other in the first place. This problem is summarized under the term *service discovery*. One possible solution to service discovery is the usage of DNS[11]. Services use hostnames as entry point to the IP addresses of services [New15]. DNS is a standard that is widely used and supported. One of its few disadvantages is that updating DNS entries can become a troublesome process if caching-mechanism are involved [New15].

### 2.3.4 Metrics in Microservices

As metrics play an important role in data-driven release processes, their collection and relevant tools are discussed in the following sections in the context of microservices web applications,

[11] Domain Name System, a distributed and decentralized naming system for computers

including a short overview of the difficulties that dynamic environments such as cloud providers introduce.

## Collecting Metrics

There exists a range of low-level metrics that can be analyzed to gain an understanding of a web services performance, such as its throughput, latency, response- or execution-time [Lad12]. High quality web services achieve high throughput, low execution- and response-time and low latency. These metrics concentrate on the user experience of a web service and can be derived from low-level metrics such as CPU load, I/O bandwidth or networking throughput. Apart from the typical metrics that can be measured by monitoring the hard- or software, business metrics play an important role in helping to make decisions [VM01]. A large amount of tools help to monitor and visualize business metrics such as NewRelic[12] or Ruxit[13]. Many available tools allow for systematic collection of key QoS metrics. cAdvisor[14] by Google is one example used to collect resource usage and performance characteristics of running containers. CollectD[15] is a daemon that collects system performance metrics. As certain events can go undetected by the underlying system, but are still valuable for the applications health and behavior. Utilities like Logster[16] parse log files of applications, extract useful information and aggregate that data into user-defined metrics. Such aggregation behavior that happens in the scope of individual services can also be found in statsD[17], which is a network daemon built upon NodeJS that is able to aggregate statistics received over UDP or TCP connections.

## Cloud Environments

The collection of metrics is a complex endeavor on its own. Many modern web applications that are built upon microservices-based architecture are deployed to virtualized environments or the cloud. Developers often use *Infrastructure as a Service* platforms (*e.g.*, Amazon Web Services[18], Google Cloud Platform[19] or Microsoft Azure[20]) or *Platform as a Service* platforms (*e.g.*, Heroku[21] or IBM Bluemix[22]). These platforms introduce new difficulties and challenges such as performance fluctuations [SLCG14], variable resource consumption caused by physical co-location with other services [TSM+15, HSVT12], or shared responsibilities between multiple parties concerning the service quality [HSVT12].

[12] NewRelic - Application Performance Management & Monitoring: `http://newrelic.com/`
[13] Ruxit - All-In-One Application Performance Management: `https://ruxit.com`
[14] cAdvisor: `https://github.com/google/cadvisor`
[15] collectd - The system statistics collection daemon: `http://collectd.org/`
[16] logster: `https://github.com/etsy/logster`
[17] statsD: `https://github.com/etsy/statsd`
[18] Amazon Web Services: `https://aws.amazon.com/`
[19] Google Cloud Platform: `https://cloud.google.com/compute/`
[20] Microsoft Azure: `https://azure.microsoft.com`
[21] Heroku - Cloud Application Platform: `https://www.heroku.com/`
[22] IBM Bluemix: `http://www.ibm.com/cloud-computing/bluemix/`

## 2.4 Summary

The release process of software is adapting to the fast-paced iterative development, using concepts such as continuous integration, delivery and deployment. Organizations are recognizing the need to bring development and operations closer together using the DevOps notion and thus create a fully automated deployment pipeline. This paves the way for new, data-driven approaches to release software. Especially in the field of web applications, which is enabled through cloud computing and highly-flexible approaches such as microservices-based architecture.

# Chapter 3

# Related Work

This chapter describes previous research, prototypes, and tools in the field of release management. The first section focuses on work in the area of release and deployment practices in general, whereas the second section concentrates on a selection of research prototypes and industry tools that are relevant to this thesis.

## 3.1 Releasing Software

Rahman *et al.* [RHWP15] have synthesized continuous deployment practices used in software development. However, the paper uses its own definition of continuous deployment that describes a software engineering process, where new software is automatically tested and "frequently deployed to production environments". This contrasts the more narrow definition established in the background section of this thesis, where continuous deployment releases every built artifact from a certain branch. Their study found that all adoptees "use the practices of automated deployment, automated testing, and repositories". Olsson *et al.* [OAB12] have conducted a multiple-case study to explore the transition from agile development towards continuous deployment. The paper uses a strong metaphor, speaking of the "stairway to heaven" by describing the typical evolution path for companies starting at traditional developing, going agile and finally using research and development as an experiment system where "deployment of software is seen more as a starting point for further tuning". However, no company that took part in the research study was classified in this last stage. The given reasons state that companies miss "the capability to effectively use the collected data to test new ideas with customers" [OAB12].

To actually learn and understand how continuous deployment may benefit a software process, one way is to learn about specific case studies of companies implementing it [RHWP15]. One example is the case study by Neely *et al.* [NS13], which focuses on how Rally Software transitioned from an eight-week release schedule to a continuous delivery model. The paper notes that the transition is not painless, not only from an engineering point of view. Also the business side of their company had to adapt to the new deployment technique, as a "consolidated enablement push every eight weeks was no longer an option", forcing to communicate information about new features differently than before. They acknowledged the change in process by communicating in smaller chunks, replacing extensive meetings to review new features with blog posts and emails. This finding is shared by Claps *et al.* [CSA15], stating that feature discovery was a challenge

introduced by adapting to continuous deployment.

Faster releases come at a certain price though. A study of Mantyla *et al.* [MKA+13] analyzed the results of execution runs and manual system-level test cases of Mozilla Firefox from 2006 to 2012, and concluded that to be able to release more often, the company had to cut down its testing suite. This was done to speed up their build time and allowed the team to "to cut the fat and focus on those test areas which are prone to failure". Selected areas were tested deeper in scope, whereas non-critical parts have been excluded from their automated setup. A similar conclusion was drown by Neely *et al.* [NS13], stating that long-running tests (one particular test suite took over 9 hours) limit fast releases. This due to the fact, that the software can only be released as fast as the test suite completes. A study conducted by Olsson *et al.* [OAB12] has shown that many developers that have hardware-oriented businesses also suffer from a lack of tooling support and have difficulties to adapt their processes to the fast-paced software development.

## 3.2 Dynamic Release Management

Rahman *et al.* [RHWP15] found that 16 of 19 adoptees of continuous deployment use monitoring systems to collect deployment related metrics. Further, most of the adoptees (17 out of 19) use gradual rollouts, and more than half (12 out of 19) practice dogfooding. This shows that gradual rollouts are relatively common in the industry. Big companies like Facebook have built specialized toolsets to enable them the automated delivery of complex release and deployment strategies [TKV+15]. They argue that frequent software upgrades in Internet services are not only possible today, but rather a necessity to survive against competition [TKV+15]. A similar argumentation can be found by Neely *et al.* [NS13], stating that eight week release cycles are too long. Schermann *et al.* [SCL15] conducted a quantitative, web-based survey that targeted developers and companies alike that identify themselves with building service-based products. They found that monitoring QoS attributes is quite common, although the majority of developers did not collect any business and custom metrics but focused on health and performance related runtime monitoring.

There exists a range of tools and products used in production that are used to monitor web applications. Tools like Splunk[1] or Nagios[2] allow the collection and analysis of data. However, the automatic integration of this data into the release process is lacking, and the task to automatically undeploy faulty releases is left to the operations or development team [AM16].

## 3.3 Existing Tools

There exist a number of tools and approaches that are similar in usecase or concept to the proposed model of this thesis. In the following sections, a subset of suitable tools are presented that either follow similar approaches to this thesis or aim at solving the same challenges.

[1] Splunk - Operational Intelligence: `http://www.splunk.com/`
[2] Nagios - The Industry Standard in IT Infrastructure Monitoring: `https://www.nagios.org/`

### 3.3.1 Research Prototypes

#### Rondo

Gunalp *et al.* [GEL15] presented "a tool suite for continuous deployment in dynamic environments". The motivation of the proposed solution lies therein, that systems become increasingly flexible and cloud-applications have access to dynamically allocated computing resources. Rondo aims at tackling four key requirements of a deployment process, namely "Reproducibility, Fault-tolerance, Continuous Adaptation and Customizability". The tool consists of a deployment description DSL implemented as an embedded-DSL using a fluent Java API and a second implementation using the Groovy scripting language. The DSL was used to define the final state of the deployment platform. A component called deployment manager was used to analyze and plan the transition from one deployment state to another, as well as execute and monitor its transition. The tool has been validated in the context of two different service platforms and by measuring its performance overhead concerning memory and deployment execution time. Their tool suite follows a different approach than this thesis though. Rondo handles deployable services as resources that are transitioned from a derived state to a expected final state during deployment. The tool has to be deeply integrated into the given application, as its deployment agent takes care of service modification and provisioning. Bifrost on the other hand, neglects this part in favor of a more generalizable approach that works with any web service given it speaks HTTP.

#### CanaryAdvisor

Tarvo *et al.* [TSM+15] propose a tool targeted towards a specific use case called CanaryAdvisor. The tool is capable of monitoring newly deployed versions using open source tools such as logstash[3] and collectd. Tarvo *et al.* apply statistical methods to their data analysis part to compute the correct decision whether the newly provided version (called canary) performs significantly worse than the already deployed version (called baseline). They encountered that fully automated decision making during canary testing is nontrivial, primarily due to the high variance in raw metrics. Bifrost aims to alleviate this issue by allowing users to choose their own metrics in the decision process, that may already be preprocessed, or are higher-level in general. Another challenge tackled included to define failure of releases, as this notion might vary from application to application or even from one version to another. The tool does not handle traffic-routing or deployment altogether, but aims solely at the data collection part of the problem.

#### GateKeeper

Tang *et al.* [TKV+15] describe GateKeeper as part of a broader research paper, focusing on Holistic Configuration Management at Facebook. Facebook uses a group of internally developed tools to handle their deployment and release setup, where one of the building blocks is GateKeeper. Embedded as a HHVM[4] extension, GateKeeper manages their feature toggles. A feature toggle can depend on so-called restraints, which are statically implemented in PHP[5] or C++[6]. These re-

[3] Logstash - collect, enrich & transport data: `https://www.elastic.co/products/logstash`
[4] HipHop Virtual Machine - VM that executes PHP and Hack using a JIT-Compiler: `http://hhvm.com/`
[5] PHP: Hypertext Preprocessor: `http://php.net/`
[6] C++ Programming Language: `https://isocpp.org/`

straints are specific to their application-setup and able to check various conditions of users. This allows GateKeeper to conduct complex A/B tests or phased rollouts. To prevent configuration errors, the gating logic is limited to the available restraints. If restraints are heavy to compute, GateKeeper utilizes a key-value store called Laser to retrieve either pre-computed results or outcomes of scheduled jobs.

### 3.3.2 Industry Tools

#### Vamp

Vamp[7] is conceptually similar to Bifrost. The tool is described as the "very awesome microservices platform", consisting of "a platform-agnostic microservices DSL, powerful A/B testing, canary launches, autoscaling and an integrated metrics & event engine". Vamp follows similar principles as Bifrost, by using a YAML-based DSL to define deployments and services. The tool has advanced capabilities for custom traffic routing and filtering based on headers, but does not allow users to dynamically change deployments based on real-time metrics or events. The latter are primarily used for monitoring purposes.

#### Scientist!

Scientist![8] is a ruby-based library, whose intended usage is to "carefully refactor critical paths" [jba14]. It is basically a library that provides feature toggling combined with the possibility to automatically set up experiments, controlling which users should receive which version. At the same time, metrics can be collected that are able to be reused for further analysis or published directly. No DSL is used, as the library is embedded in the source code of the targeted ruby project.

#### ION-Roller

ION-Roller[9] is a service consisting of an API, web app and CLI tool that orchestrates Amazon's Elastic Beanstalk[10] to provide safe immutable deployment, health-checks, traffic redirection and more. Its basic use case is to guarantee a smooth traffic redirection between service versions (*e.g.*, using a blue/green deployment) and thus facilitate an update process without service disruptions, especially if having multiple instances of the same service version.

[7] Vamp - the very awesome microservices platform: `http://vamp.io/`
[8] github/scientist: `https://github.com/github/scientist`
[9] gilt/ionroller: `https://github.com/gilt/ionroller`
[10] PaaS-Solution of Amazon: `https://aws.amazon.com/de/elasticbeanstalk/`

# Chapter 4

# Data-Driven Release Strategies using Bifrost

This chapter focuses on constructing a generic model for data-driven release strategies and how a tool based on this model can be built. First, the problem is analyzed in light of the background information given in Chapter 2. Second, an approach is introduced and explained how this thesis aims to tackle the problems of data-driven release strategies. It is shown how the chosen approach solves the problems in question and the developed prototype is presented.

## 4.1 Problem Analysis

Existing tools and research prototypes focus either on single usage scenarios [TKV+15, GEL15, TSM+15] or provide only basic support for data-driven release methods [Mag14]. In reality, developers are more interested in combining and chaining different methods with each other. Each method of live-testing can be regarded as merely another step in the deployment pipeline until a specific release is seen as truly production-ready. While there exists a plethora of tools that support the process of generating software artifacts and deploying these on infrastructure (either on bare-metal machines or virtualized cloud environments), no toolkit exists that supports data-driven release methods end-to-end and allows for deep customization.

### 4.1.1 Usage Scenario

A sample usage scenario has been constructed to showcase what a release process incorporating data-driven release methods might look like. This usage scenario is also being reused in Chapter 5 to evaluate the built system.

> "You are the developer of a system that sells goods and you have developed a web application following the microservices architecture approach. One of your services, called *Products*, provides a REST-API that enables other services to query information. As part of an ongoing process to increase the sales performance, you have been tasked to test out 2 new mechanisms in the application. These have been implemented in different code branches of *Products*, called *Products A* and *Products B*. Both of these

services have run through your deployment pipeline, passed all necessary tests and are ready to be deployed. The question remains which of the services offers improvements in real usage. The following release scenario is envisioned to roll out the new services, making sure that they perform as wished and fulfill the desired requirements:

- **Canary Launch**: Both services are deployed and served only to a predefined subset of customers, known to be interested in trying out new product experiences. They are monitored for runtime exceptions to detect troublesome behavior.
- **Shadow Launch**: Both services should receive the real load of all users, without interfering with daily business. Therefore, both are shadow launched to receive a realistic load pattern over an extended amount of time.
- **A/B Test**: Both services have proven themselves fit for production. However, it is not yet sure which of the new mechanisms performs better. You deliver the services to a small set of regular users, half of them testing *Product A* whereas the other half receives *Product B*. The existing service is still used to serve the majority of your customers. The sales performance is monitored throughout the process.
- **Phased Rollout**: You roll out the winner of the previous A/B test to all your users, albeit step by step to make sure the transition goes smoothly and without unnoticed side effects."

This usage scenario has subsequently been used to deduce a suitable approach to what the toolkit should include and narrow down its scope.

## 4.2 A general model for releasing software

In order to create a general model for releasing software, a set of characteristics has to be identified that are specific to data-driven release methods. These are analyzed and subsequently worked into a formalized representation of a general model of data-driven software releases.

### 4.2.1 Characteristics

The described methods of live-testing in Section 2.2.1 serve as a basis to determine key characteristics the developed model should fulfill. Four main characteristics have been identified that will be discussed in detail in the following sections.

**Ordered Execution**. Live-testing methods rely on the ability to dynamically change their behavior during the process. Examples of such behavior can be identified in A/B testing, canary launches or phased rollouts. These methods rely on multiple steps that modify the state of the application over time. Therefore, a release can consist of multiple states that are separated of each other. An example of such a behavior can be seen during canary launches, where candidates are deployed into the application and monitored. Depending on the outcome of the monitoring, the

application either gets modified and a service replaced or updated, or the previous state is restored.

**Parallel Execution**. The ability to change the behavior of multiple services at the same time is key to implement certain live-testing methods. Canary launches or shadow launches rely on continuous monitoring during the testing phase in order to trigger rollback strategies or automatic failover if they behave unexpectedly or to base the decision process on data collected during the testing period. Additionally, continuous monitoring may be used during phased rollouts of services to determine whether a service changes its behavior and quality when facing more load.

**Timed Execution**. Live-testing methods may require the collection, analysis or processing of data in regular intervals. Methods such as phased rollouts depend on timed increments to gradually introduce new services. Canary launches monitor candidates over extended periods of time in order to gain a representative set of usage data [AM16]. Depending on the usage scenario these methods may stretch over a couple of minutes, hours or even days.

**Data-Driven**. Certain methods of live-testing require additional information to proceed. Canary launches and A/B tests usually rely on manual intervention or extensive monitoring [AM16] to decide about their outcome. Considering that existing tools in the application landscape are tasked with analyzing log-data and deciding about the fitness of a particular service version, the model should allow the inclusion of external data into the decision process.

These four characteristics serve as a foundation to model a general representation of a release process. They represent a set of simple yet distinctive properties that are found in most of the described live-testing methods and are key to successfully create a general model.

### 4.2.2 Formal Definition

To begin with, a *Release* is modeled as follows:

$$Release: \qquad \{\{c_1, \ldots, c_n\}, (s_1, \ldots, s_n)\}$$

A release consists of a set of service configurations $\{c_1, \ldots, c_n\}$. Service configuration means that each identified service of the application needs its own configuration $c_i$ that exposes the necessary information to the release process. This information is used to further configure the services through actions, if necessary. The second part is a tuple of *strategies* $(s_1, \ldots, s_n)$, thus describing the characteristic of **ordered execution**. The order of the tuple elements determines the order of execution, whereas the default strategy following any $s_i$ is defined as $s_{i+1}$. Each strategy itself is defined as a set:

$$Strategy: \qquad \{Parallel_A, Success_A, Failure_A\}$$

A strategy's elements $Parallel_A$, $Success_A$ and $Failure_A$ are defined as follows:

$$\begin{aligned} Parallel_A: &\qquad \{a_1, \ldots, a_n \mid a_i \in Action\} \\ Success_A: &\qquad \{x \mid x \in Action\} \\ Failure_A: &\qquad \{x \mid x \in Action\} \end{aligned}$$

A strategy consists of a set of actions, $Parallel_A$. These are scheduled to be executed at the same time, thus match the characteristic of **parallel execution**. Two additional actions $Success_A$ and $Failure_A$ are used to model behavior upon either successful or failing outcome. Actions themselves, are defined as a tuple:

$$Action: \qquad A = \{\Theta, \Delta, \Omega\}$$

Actions consist of a state-modifying function $\Theta$, a timer $\Delta$ and supplied data $\Omega$. $\Theta$ is the function that is able to modify the state of a service configured as $c_i$:

$$\Theta: \qquad f(c_i) \rightarrow c_i'$$

This allows actions to modify the state of service configurations. The evaluation of an action is defined as a function that returns a boolean value as result:

$$f_a(a_i) = result \mid result \in \{True, False\}$$

The timer $\Delta$ controls the execution of an action by allowing it to be executed multiple times (and thus also multiple state manipulations) using an interval between executions. This introduces the characteristic of **timed execution** into the model. An action that is executed multiple times only returns true, if a certain threshold supplied to the timer $\Delta$ is reached:

$$f_\Delta(a_i) = \{f_a(a_i)_1 ... f_a(a_i)_n\} = \begin{cases} True & |f_a(a_i) \in True| \geq \Delta_{Threshold} \\ False & |f_a(a_i) \in True| < \Delta_{Threshold} \end{cases}$$

The supplied data $\Omega$ is able to influence the result of an action as its value can change over time, allowing for a **data-driven** outcome of an action. The result of a strategy is then represented by a tuple of boolean values, resulting from the individual evaluation of the strategy's actions. To determine the final outcome of a strategy the results are chained together using the boolean AND-operator:

$$f_s(\{a_1, .., a_n\}) \rightarrow f_a(a_1) \wedge ... \wedge f_a(a_n) \rightarrow result \mid result \in \{True, False\}$$

Strategies are thus guaranteed to only succeed if all their corresponding actions were successful. Following the evaluation of a strategy, either the $Success_A$-action or the $Failure_A$-action are executed. This mechanism introduces a flexible way to achieve goals such as:

- Skipping or explicitly specifying the next strategy by manipulating the strategy tuple $(s_1, \ldots, s_n)$
- Reacting to erroneous behavior *e.g.*, by initiating automatic rollbacks
- Modifying the state of service configurations depending on strategy outcome

## 4.3 Bifrost Toolkit Prototype

Using the problem analysis and the developed Bifrost release model, a prototype has been implemented to showcase the model's potential and discover possible improvements.

### 4.3.1 Requirements

Based on the previously introduced model, a high-level approach has been developed that is able to implement the specified model. As there are many potential ways of how the release model can be implemented, a set of requirements has been defined that have to be fulfilled by the developed prototype to narrow down the scope of this specific implementation.

- **Automatable**: To make it possible for developers to integrate the Bifrost Toolkit into their release process, the tool should be able to be fully automated by a CI-Environment, thus featuring file-based release strategies that can be version controlled and scheduled without human interaction.

- **Scalability**: The chosen approach should scale well, considering that applications experience changing load patterns and might be deployed in cloud environments featuring automatic horizontal scaling.

- **Microservices**: As an increasing amount of web applications follow the trend of using a microservices-based architecture, the chosen approach should be compatible with said architectural style. This means that highly decoupled web applications using HTTP as communication protocol should be fully supported.

- **Extendability**: Existing applications often feature monitoring solutions that are already deeply embedded in the software stack. The toolkit should respect existing work and allow developers to leverage existing data or easily extend the toolkit to do so.

- **Non-Intrusive**: It should be easy to integrate the toolkit into existing applications, without altering or rewriting the software that already exists.

The developed prototype focuses on the implementation of the Bifrost release model in the scope of microservices. As microservices push the decoupling of functionality through independent web services, dynamic routing (referring to Section 2.2.2) has been chosen as the technical approach to implement the functionality. The release model should be easy to understand by developers and allow them to formulate flexible and customizable release strategies on their own. Therefore, a domain specific language has been developed on the basis of the formalized release model. The main advantages of this approach are that it allows for a transparent, and better understandable representation of dynamic deployments [Fow08], making it easier to formulate non-trivial scenarios. At the same time, it is another step towards a fully automated process in comparison to GUI-controlled tools.

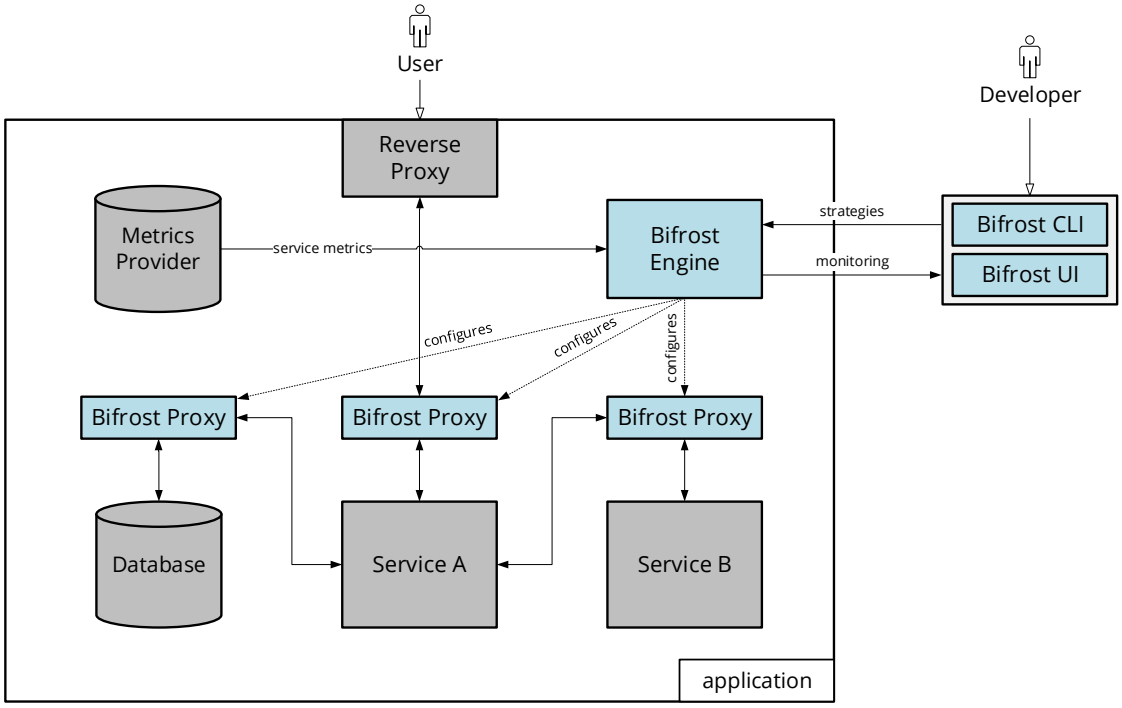



Figure 4.1: Architecture diagram of the Bifrost Toolkit, embedded in a sample application

### 4.3.2 Approach

The prototype consists of four main components, which together form the Bifrost Toolkit depicted in Figure 4.1.

- **Bifrost Engine**:This component has the main responsibility to orchestrate and properly configure the deployed proxies in the system, using the rules specified in the DSL that get interpreted by the Bifrost Engine component. It is also responsible to query metrics providers or external services (*e.g.*, NewRelic[1]) in order to use the supplied data when making decisions about the release process.
- **Bifrost Proxy**: The proxy implements the dynamic routing functionality to facilitate different release methods and to properly route HTTP-requests to the targeted service. Each service receives its own proxy, which prevents traffic bottlenecks and keeps the services decoupled.
- **Bifrost CLI**: Being a tool rather than a component, the Bifrost CLI is a command-line interface that is used to schedule new releases, by supplying instances of the DSL to the engine.
- **Bifrost UI**: The web interface allows for a graphical representation of the current application state, making it easier to follow complex release behaviors.

[1]NewRelic: `http://newrelic.com/`

The architecture of the planned system is demonstrated in Figure 4.1, embedded in a sample application consisting of three seperate services and a metrics provider that is monitoring the services. The requirements specified in Section 4.3.1 have been adressed as follows:

- **Automatable**: The system can easily be combined and accessed through integration services (*e.g.*, Jenkins CI[2]) and used in scripted deployment setups, as the command-line interface provides the possibility to automatically launch releases without user interaction. Release processes are encoded in external files, which can be version-controlled and bundled with the service.

- **Scalability**: The proxy works in combination with load balancers, reverse proxies or request gateways. The deployment process is decoupled from Bifrost, giving developers the freedom to deploy the application in any fashion and at any scale.

- **Microservices**: Each instance of the proxy is simply another service added to the application. This principle also applies to the Bifrost Engine. The Bifrost Toolkit is therefore easy to embed into an existing application that follows the microservices architecture approach. It is necessary that the Bifrost Engine is able to access the proxies over HTTP to properly configure their routing mechanism.

- **Extendability**: The engine features a flexible DSL interpreter that can easily be extended by introducing new types of actions. Section 4.5.2 provides further details of this implementation.

- **Non-Intrusive**: The approach of dynamic request routing (as introduced in Section 2.2.2) was chosen to realize the live-testing methods. No code has to be added to existing services. The Bifrost Toolkit can run alongside existing applications with ease, supporting any web-based service including databases and external services accessed through HTTP.

### 4.3.3 Technologies

All tools have been developed as web applications using JavaScript as the main programming language. Node.js[3] has been utilized as the server-side runtime, in combination with Babel[4] which is a backwards-compatible JavaScript transpiler that allows the usage of the latest ECMAScript features in unsupported environments.

Node.js was chosen due to its lightweight and efficient architecture that favors event-driven applications, which Bifrost heavily uses due to the asynchronous nature of its release process. In addition, JavaScript makes use of the npm package manager[5] which constitutes the largest package management system currently in use[6], thus providing a wide range of third-party libraries and API integrations that make it possible to easily extend the Bifrost Toolkit in the future and integrate external services.

[2] Jenkins CI: `https://jenkins-ci.org/`
[3] A JavaScript runtime built on Chrome's V8 JavaScript engine: `https://nodejs.org/en/`
[4] BabelJS: `https://babeljs.io/`
[5] npm - what is npm? `https://docs.npmjs.com/getting-started/what-is-npm`
[6] Modulecounts: `http://www.modulecounts.com/`

The communication between the components is handled through RESTful HTTP-APIs that make use of ExpressJS[7], a "fast, unopiniated and minimalist web framework" [Exp16]. Where real-time communication was necessary, *e.g.*, updating the CLI or UI with real-time information, WebSockets[8] using Socket.IO[9] were utilized to implement the features.

## 4.4 Bifrost DSL

A domain specific language is used to implement the previously introduced model. This enables a more transparent process [Fow08] and allows for easier automation. This section describes the process of deriving the DSL from the formalized model and its practical implementation.

### 4.4.1 Converting the Release Model into a DSL

The developed Bifrost release model has been converted into a DSL, using an EBNF [Wir96] syntax model as an intermediate step. This was done to avoid costly changes of the prototype and make sure the language implements the concepts of the Bifrost release model. The EBNF representation has been validated by constructing the various methods of live-testing and combinations thereof to verify its completeness. The presented EBNF in Listing 4.1 represents the final version of the iterative process. Not all elements displayed are completely described, in order to give a better overview. Details about the actions and their specific properties can be found in Section 4.4.4. In order to clarify the used notation and symbols, the following list gives an exempt of the EBNF syntax:

1. $'x'$ denotes a terminal symbol
2. $x$ denotes a non-terminal symbol
3. $x*$ denotes one or more occurrences
4. $x?$ denotes zero or exactly one occurrence

The EBNF syntax model has then been used as a blueprint for the Bifrost DSL. The DSL was built as an internal DSL on top of YAML[10], a data serialization language designed to be readable by humans [BKEI05]. YAML was chosen as it is not bound to a programming language but provides enough expressiveness to model the necessary properties. It also allows the approach and DSL to be reused for further research or prototype development. Additionally, its readability is a plus for people that are not accustomed to programming languages or only know a selection thereof.

[7] ExpressJS: `http://expressjs.com/`
[8] The WebSocket Protocol, RFC 6455: `https://tools.ietf.org/html/rfc6455`
[9] SocketIO: `http://socket.io/`
[10] Recursive acronym for "YAML Ain't Markup Language"

```
release ::= 'name' deployment strategy*
deployment ::= orchestrator service*

orchestrator ::= (Proxy | Docker)

service ::= 'name' 'host' 'port'
strategy ::= 'name' action* next?

action ::= (route | metric | bool | request)* executionWrapper onTrue onFalse

route ::= 'from' 'to' filter*
metric ::= provider* validator
bool ::= (AND | OR)
AND ::= action*
OR ::= action*
request ::= 'url' 'status'

executionWrapper ::= (timedExecution | defaultExecution)
timedExecution ::= 'intervalLimit'? 'intervalTime'? 'threshold'?

onTrue ::= action
onFalse ::= action
```

Listing 4.1: EBNF representation of the Bifrost DSL

### 4.4.2 Deployment

The deployment section of the YAML DSL (an example can be seen in Listing 4.2) allows the Bifrost Engine to modify or leverage the existing deployment situation properly. It is divided into an orchestration and a services part. The orchestration object is used to configure an orchestrator of choice. The orchestrator's responsibilities are to make sure that the engine either knows which proxy belongs to which service, or to supply information about existing infrastructure.

```
---
deployment:
 orchestrator:
  proxy:
    mapping:
      frontend: frontend_proxy
      products: products_proxy
      auth: auth_proxy

 services:
  - name: frontend_a
    host: frontend
    port: 80

  - name: frontend_b
    host: frontend
    port: 80
```

Listing 4.2: Example of a deployment specification in Bifrost

The second part of the deployment section is used to declare the affected services. Services have to be specified by providing a name, host name and a port upon which the service is listening on. Each service that actions would like to refer to has to be specified, including different versions of the same service if operating under another host name.

### 4.4.3 Strategies

The strategies section of the YAML DSL is used to define the sequential order of individual parts of the release. As introduced in Section 4.2.1, one of the key characteristics is the combination of *ordered* and *parallel execution* of steps during releases. A strategy is executed in order of its specification per default. Each strategy consists of a name, a set of actions and an optional link to the strategy that should be executed next. This property can be overridden at runtime by its actions as seen in Listing 4.3. This allows multi-staged releases and live-testing setups, which were identified as a key requirement in Section 4.2.1 to allow for data-driven decisions during releases. There exists no limit how many strategies a release can consist of. It is important to consider though that introducing looping behaviors in release processes can have severe consequences if the release is unable to terminate. The prototype implementation allows such constructs to give engineers the freedom they need in order to construct releases that cycle through some of the strategies at multiple times. The Bifrost release model, upon which this behavior is based, also allows introducing formal ways of checking releases for loops or simulating a release process. This could be used in the future to warn developers about possible implications of their specified release processes.

```
1  ---
2  strategies:
3   - name: ab_test
4     actions:
5       ...
6       onTrue: phased_rollout_a
7       onFalse: phased_rollout_b
8
9   - name: phased_rollout_a
10    actions: ...
11    next: finish
12
13  - name: phased_rollout_b
14    actions: ...
15    next: finish
```

Listing 4.3: Example of multiple strategies that make use of the *next* property

### 4.4.4 Actions

The primary part of the release specification in Bifrost are its actions. Each strategy can contain an arbitrary number of actions that are executed in parallel (in comparison to strategies). This adheres to the characteristic of *parallel execution* as introduced in Section 4.2.1 and formalized in

Section 4.2.2. The prototype described in this thesis implements a set of generic actions that are able to model all specified methods of live-testing as presented previously (see Section 2.2.1).

### Route

The *Route*-Action (see Listing 4.4) allows the configuration of Bifrost Proxies in the context of the current release. Each action requires to specify an outgoing service (*from*) as well as a receiving service (*to*). In order to persist routing behaviors after finishing the action, a *persistence* parameter is available to specify long-term changes in routing. Otherwise, all changes are automatically reverted after the action has been completed. Actual route configuration works using so-called *filters*. Two filters have been written, namely a *header*- and a *traffic*-filter which are further described in Section 4.6.

```
1 ---
2 route:
3   from: A
4   to: B
5   filters:
6     - traffic:
7         percentage: 50
```

Listing 4.4: Route action that redirects 50% of the traffic from service A to service B

### Request

The *Request*-Action (see Listing 4.5) allows calling arbitrary services within reach of the Bifrost Engine. Developers can specify an expected HTTP-Statuscode and the action returns *True* when the received status code matches the expected one or *False* if not. The action can be used for health- or availability checks, *e.g.*, to prevent the execution of multi-staged strategies given that the service failed in the meantime.

```
1 ---
2 request:
3   url: "http://www.google.ch"
4   status: 200
5   intervalTime: 5
6   intervalLimit: 10
```

Listing 4.5: Request action that checks http://www.google.ch once in 5 seconds for 10 times

### Pause

The *Pause*-Action (see Listing 4.6) allows engineers to deliberately pause the execution of releases between strategies. This can be used either for testing purposes, or if certain decisions on how to proceed are not able to be automated. User input is necessary to continue the execution, either using the Bifrost CLI or the Bifrost UI. The action also allows for manual selection of the next strategy.

```
1 ---
2 pause:
```

Listing 4.6: Pause action that stops the release after finishing the running strategy

### Metric

The *Metric*-Action (see Listing 4.7) allows metrics to be pulled in from *MetricProviders*. The prototype implementation currently supports Prometheus[11], and allows querying the provider using its own query language. Results from providers can either be processed using simple validators that allow for comparison of scalar values[12], or by outsourcing the processing for more complex algorithms by specifying an URL. Results collected from the metric provided are then relayed to a HTTP-Service that will determine the output of the metric-action.

```
1  ---
2  metric:
3    providers:
4      - prometheus:
5          name: aCPU
6          query: avg_over_time(container_cpu_system_seconds_total{name='A'}[60s])
7      - prometheus:
8          name: bCPU
9          query: avg_over_time(container_cpu_system_seconds_total{name='B'}[60s])
10
11   validator: aCPU>=bCPU
12   delay: 60
```

Listing 4.7: Metric action that compares CPU-Load using Prometheus as provider

### Boolean AND/OR

To allow for more interesting interactions between individual actions, it is possible to combine actions using boolean logic. Two actions of type *AND* (see Listing 4.8) and *OR* (which works identically) allow chaining multiple actions together and test whether they all, or at least one of them succeeded. This allows developers to model more complex decision trees *e.g.*, using *Metric* or *Request* actions. As any other actions, *AND*- and *OR*-actions can also be used to modify

[11] An open-source service monitoring system: `http://prometheus.io/`
[12] See "Expression language data types": http://prometheus.io/docs/querying/basics/

the following strategy which allows developers to introduce branching paths into their release strategy.

```
---
AND:
  actions:
    - request:
        url: "http://serviceA"
        status: 200
    - request:
        url: "http://serviceB"
        status: 200
  onTrue: ab_test
  onFalse: rollback
```

Listing 4.8: AND action that tests the reachability of two services

## 4.5 Bifrost Engine

The Bifrost Engine is the core part of the Toolkit. Its responsibilities are the orchestration of the Bifrost Proxy and the proper execution of supplied releases. The engine consists of multiple components, each responsible for a specific part of the release execution. The following sections will show how the engine parses the DSL using a custom interpreter and how the previously described actions have been implemented.

### 4.5.1 Overview

The project is structured into the following five components:

- **Engine**: Schedules and manages ongoing and waiting releases.
- **Interpreter**: Creates an instance of a *Release* based on the supplied YAML DSL.
- **Deployment**: Provides connectors to the proxies such that the engine can properly configure them and modify the appropriate routes. Is also able to automatically deploy and undeploy proxies in certain deployment setups.
- **Model**: Contains the actual application logic specific to the executed actions and therefore implements most of the functionality of the Bifrost release model.
- **API**: Provides a REST-API and a socket endpoint to schedule releases and receive data about its current state, used by the CLI and UI.

In the following sections, three of these components are presented in detail to highlight some of the more interesting implementation specific features that were used.

### 4.5.2 Interpreter

Before actually executing a release, the YAML-DSL needs to get interpreted and translated into the internal model representation of the Bifrost Engine. The *Interpreter* parses and creates a *Release* object out of the provided YAML. This makes it possible to check the DSL for missing parameters, filter out invalid structure and prevent basic mistakes that could lead to a corrupted release flow. Every parsed object is checked by a validator to determine whether all necessary parameters are supplied and may cancel the parsing process if necessary.

The different parts of the DSL are internally represented as classes that implement their corresponding functionality. After transforming the input (YAML DSL) into JSON[13], the JSON data has to be converted into a full object-graph.This is done using a parser that recursively traverses the input JSON and creates JavaScript class instances[14]. This abstraction makes it possible to easily extend the DSL in the future, as objects are inflated dynamically using a mapper that links JSON keys to class instances. The interpreter could be modified to prevent breaking changes that a new DSL version introduces.

### 4.5.3 Deployment

Two implementations have been provided:

#### Proxy-Orchestrator

A generic orchestrator that works with a wide range of setups. It takes a list of key-value pairs mapping host names of services to host names of corresponding Bifrost Proxy instances. This orchestrator allows the tool to work in different deployment setups, either in the cloud or when using bare-metal deployments.

#### Docker-Orchestrator

A Docker focused orchestrator that allows the engine to automatically deploy instances of Bifrost Proxies if needed and also undeploy them from the application when the release has finished. As Docker-Networks use host names to communicate internally[15], Bifrost is able to inject itself into the application by spoofing existing services and therefore receiving their traffic. The implementation was focused on Docker <1.9.1 and is subject to change depending on the future development[16].

### 4.5.4 Model

According to the previously presented Bifrost release model in Section 4.2.2, every release consists of strategies that are executed in series and actions that are run in parallel during a strategy. The

[13] Java Script Object Notation
[14] JavaScript Classes: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes`
[15] Networking: `https://docs.docker.com/engine/userguide/containers/networkingcontainers/`
[16] Docker 1.9.1 Changelog: `https://github.com/docker/docker/blob/master/CHANGELOG.md#191-2015-11-21`

model has been used as specified in the implementation of the prototype. An essential part of the functionality that the Bifrost Toolkit provides is encoded in its actions. Every release strategy consists of $1ton$ actions that can run in parallel, *e.g.*, monitoring metrics, modifying the traffic flow, performing tests and altering the release behavior accordingly. The following section explains how the required functionality from the release model has been implemented in the code.

### Actions

Each action extends an abstract *Action* class that provides its methods to overwrite, which can be seen in Listing 4.9. Actions are required to at least implement the *evaluate*-method, which contains the actual business logic a specific action should execute. As an example, the *Route* action has been chosen. Each route action contains a set of filters that can modify the routing behavior of a service. During the *evaluate*-method of this action, the proxies are reconfigured using the filters. Actions have the additional possibility to hook into a pre- and post evaluation method that is executed either before the action starts or right after it finishes. Both methods are able to influence the result of an action. In the described *Route* action, the post-hook is used to remove the filters from the configured proxies if the *persistence*-flag is not set in order to rollback the proxies configuration to its original state.

### ExecutionWrapper

The timed execution behavior of certain actions that allow for multiple runs of the same action has been generalized using a so-called *ExecutionWrapper*. This behavior follows the Decorator-Pattern [Gam95] allowing to change the behavior of an action depending on its given *ExecutionWrapper*. Two wrappers have already been implemented. The *DefaultExecution* simply executes the action once, returning the corresponding outcome of the method. The *TimedExecution* allows developers to specify a delay, an interval, a number of intervals and a threshold. These four properties can be used to model situations where actions are run multiple times but do not have to succeed in all cases in order to mark a successful case. An example of this functionality are extended monitoring periods that wait for thresholds of certain metrics to be surpassed. For example, a developer would like to monitor the amount of exceptions a service produces. It is known that the current implementation of the service produces less than one exception per hour on average. A new implementation is tested, and the developer specifies to monitor the service for a day using an interval of 60 minutes and checking whether more than one exception occurred during that time frame. The threshold then determines, how many checks are necessary to succeed in order to evaluate the action to be true.

```
export default class Action {

   constructor() {
      this.executionWrapper = new DefaultExecution();
      ...
   }

   /**
    * Executes the action considering its executionWrapper.
    * @param {Strategy} strategy
    * @param {Release} release
    * @returns {Boolean}
    */
   async execute(strategy, release) {
      this._startedAt = new Date();
      var result = await this.executionWrapper.execute(this, strategy, release);

      ...

      return result;
   }

  /**
    * Hook that gets executed before the action is actually executed.
    * @param {Strategy} strategy
    * @param {Release} release
    * @returns {boolean}
    */
   async preEvaluate(strategy, release) {
      ...
   }

   /**
    * Hook that holds the actual implementation of what the action does.
    * @param {Strategy} strategy
    * @param {Release} release
    * @returns {boolean}
    */
   async evaluate(strategy, release) {
      ...
   }

   /**
    * Hook that gets executed after the action is completely executed (all runs if there
        are multiple)
    * @param {Strategy} strategy
    * @param {Release} release
    * @returns {boolean}
    */
   async postEvaluate(strategy, release) {
      ...
   }

  ...

}
```

Listing 4.9: Action-class implementation in Bifrost Engine

## 4.6 Bifrost Proxy

The Bifrost Proxy is a standalone Node.js based HTTP-Proxy, which can be configured to route requests to arbitrary HTTP-Hosts depending on configured filters. Each proxy has a default service that requests will be routed to. Depending on the configuration, requests can be routed to arbitrary services that are reachable from the proxy. The proxy functionality has been implemented using the node-http-proxy[17] and supports HTTP as well as secure connections with HTTPS.

### 4.6.1 Configuration

The default service gets specified either by using environment-variables (*HOSTNAME* and *PORT*) or by using starting parameters. As soon as the proxy has been started, a REST API on port 9090 is available that allows the configuration, *e.g.*, through an instance of the Bifrost Engine or using custom configuration mechanisms.

### 4.6.2 Sticky Sessions

Depending on the implementation of the proxied service, it is important that requests from the same users are always routed to the same service instance. This behavior is generally called sticky sessions and one of the basic problems that are solved by load balancers, especially in cloud deployments [Tof12]. Bifrost addresses this problem as well, guaranteeing that during A/B tests users get assigned to a particular service version and will not experience different services handling their requests. The proxy accomplishes this by setting a Cookie on the client using the Set-Cookie Header[18] in its response. The cookie contains an RFC-compliant UUID[19] that is used to identify the client in subsequent requests. If the proxy has been configured to apply certain filters, the usage of the filters are determined uniquely per session and stored by the proxy.

### 4.6.3 Filters

Actual route configuration works using so-called filters. Two filters have been written, namely a *header*- and a *traffic*-filter. As the name implies, the header-filter is used to modify routing behavior based on inspecting request headers. As the proxy currently only supports HTTP, the headers mentioned refer to the header fields specified in RFC 2616 section 14 [FGM+99b]. Apart from standardized fields, every custom-named header field can be used for filtering as well. The traffic-filter can be used to reroute traffic in its entirety. Developers are able to define how much of the traffic should be rerouted, as well as whether the proxy should use sticky sessions to permanently reroute users.

[17] A full-featured http proxy for Node.js: `https://github.com/nodejitsu/node-http-proxy`
[18] HTTP State Management Mechanism: `https://tools.ietf.org/html/rfc6265`
[19] A Universally Unique IDentifier (UUID) URN Namespace: `http://www.ietf.org/rfc/rfc4122.txt`

## 4.7 Bifrost CLI

As most of the work of DevOps engineers happens in shell-based systems, one part of the Bifrost Toolkit is a CLI[20] that allows developers to schedule and execute strategies remotely or as part of release scripts in CI-Systems such as Jenkins. The CLI tool has been built using commander.js[21], a solution that makes it easier to build command-line interfaces for Node.js.

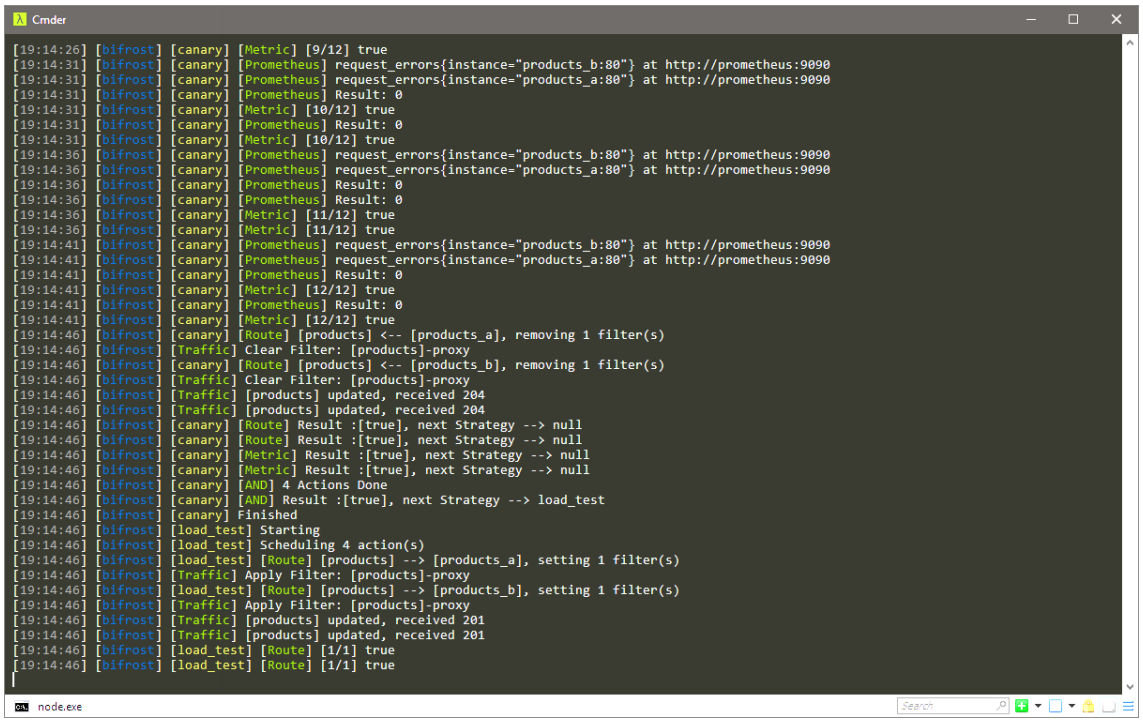



Figure 4.2: The Bifrost CLI provides real-time status information while a release is running

### 4.7.1 Overview

The Bifrost CLI is an easy way to use the deployment and live-testing capabilities of Bifrost. Using *bifrost run*, the CLI checks whether a local *bifrost.yml* file can be found in the folder. Custom filenames are supported as well. The CLI has been tested on Windows and UNIX-based systems alike. While a release is deployed, the CLI outputs real-time status information about the ongoing release (as seen in Figure 4.2). It can also be used to reset the state of individual proxies using *bifrost reset*, in order to restore the original routing should the release process fail.

[20] Command-Line-Interface
[21] commands.js: `https://github.com/tj/commander.js`

## 4.8 Bifrost UI

To give developers easy access to information about the current state of the Bifrost Toolkit, a web application has been developed that visualizes the current state of releases in a graphical user interface.

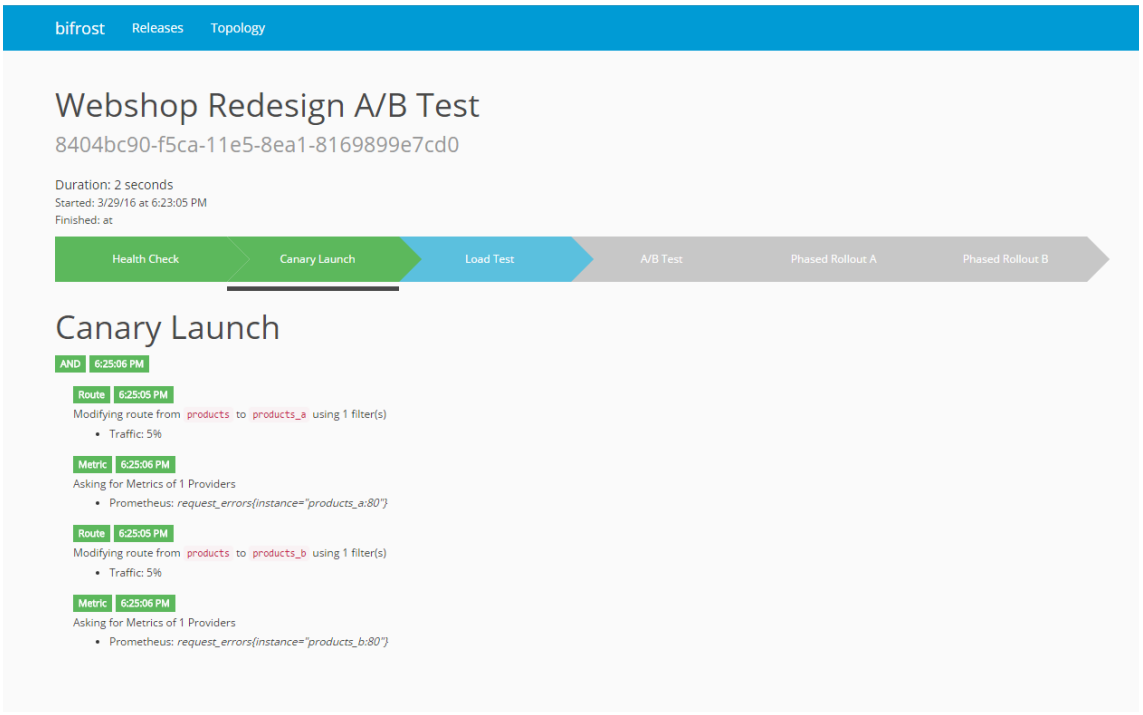



Figure 4.3: Ongoing release in Bifrost UI, updating its progress in real-time

### 4.8.1 Overview

Bifrost UI consists of a backend providing a REST-API and a JavaScript frontend written using the AngularJS[22] framework. The backend-component of the UI automatically connects to the Bifrost Engine, receiving real-time updates about the current release state using WebSockets. These updates are relayed to the AngularJS frontend, which dynamically redraws its interface in order to reflect the current release state. Developers thus have the possibility to check upon individual releases and their progress. Each release can be studied in detail. This includes individual actions, their results and their hierarchical structure (*e.g.*, through boolean actions) including their results and time of execution. This allows developers to easily follow up on a multi-staged release that has a longer time window. Additionally, the UI polls the known proxies for their current routing setup to visualize the traffic flow throughout the application. This makes it easy to detect persistent filters on proxies.

[22] AngularJS - Superheroic JavaScript MVW Framework: `https://angularjs.org/`

# Chapter 5

# Evaluation

The evaluation of the Bifrost Toolkit is divided into two parts. The first part compares the developed prototype to similar tools from research and industry, whereas the second part focuses on a quantitative performance evaluation to determine the overhead of the tool when used in practice.

## 5.1 Qualitative Evaluation

To compare the Bifrost Toolkit and approach to other research and industry tools, a set of dimensions has been determined that are analyzed subsequently. First, the tools are presented and compared concerning their functionality. Second, a set of dimensions are identified and explained. A comparison matrix is developed and the differences between tools are analyzed in detail. Lastly, a conclusion sums up the qualitative evaluation.

### 5.1.1 Tools

A set of tools has been selected for comparison to Bifrost. The selection of tools in the research section represent the current state of the art. Additionally, a number of industry tools have been added to additionally compare Bifrost to systems that are used in production. A more detailed description of the selected tools for comparison can be found in the Section 3.3. To begin with, the tools have been compared in regards of their features to give an overview of the capabilities and usage scenarios. Each method of live-testing presented in Section 2.2.1 has been included as well as an additional dimension that covers the possibility to combine different methods with each other. In comparison to the later (see Section 5.1.2) introduced dimension of *complex releases*, combination means that the tool offers mixed versions of the defined live-testing methods that do not clearly belong to either category or allow the combination of multiple methods into a new one. An example of such a behavior would be a method called "Canary Release" that automatically tests a candidate and deploys it upon success. *Complex releases* however would allow to generically specify this in a two-step process consisting of a canary launch first and a rollout afterwards. The table 5.1 summarizes the findings of this comparison.

| | A/B Testing | Canary Launches | Shadow Launches | Phased Rollouts | Blue/Green Deployments | Combination |
|---|---|---|---|---|---|---|
| GateKeeper | Yes | Yes | Yes | Partial | No | Yes |
| CanaryAdvisor | No | Yes | No | No | No | Yes |
| Vamp | Yes | Yes | No | No | Yes | No |
| Scientist! | Partial | Partial | Yes | Partial | No | Partial |
| ION-Roller | No | Partial | No | Yes | Yes | No |
| *Bifrost* | Yes | Yes | Yes | Yes | Yes | Yes |

Table 5.1: Feature-comparison of analyzed tools

### GateKeeper

GateKeeper [TKV+15] allows a multitude of different release methods. However, part of its functionality is provided by external services that are not specified in detail in the research paper. It is safe to say however, that its architecture also allows for a combination of live-testing methods.

### CanaryAdvisor

CanaryAdvisor [TSM+15] focuses only on canary launches and does not provide support for other methods of live-testing. When used in combination with IBM's Active Deploy [Sni16], it also allows to initiate automatic rollbacks upon failure of canary launches.

### Vamp

Vamp [Mag14] provides support for A/B testing services, albeit without included monitoring. Due to their chosen approach of dynamic routing, canary launches and blue/green deployments are also possible. However, no support for shadow launches or any type of phased rollouts exists. The combination of methods is not supported, as any service can only have one active configuration at any given time.

### Scientist!

Scientist! [jba14] follows an approach that is similar to feature toggles and therefore provides support for a range of methods. In comparison to feature toggles, Scientist! is used to compare two code fragments and their effects with each other. It executes the old and new behavior and

```ruby
require "scientist"

class MyWidget
  def allows?(user)
    experiment = Scientist::Default.new "widget-permissions"
    experiment.use { model.check_user?(user).valid? } # old way
    experiment.try { user.can?(:read, model) } # new way

    experiment.run
  end
end
```

Listing 5.1: Example of how to use Scientist! [jba14]

monitors the outcome for comparison purposes. A basic example taken from their documentation can be seen in Listing 5.1 to clarify its usage. The way Scientist! works considerably limits the number of usage scenarios, as state modifying actions that are not idempotent cannot easily be tested.

#### ION-Roller

ION-Roller focuses on deployment using Docker images. It allows blue/green deployments and phased rollouts. Canary launches are possible with manual monitoring, as it features rollback capabilities upon manual intervention.

Most of the compared tools allow multiple usage scenarios, but rarely offer the possibility to combine multiple methods freely with each other. This is the main distinction from the developed Bifrost Toolkit, that gives developers the flexibility to mix and match methods of live-testing according to their needs and their envisioned release process.

### 5.1.2 Dimensions

The evaluation includes 8 dimensions as listed below. Each dimension can be either classified as fulfilled (yes), partially fulfilled (if not enough information was available or the feature set restricted) or not fulfilled (no).

- **Platform Agnostic**: Determines whether the tool can be used independent of the type of deployment, underlying platform or programming language of the service. Is considered fulfilled if the tool supports certain kind of applications *e.g.*, web-based services on all supported platforms of such applications, partially fulfilled if constraints it relies on system-components such as custom runtimes.
- **Code-Neutral**: Describes the ability to work without modifying the application's source code, thus supporting both existing applications and applications created form scratch. Is

considered fulfilled if the tool does not require the inclusion of tool-specific source code into the application. Partially fulfilled, if the inclusion of the tool happens automatically upon compilation or building the artifact.

- **Performance-Neutral**: Covers whether the tool introduces measurable runtime performance overhead into the application or not.

- **Traceability**: This dimension covers how transparent the release process is to developers. This includes whether the tools allow version-controlled configuration and the observability of its current state.

- **Automated Data-Driven Decisions**: Includes whether the tool is able to use monitoring data, either by collecting it itself or by utilizing third-party tools to decide upon further steps. Is considered fulfilled if the tool utilizes data, either by collecting it itself or through third-party extensions, to influence its run-time behavior without further user interaction.

- **Multi-Service Support**: The dimension is considered fulfilled if the tool can modify multiple services at once and therefore also be used in environments where dependencies between services have to be considered in release strategies.

- **Complex Releases**: Describes the ability to orchestrate multi-staged releases, consisting of more than one live-testing method (as introduced in Section 2.2.1) . Additionally, the tool must allow choosing further live-testing methods based on previous outcomes.

- **Open-Source**: If the tool's source code is openly available under an OSI[1]-approved license, *e.g.*, Apache, GPL or MIT.

### 5.1.3 Analysis and Discussion

#### Platform-Agnostic

Most tools are platform-agnostic, choosing an approach that can work with various different types of applications. GateKeeper uses features toggles [TKV+15] and therefore requires libraries or integration-efforts if used on a new platform. Additionally, it is unclear whether certain parts embedded in the HVVM are mandatory or not. CanaryAdvisor does use third-party tools to collect monitoring data and not directly the services it observes, and therefore works on all types of applications. Vamp requires to use containers, either with Docker[2] or Mesosphere Marathon[3]. ION-Roller works with containers as well, and is tailored to Amazon's PaaS[4] offering. Scientist! only works for Ruby (albeit a similar library also exists for PHP[5]) and is therefore the most restrictive.

[1]Open Source Initiative: `https://opensource.org/licenses`
[2]Docker: `http://docker.com/`
[3]Mesosphere Marathon: `https://github.com/mesosphere/marathon`
[4]Platform-as-a-Service
[5]Scientist!: `https://scientist.readme.io/`

| | Platform-Agnostic | Code-Neutral | Performance-Neutrality | Traceability | Automated Data-Driven Decisions | Complex Releases | Open-Source |
|---|---|---|---|---|---|---|---|
| GateKeeper | Partial | No | Yes | Yes | Yes | Yes | No |
| CanaryAdvisor | Yes | Yes | Yes | No | Partial | No | No |
| Vamp | Partial | Yes | No | Yes | No | No | Yes |
| Scientist! | No | No | Partial | No | No | No | Yes |
| ION-Roller | Partial | Yes | Yes | Yes | No | No | Yes |
| *Bifrost* | Yes | Yes | No | Yes | Yes | Yes | Yes |

Table 5.2: Comparison of live-testing tools

### Code-Neutral

Whether the tool needs code modifications to run depends mainly on the chosen approach and scope of the tool. GateKeeper needs specific code to be embedded in the targeted application and is therefore clearly intrusive, as it mandates the modification of a service that would like to use its functionality. CanaryAdvisor does only monitor, but it relies on specific log aggregators to be embedded in the application. This is similar to Bifrost, however, that also needs third-party providers to aggregate data for its decision-process. Vamp does not require any modifications as it works on service-level, as does ION-Roller. Scientist! has to be embedded into the project's code.

### Performance-Neutrality

In terms of the performance impact, the chosen approach basically mandates whether the tool impacts performance or not. The paper describing GateKeeper does not directly talk about performance-implications. However, long-running metric-tasks are stated to be asynchronously processed in order to keep applications responsive. CanaryAdvisor does only monitor and has therefore no direct impact on the application's performance. Vamp follows a very similar architecture to Bifrost and proxies requests. Therefore, a small hit to performance is to be expected, although no numbers are named. Scientist! adds a performance overhead depending on its usage: If experiments should only be served to a subset of users, this distinction is made every time on every method that gets feature toggled. ION-Roller does not proxy the services directly and can therefore be considered as performance neutral.

### Traceability

Many of the analyzed tools use DSLs to make their configuration approachable and easy for developers to understand. GateKeeper has a restraint-system, which is represented as an internal DSL based on PHP. However, users configure the restraints using a web platform. The UI creates DSL-code that is committed to a version control system to allow proper code reviews, and thus also a transparent history of release-changes. CanaryAdvisor does not use a DSL or any other method at all and seems to rely upon manual configuration. Vamp and Bifrost both use a YAML-based DSL to represent release and service configuration. Scientist! gets embedded directly into the source code. In comparison to GateKeeper, however, there exists no restraint system that is decoupled from the application's code. ION-Roller uses external configuration files encoded in JSON, which is very similar to the YAML-based DSL approach of Vamp and Bifrost. It also features a command line tool to modify the configuration and allows developers to easily observe the current state of releases.

### Automated Data-Driven Decisions

GateKeeper allows for automating the complete process when used in combination with other tools mentioned in the research paper [TKV+15]. IBM's CanaryAdvisor gives recommendations as to whether a new candidate is fit or not by monitoring it and watching for statistically significant changes in behavior (*e.g.*, errors in log files), but does not deploy the service itself. Vamp does not allow for any dynamic behavior, instead its traffic routing has to be configured and contains no means of change over time except for manual adjustment. The same holds true for Scientist! which features monitoring capabilities out of the box, but no way to automatically deploy the new feature. Developers have rather to modify the source code again to remove the feature toggle in order to permanently deploy a certain application behavior. ION-Roller does not take any data into consideration but only allows for phased rollouts. However, in combination with other services (*e.g.*, the Bifrost Toolkit itself) such behavior may be possible.

### Complex Releases

Apart from GateKeeper and Bifrost, no other tool provides means to create multi-staged and complex releases. Part of the reason is that tools either focus on a specific area of the release process (*e.g.*, CanaryAdvisor), or simply do not offer the functionality (Vamp) and chose a static representation of a release.

### Open-Source

GateKeeper is part of a broader tool set developed by Facebook and thus closed-sourced. CanaryAdvisor was developed as a research project by IBM, but is closed-source and integrated into their PaaS offer called BlueMix, where it is currently available under the same name [Sni16]. Vamp is open-source and licensed under Apache 2.0. Scientist! and ION-Roller are open-source as well, both licensed under the MIT License.

### 5.1.4 Summary

The Bifrost Toolkit shows its strength with comparable tools due to the general approach to release methods, allowing developers to use it for different scenarios. The Bifrost release model upon which the prototype is based allows developers to specify non-trivial ways of releasing their software, including a combination of two or more methods. Existing software in research as well as industry lack this kind of flexiblity.

Due to its chosen technique of implementing live-testing methods (*i.e.*, dynamic request-routing) it stays platform-agnostic and code-neutral, allowing its inclusion in existing software without hassle. Vamp follows a very similar approach, with the difference being that it concentrates more on deployment than on releasing software. This focus makes it very difficult to introduce release methods that modify the setup over time, as deployment software is generally interested in creating a stable setup.

The Bifrost Toolkit itself supports a wide range of possible deployment solutions, from containers to bare-metal machines. This gives developers more flexibility and reduces vendor lock-in tremendously, as it is not coupled with specific IaaS[6] providers. However, the same approach that gives the prototype its flexibility also introduces performance penalties as the dynamic request-routing adds additional services into the application's landscape. These concerns are addressed in the performance evaluation in Section 5.2.

[6] Infrastructure as a Service

## 5.2 Quantitative Evaluation

The Bifrost Toolkit provides developers with a flexible and general approach to introduce live-testing methods into their software releases. However, the feasibility of this approach is influenced by how much the Bifrost Toolkit negatively impacts application performance. As the Bifrost Proxy is a mandatory component to provide routing and filtering capabilities, its impact on application performance has been evaluated using a number of performance tests.

### 5.2.1 Method

To properly test the setup, a sample application simulating a generic microservices application was necessary. Unfortunately, only very few sample applications exist that are able to be deployed in a non-monolithic way, such as Acme Air[7]. In contrast to their claim, the microservices version of the tool only separates the authentication service from the rest of the application, which is still deployed monolithically. A fixed version exists in form of NetflixOSS Acme Air[8] that was extended by leveraging Netflix OSS[9] tools. In comparison to the NodeJS version of the Acme Air Sample, it uses WebSphere[10] and does not provide a containerized deployment setup that would allow for an easy integration of the Bifrost Toolkit at this stage. Another sample application also serves as a testbed to demonstrate Netflix OSS tools but only consists of 3 services[11]. To remedy these issues, a NodeJS based sample application was developed to run performance tests against.

#### Bifrost Microservices Sample Application

The application simulates a generic e-commerce website selling goods. It consists of different services that handle authentication, data storage and querying, search functionality and the delivery of the frontend. It was kept simple in order to provide a testbed for the performance evaluation and demonstration of the capabilities of the Bifrost Toolkit. An overview of the architecture is provided in Figure 5.1. The application consists of 7 services in total, whereas 4 were developed for this application (marked in bold):

- **Frontend**: HTML/JavaScript frontend using AngularJS.
- **Products**: HTTP-based REST-API that allows querying products, detail information and placing buy orders.
- **Search**: HTTP-based REST-API that allows to search for products using text-queries.
- **Auth**: HTTP-based REST-API that authenticates and authorizes users based on their provided e-mail and password and validates tokens.
- *MongoDB*: Document-based database for storage of users and products.

[7] Acme Air Node.js: `https://github.com/acmeair/acmeair-nodejs`
[8] NetflixOSS Acme Air: `https://github.com/aspyker/acmeair-netflix`
[9] Netflix OpenSource: `https://netflix.github.io/`
[10] IBM WebSphere Application Server: `http://www.ibm.com/software/websphere`
[11] Netflix OSS RSS Recipes: `https://github.com/Netflix/recipes-rss`

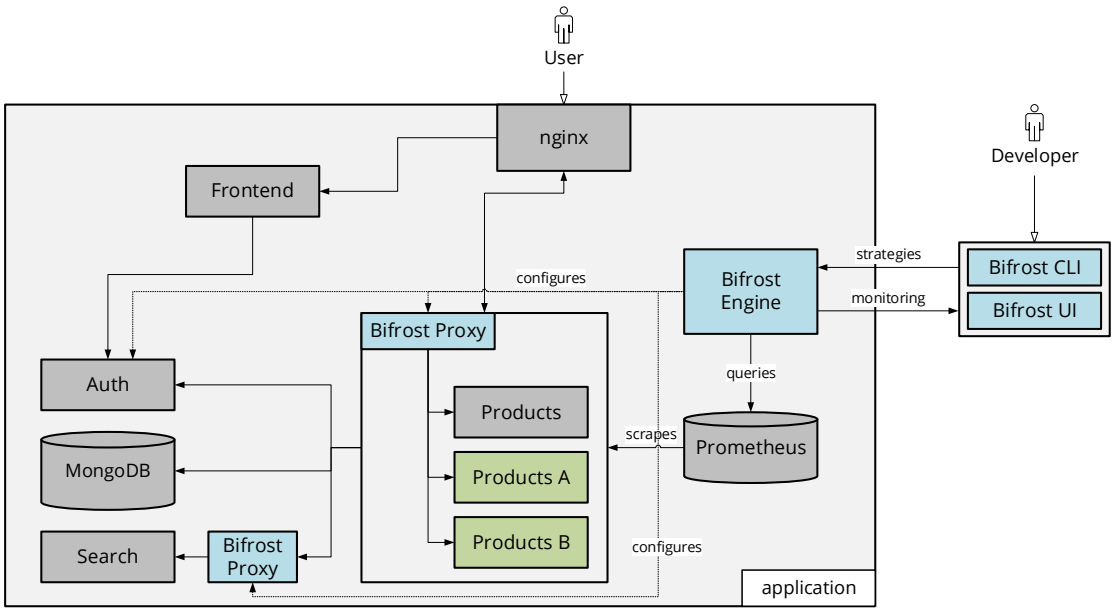



Figure 5.1: Bifrost microservices sample application as used in the performance evaluation

- *Prometheus*: Collects container and low-level performance metrics as well as business metrics from services that expose them.
- *nginx*: A reverse-proxy used as a central entry-point to the application for real users. It is used to proxy incoming requests to either the frontend service or to the products service.

## JMeter Test Suite

To conduct performance or load tests, a test suite was created to enable automated and repeatable testing. The test suite was made using Apache JMeter[12] and simulates standard user behavior. As JMeter is not browser-based, only HTTP-endpoints can be tested. No site rendering or JavaScript execution is performed. The performance tests are not influenced by this, as the proxied services are merely used as dependencies by the frontend. The test suite targets the *Products* service and consists of 4 different requests that touch different parts of the system.

- **Buy**: POST-Request to the *Products*-service, intends to buy a product. Write-operation on database. No response body. Authorized by *Auth*-service.
- **Details**: GET-Request to the *Products*-service, returns information about a single product. Read-operation on Database. Small response body. Authorized by *Auth*-service.
- **Products**: GET-Request to the *Products*-service, returns a list of all products. Read-operation on database. Large response body. Authorized by *Auth*-service.

[12]JMeter - An open-source load-testing tool written in Java: `http://jmeter.apache.org/`

- **Search**: GET-Request to the *Products*-service, which in turn asks the *Search*-service. Read-operation on database. Small response body. Authorized by *Auth*-service.

There exists no general recommendation in research on how much load makes for a good test. Parse[13] states that 30 requests per second for their service translate roughly to 250'000 daily active users if spread even throughout the day [Par16]. Other sources assume that 70 requests per second average to 25'000 unique visitors per hour [Ser14] which corresponds to 300'000 daily active users. This thesis uses 50 requests per second unless stated differently to simulate a realistic load scenario that can be supported sufficiently by the used hardware.

## General Setup

All tests were performed on a Intel Core i5-3570K CPU with 16GB of RAM, in a virtual machine running on VirtualBox 5[14] that was allocated 4GB of RAM and 4 Cores. The virtual machine uses boot2docker[15] as operating system, a Linux distribution that brings native container support with a minimal footprint.

[13] An open source cloud backend from Facebook: `https://www.parse.com`
[14] VirtualBox: `http://www.virtualbox.org`
[15] boot2docker: `https://github.com/boot2docker/boot2docker`

### 5.2.2 Request Performance

This part of the quantitative evaluation focuses on the performance impact of the proxy upon individual requests when no filters are applied, with various request methods and response sizes.

#### Test Setup

To determine the idle overhead of the proxy, a load-test using approximately 50 requests per second was conducted without running any release strategies. In order to eliminate influences on the overhead by the machine's performance limitations, the load was chosen deliberately on the lower side. The test had a runtime of 60 seconds and was conducted $n = 5$ times per request and proxy inclusion.


Figure 5.2: Average response time per tested endpoint

#### Results

Figure 5.2 shows the average response time per request with and without proxy. The proxied deployments showed consistent performance throughout all tested requests. A single proxy instance adds 1.71 to 5.63ms to the average response time, depending on the type of request. The *Search* request got slower by +5.63ms, which can be easily explained by the fact that the request passes two proxies. A request to search for products is handled by the products service, but gets

internally relayed to the search service which is proxied itself (as visible in the architecture diagram in Figure 5.1). The search service was deliberately proxied as well, to explore whether multiple proxies in the system create unintended side effects. As a consequence of the chosen approach, if strategies and route modifications should encompass multiple services (*e.g.*, letting a hypothetical service Products C only query Search C) the Bifrost Proxy needs to be deployed in front of all involved services to properly re-route the traffic. In comparison to the *Buy* (+2.5ms) or *Products* request (+2.45ms), the *Details* request only got slower by 1.71ms on average. This gap can be explained due to the fact that the request and response sizes are different. Whereas the POST-request has a request body that the proxy has to relay, the *Products* GET-request has a larger response payload.

| Request | Type | Mean | +/- | SD | Min | Max | Median |
|---|---|---|---|---|---|---|---|
| Buy /POST | Proxied | 13.64 | +2.5 | 16.82 | 5 | 252 | 8 |
| | Unproxied | 11.14 | | 16.11 | 4 | 201 | 6 |
| Details /GET | Proxied | 11.58 | +1.71 | 16.33 | 5 | 269 | 7 |
| | Unproxied | 9.9 | | 16.33 | 4 | 215 | 6 |
| Products /GET | Proxied | 32.25 | +2.45 | 20.82 | 14 | 281 | 27 |
| | Unproxied | 29.80 | | 21.45 | 12 | 339 | 24 |
| Search /GET | Proxied | 31.69 | +5.63 | 25.23 | 10 | 340 | 25 |
| | Unproxied | 26.06 | | 23.89 | 7 | 267 | 19 |

Table 5.3: Results of request performance test in milliseconds

The average response time, measured over all types of requests, without using a proxy was 19.32ms, whereas the response time using the Bifrost Proxy was 22.28ms. All request types are significantly slower than their unproxied counterparts ($p < 0.05$), albeit only by a small margin. The hit on performance introduced by the Bifrost Proxy during idle is generally very small and predictable. Whether proxied or unproxied, the standard deviation remained similar which means that the proxy had no impact on the overall response time distribution, which can also be observed in Figure 5.3 and from Table 5.3. The following conclusions can be made:

- The Bifrost Proxy does add a measurable, albeit small performance overhead to the application.
- One proxy instance adds approximately 2.4ms of delay.
- The size of request and response can influence the delay of a Bifrost Proxy instance.
- When deploying multiple proxies into the application landscape, a linear decrease in performance is to be expected.

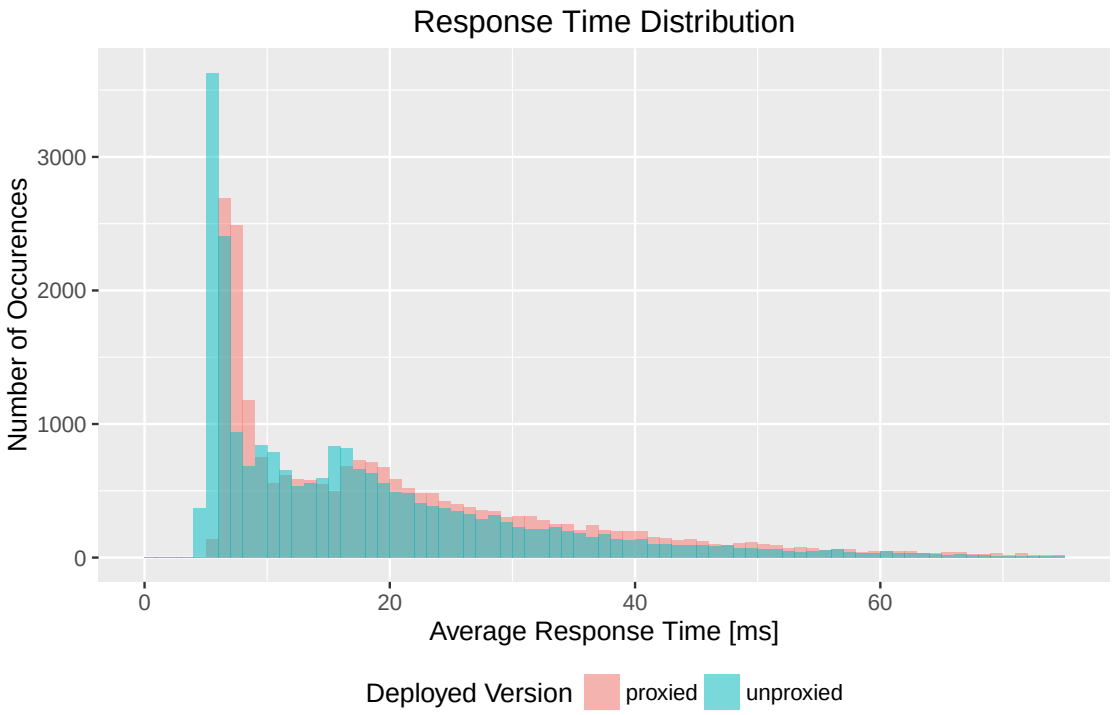



Figure 5.3: Histogram of response times, depicting both proxied and non-proxied

### 5.2.3 Filter Performance

To determine the performance implications of various live-testing methods, 4 filter configurations have been applied to the proxy and been tested with increasing load.

#### Test Setup

The test suite was set to scale from 0 to 120 parallel threads over a timespan of 120 second. The average number of requests scaled from 0 requests per second to 150 requests per second. In addition to the 4 filter configurations two baseline measurements have been collected. The following runs have been conducted, each for $n = 5$ times:

- **Baseline**: Only the sample application is deployed.
- **Proxied**: Bifrost plus the sample application are deployed. No filters are configured.
- **Random A/B Testing**: Traffic filter with 50:50 split.
- **Fixed A/B Testing**: Traffic filter with 50:50 split, sticky sessions[16].
- **Canary Launch**: Header filter that applies for 5% of users
- **Shadow Launch**: 100% traffic filter that duplicates traffic to another service

[16]The proxy assigns sessions to individual users to allow for fixed filter allocation. See Section 4.6

## Results

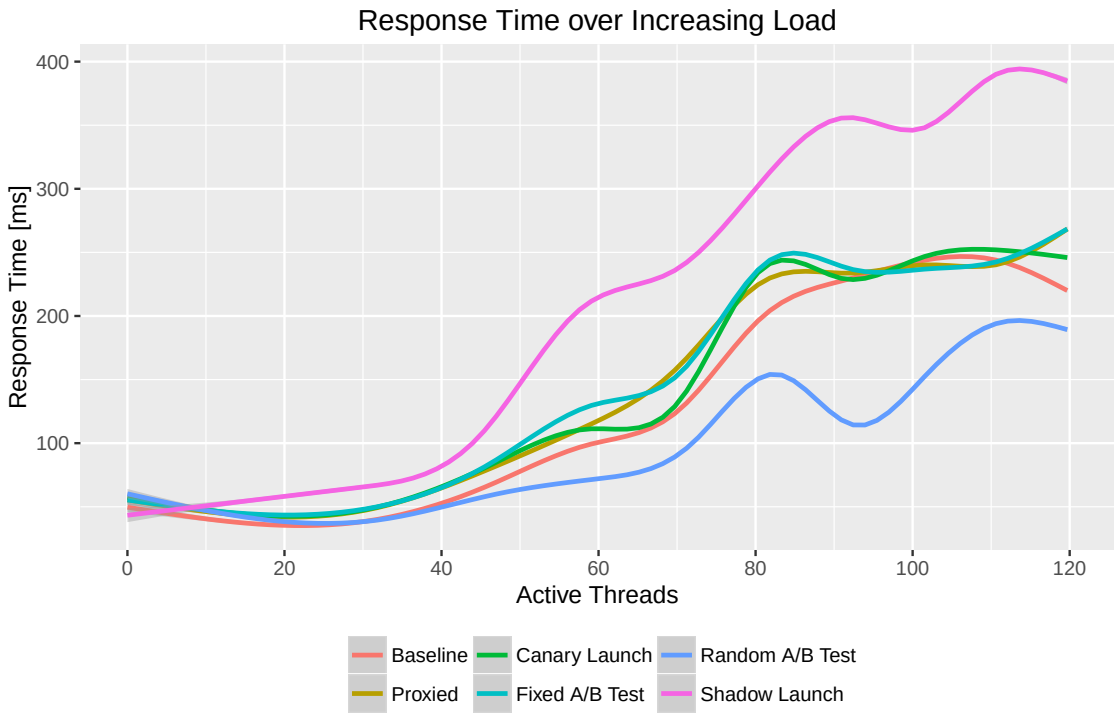



Figure 5.4: Achieved response time while scaling from 0 to 120 threads

The results show that the proxy, including its various filter configurations, scales mostly linear with increasing load. A number of observations are obtained from the test:

- Random A/B testing is clearly faster than fixed A/B testing. This indicates that sticky-sessions add a performance overhead which are negated in the case of random A/B testing as the load is split between two backend services and thus effectively load-balanced.
- Canary launches (and thus header-filtering) show similar performance development as running the proxy without any filter.
- Shadow launches perform well until a certain threshold of active threads is reached, where the configuration increased the response time noticeably. This hints at the proxied service being performance-limiting, rather than the proxy itself.

The filter performance test indicates that the limiting factor is not the proxy itself but rather the services being proxied. Another observation is the fact that shadow launching services can have severe consequences in applications if the increased load is not handled, and should be used with caution. Load-testing new services in production also strains services they depend upon, *e.g.*, databases.

| Active Threads | 0-20 | 20-40 | 40-60 | 60-80 | 80-100 | 100-120 |
|---|---|---|---|---|---|---|
| Baseline | 38.08 | 41.54 | 77.70 | 137.67 | 224.23 | 238.39 |
| Proxied | 43.70 | 49.75 | 91.55 | 165.77 | 234.10 | 245.76 |
| Canary Launch | 45.44 | 50.07 | 93.80 | 150.01 | 236.31 | 249.39 |
| Fixed A/B Test | 44.54 | 52.02 | 100.07 | 167.18 | 241.83 | 244.21 |
| Random A/B Test | 44.07 | 41.05 | 63.74 | 106.80 | 127.40 | 181.47 |
| Shadow Launch | 52.28 | 68.55 | 145.33 | 245.77 | 348.84 | 376.06 |

Table 5.4: Average response time in milliseconds grouped by active threads

### 5.2.4 Release Performance

Lastly, the Bifrost Toolkit was put to the test to evaluate its performance impact on running applications during ongoing release strategies that include various live-testing methods.

#### Test Setup

To show the performance development over a complete release cycle, two new service versions get deployed over a timespan of 380 seconds. The release consists of multiple phases:

1. **Canary Launch**: Tests *Product A* and *Product B* Service while monitoring for errors. 5% of the traffic gets redirected to A and B respectively, and an aggregated error count from Prometheus is monitored. Lasts for 60 seconds.
2. **Shadow Launch**: *Product A* and *Product B* receive 100% of all original traffic to the *Product*-service, while their CPU load is being monitored. Should they experience abnormal CPU load (>80%) an automatic rollback is initiated. Lasts for 60 seconds.
3. **A/B Test**: Routes 50% to *Product A* and 50% to *Product B*. Monitors their sales performance over 60 seconds. Uses sticky-sessions. Reverts the traffic distribution to the original *Product* service after completion.
4. **Phased Rollout**: Rolls out the winner from the previous A/B test from 5% Traffic to 100%, increasing traffic 5% every 10 seconds over 200 seconds.

During the time of the release, normal traffic was simulated using the JMeter test suite and its results were collected. The load consisted of 30 parallel threads. On average about 50 requests per second were directed against the system. The following runs have been conducted:

- **Baseline**: Run without Bifrost deployed.
- **Proxied**: Run with Bifrost deployed, but without executing a release.
- **Running**: Run with Bifrost deployed, executing the aforementioned release.

The data gathered from $n = 5$ runs was combined and aggregated using a moving average, which averages collected data-points from 1 second.

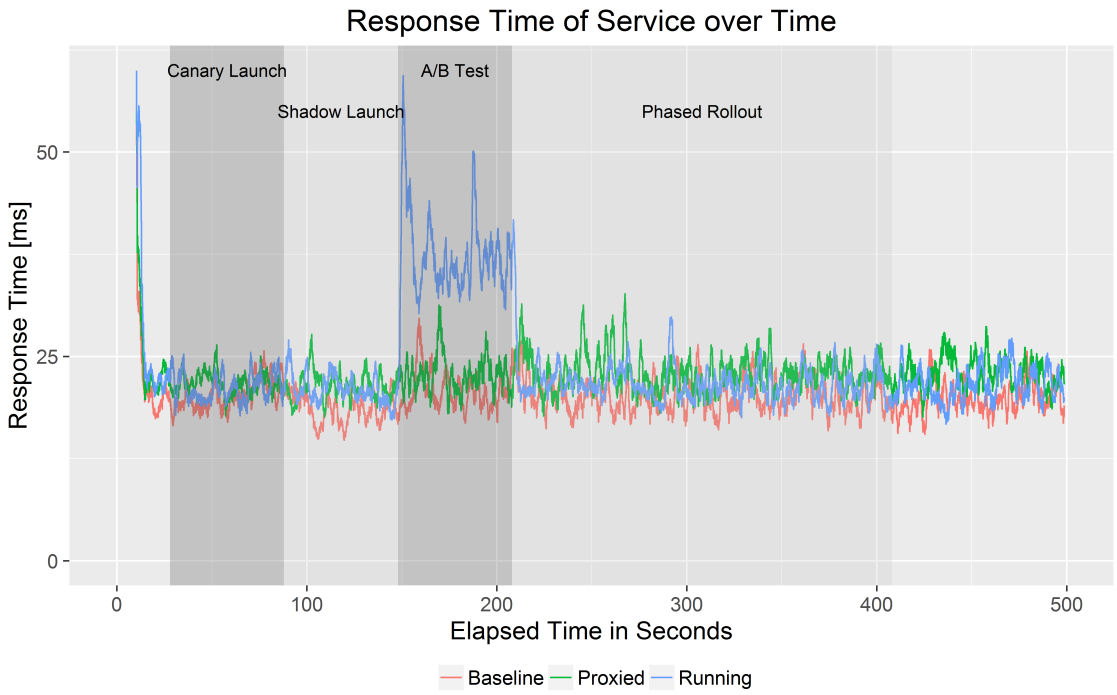



Figure 5.5: Average response time during the release test in milliseconds

## Results

The plot in Figure 5.5 shows the development of the average response time during the release. The different release strategies have been marked in the plot for better distinctiveness. Additionally, Table 5.5 shows the average response time for specific release phases. There are a number of observations that will be discussed in the following:

- **Canary Launch**: The performance of canary launching is consistently similar to running the proxy itself. The average response time is 0.26ms faster during the canary launch, which can be attributed to normal performance variation. The measured delay of 0.91ms between the baseline and the proxied run is consistent with earlier results (see Section 5.2.2).
- **Shadow Launch**: The performance of shadow launching is similar as well. The average response time is 0.53ms faster during the shadow launch, which again can be attributed to normal performance variation.
- **A/B Testing**: All runs show an increase in response time right after switching to the A/B testing phase. The A/B testing adds 17.01ms to the average response time in comparison to the baseline version. This seems contradictory to the results obtained in Section 5.2.3, where fixed A/B testing did not noticeably influence the performance. However, in comparison to the filter performance test the traffic was redirected to two services, each receiving 50% traffic whereas in the filter performance setup, only 50% was redirected to an alternative service. Another possible explanation for this behavior is the sudden introduction of a new

filtering mechanism to an existing set of Bifrost sessions, where the proxy has to quickly assign users to their matching session, whereas the filter performance test slowly added more users.

- **Partial Rollout**: During the last phase while partially rolling out the new service, no performance difference is visible. This observation is in line with previous results from the filter performance test in Section 5.2.3, where randomized A/B tests (which use the same mechanism) did not negatively influence the applications performance.

| Type | Canary Launch | Shadow Launch | A/B Testing | Partial Rollout |
|---|---|---|---|---|
| Baseline | 20.28 | 18.67 | 20.74 | 20.28 |
| Proxied | 21.72 (+1.44) | 21.34 (+0.6) | 22.79 (+2.51) | 23.05 (+2.77) |
| Running | 21.19 (+0.91) | 21.06 (+0.32) | 37.29 (+17.01) | 21.76 (+1.48) |

Table 5.5: Average response time during a phase in milliseconds and delta to the baseline

The results show that certain release strategies can negatively influence application performance. When fixed A/B tests apply sticky sessions, the performance drop is noticeably. The functionality assigns users to an alternative service for the current and all future requests. This behavior is the most complex case the proxy can handle from a technical perspective, which explains the introduced latency. However, a generally acceptable response time for user interfaces is <100ms [Nie94], a value which was never exceeded during the release process.

# Chapter 6

# Final Remarks

This chapter summarizes the contribution, presents a final conclusion and shows how future work could advance the work presented in this thesis.

## 6.1 Conclusion

This thesis focused on software releases and how data-driven strategies can be automated in order to allow their integration into the deployment pipeline of continuously developed applications. Chapter 2 introduces important concepts for continuous software development such as continuous integration, delivery and deployment. Furthermore, a set of wide-spread live-testing methods is presented and a short overview of modern web applications using the microservices architecture approach is given. The collected background information serves as a basis to design a generic model for data-driven releases, as stated in the first research question:

> **RQ1: How can we formalize a (generic) model for data-driven release and deployment strategies?**

Chapter 4 starts with a problem analysis and derives a usage scenario, depicting a non-trivial data-driven release process. Combining this scenario with the methods of live-testing introduced in Section 2.2.1, a set of four characteristics is derived that are key to the methods presented. These are formalized in a generic model that describes the process of data-driven software releases. The characteristics are incorporated as follows: ordered execution through strategies, parallel execution and data-drivenness through actions. The constructed model also incorporates a sense of state by allowing actions to rearrange the strategies in order to repeat phases of a release if necessary. In order to validate the derived model and learn how to further improve it, a prototype called Bifrost was built (described in Chapter 4) and evaluated (in Chapter 5) to answer the second research question:

> **RQ2: How can we build a tool that supports and automates data-driven release strategies for microservices-based architectures in a non-intrusive way?**

The prototype focuses on a specific set of applications, namely web services built using a microservices architecture. The model has been transformed into a YAML-based domain specific language that fully implements its characteristics. The prototype features a command-line

interface, making it it possible to fully automate a complex release procedure during a regular deployment process. It is non-intrusive in the way that the chosen technical approach of dynamic-request routing is able to provide various application versions without source code modification [LBR09]. In order to allow for data-driven releases, the prototype features the inclusion of metrics through third-party aggregators and providers.

In the aforementioned evaluation, the Bifrost Toolkit is qualitatively compared with existing tools from industry and research. It is shown that the chosen approach and the flexible model provide a wider range of functionality and flexibility to developers in comparison to contemporary tools. Furthermore, the Bifrost Toolkit is applicable to a wide range of possible deployment solutions ranging from bare-metal machines to virtualized cloud-environments.

In a second part, a quantitative performance evaluation focuses on the approach chosen by the prototype to implement the model. Although the Bifrost Toolkit gains advantages such as platform-neutrality, performance concerns exist that decide whether the approach is truly non-intrusive. The results show that the Bifrost Toolkit, albeit being a prototype, performs very well. However, it does add a measurable delay to the tested sample application. While the introduced performance hit is marginal, it also varies depending on the chosen live-testing method. Especially methods that rely on sticky-sessions suffer from a greater performance hit that needs further investigation and fine-tuning. In addition, the performance of the proxy shows a variation between different kinds of requests, most likely depending on their request and response size. Further data is necessary however, to completely explain the behavior.

### 6.1.1 Threats to Validity

The following section discusses threats to validity, especially concerning the results of the performance evaluation.

#### External Validity

A few of the tools involved in the performance evaluation are not deemed production ready, such as Docker-Compose[1] where its developers state that it "may be used for smaller production deployments, but is probably not yet suitable for larger deployments". A similar concern is introduced by using the boot2docker[2] linux distribution, which could influence the performance negatively as it is probably not optimized on performance as various other Linux server distributions.

#### Internal Validity

The performance evaluation was conducted on a desktop computer. Even if the machine used is relatively powerful, it is possible that its performance influenced the results of certain measurements. However, the general conclusions are still valid as hitting the performance limit of the machine will degrade the performance not only of the Bifrost services but also of the sample

[1] Docker-Compose in production: `https://docs.docker.com/compose/production/`
[2] boot2docker: `http://boot2docker.io/`

application. Another possible threat are performance fluctuations, which are addressed by running the tests a multitude of times and combining the results to filter out abnormal behavior. The JMeter test suite was constructed to simulate a real user's behavior by including waiting times between requests. Due to the restricted nature of the sample application, a real application could behave differently depending on its architecture and setup.

## 6.2 Future Work

In the following section possible additions and improvements to the model and developed prototype are discussed.

### 6.2.1 Formal Verification

The formalized release model allows for interesting approaches to verify that no deadlocks or loops occur during the execution of a release process. Such a formal verification is possible and could prevent errors introduced by developers while specifying release strategies.

### 6.2.2 Extend Bifrost Toolkit

The presented prototype has a limited set of features that could be extended and advanced in the future. While Prometheus serves as a first implementation of a metrics provider, developers that do not use it are forced to either feed their metrics into Prometheus or implement their own provider. While extending the system itself is one solution, a flexible plugin system would be of bigger use for the future. This would allow developers to integrate either third-party services or their own metric services with ease and without modification of the Bifrost Engine. The YAML-DSL and the written interpreter would allow for the inclusion of such a system with reasonable effort.

### 6.2.3 Integration in Deployment Pipeline

The evaluation conducted in this thesis focused on the tool in general, comparing it to numerous other prototypes and existing work. Additionally, one of the primary concerns regarding performance implications was addressed. An interesting and yet to be explored direction would be to test and evaluate what the toolkit is still missing in order to be properly integrated into a fully automatic work flow of a deployment pipeline. Work towards this goal has already been done by providing a CLI and APIs to control the Bifrost Engine. It is possible though, that a more fine-granular interface is necessary in order to for example allow for production-safe deployment. Also, certain challenges need to be addressed such as how developers can distinguish between versions that should simply be deployed directly (*e.g.*, a small bugfix release) vs. versions that should undergo a sophisticated data-driven release process.

### 6.2.4 Feature Toggles

Whereas the developed prototype uses dynamic request routing, another technique to realize data-driven releases are feature toggles as introduced in Section 2.2.2. The Bifrost Toolkit could potentially be expanded upon adding mechanisms of existing feature toggle libraries, or implementing the functionality by its own. While certain advantages would be lost, the combination of having feature toggles and dynamic request routing at disposal would allow for a fine-granular choice of methods based on the use case.

# Bibliography


[Abr08] P. Abrahamsson. *Agile Processes in Software Engineering and EXtreme Programming: 9Th International Conference, XP 2008, Limerick, Ireland, June 10-14, 2008 : Proceedings*. Lecture notes in business information processing. Springer, 2008.

[Aga11] Puneet Agarwal. Continuous scrum: Agile management of saas products. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pages 51–60, New York, NY, USA, 2011. ACM.

[AM16] Bram Adams and Shane McIntosh. Modern release engineering in a nutshell - why researchers should care. In *Proc. of the 23rd Intl Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, page To appear, 2016.

[Apa13] Apache. mod_proxy - apache http server. `http://httpd.apache.org/docs/2.0/mod/mod_proxy.html#forwardreverse`, 2013. (Accessed on 03/23/2016).

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004.

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001.

[BCG+10] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 47–52, New York, NY, USA, 2010. ACM.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[Bir11] Jim Bird. Feature toggles are one of the worst kinds of technical debt - dzone devops. `https://dzone.com/articles/feature-toggles-are-one-worst`, 08 2011. (Accessed on 03/07/2016).

[BKEI05] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml) version 1.1. *yaml. org, Tech. Rep*, 2005.

[BKK00] H. Bryhni, E. Klovning, and O. Kure. A comparison of load balancing techniques for scalable web servers. *IEEE Network*, 14(4):58–64, Jul 2000.

[Boo91] G. Booch. *Object Oriented Design: With Applications*. The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings Pub., 1991.

[BYV+09] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009.

[CCP99] Valeria Cardellini, Michele Colajanni, and S Yu Philip. Dynamic load balancing on web-server systems. *IEEE Internet computing*, 3(3):28, 1999.

[Che15] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *Software, IEEE*, 32(2):50–54, Mar 2015.

[CSA15] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software technology*, 57:21–31, 2015.

[Cun09] Ward Cunningham. Integration hell. `http://c2.com/cgi/wiki?IntegrationHell`, 2009. (Accessed on 02/25/2016).

[Dea07] Alan Dearle. Software deployment, past, present and future. *2007 Future of Software Engineering*, pages 269–284, 2007.

[Dev09] DevOpDays. Ghent 2009 program. `http://www.devopsdays.org/events/2009-ghent/program/`, 2009. (Accessed on 02/23/2016).

[DMG07] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Patterns and Anti-Patterns*. Pearson Education, 2007.

[EHN14] Frossie Economou, Joshua C Hoblitt, and Pat Norris. Your data is your dogfood: Devops in the astronomical observatory. *arXiv preprint arXiv:1407.6463*, 2014.

[Exp16] Express. Express - node.js web application framework. `http://expressjs.com/`, 04 2016. (Accessed on 04/01/2016).

[FF05] M. Fowler and M. Foemmel. Continuous integration, http://www.martinfowler.com/articles/continuousintegration.html, 2005.

[FFB13] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. Development and deployment at facebook. *IEEE Internet Computing*, (4):8–17, 2013.

[FGM+99a] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999.

[FGM+99b] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999.

[FL14] Fowler and Lewis. Microservices - a definition of this new architectural term. http://martinfowler.com/articles/microservices.html, 03 2014. (Accessed on 03/14/2016).

[Fow08] Martin Fowler. Dslqanda. http://martinfowler.com/bliki/DslQandA.html, 09 2008. (Accessed on 03/10/2016).

[Fow10a] Martin Fowler. Bluegreendeployment. http://martinfowler.com/bliki/BlueGreenDeployment.html, 03 2010. (Accessed on 03/02/2016).

[Fow10b] Martin Fowler. Featuretoggle. http://martinfowler.com/bliki/FeatureToggle.html, 10 2010. (Accessed on 03/01/2016).

[FS14] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering and beyond: Trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, pages 1–9, New York, NY, USA, 2014. ACM.

[FY00] Brian Foote and Joseph Yoder. Big ball of mud. pattern languages of program design 4. *HarrisonN, FooteB, RohnertH (eds.). Addison-Wesley: Reading, MA*, 2000.

[Gam95] Erich Gamma. *Design Patterns: Elements of reusable Object-Oriented Software*. Pearson Education India, 1995.

[GEL15] O. Gunalp, C. Escoffier, and P. Lalanda. Rondo: A tool suite for continuous deployment in dynamic environments. In *Services Computing (SCC), 2015 IEEE International Conference on*, pages 720–727, June 2015.

[HF10] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010.

[Hod16] Pete Hodgson. Feature toggles - a toggling tale. http://martinfowler.com/articles/feature-toggles.html, 02 2016. (Accessed on 03/01/2016).

[HSVT12] T. Hobfeld, R. Schatz, M. Varela, and C. Timmerer. Challenges of qoe management for cloud applications. *IEEE Communications Magazine*, 50(4):28–36, April 2012.

[Hüt12] Michael Hüttermann. *DevOps for Developers*. Apress, 2012.

[jba14] jbarnette. github/scientist: A ruby library for carefully refactoring critical paths. https://github.com/github/scientist, 11 2014. (Accessed on 03/17/2016).

[Joh10] Willis John. What devops means to me | chef blog. https://www.chef.io/blog/2010/07/16/what-devops-means-to-me/, 07 2010. (Accessed on 03/23/2016).

[KLSH09] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M. Henne. Controlled experiments on the web: Survey and practical guide. *Data Min. Knowl. Discov.*, 18(1):140–181, February 2009.

[Lad12] Mohamad Ibrahim Ladan. Web services metrics: A survey and a classification. *Journal of Communication and Computer*, 9(7):824–829, 2012.

[LBR09] D. Lipscomb, C.T. Blum, and T.R. Rice. Phased rollout of version upgrades in web-based business information systems, July 7 2009. US Patent 7,558,843.

[Mag14] Magnetic.io. Vamp :: The very awesome microservices platform. `http://vamp.io/documentation/using-vamp/blueprints/`, 2014. (Visited on 10/20/2015).

[Mar03] Robert Cecil Martin. *Agile Software Development - Principles, Patterns, and Practices*. Prentice Hall PTR, 2003.

[Mar07] M. Marschall. Transforming a six month release cycle to continuous flow. In *Agile Conference (AGILE), 2007*, pages 395–400, Aug 2007.

[McK15] Joe McKendrick. Are microservices for real, or just the latest buzzword? | zdnet. `http://www.zdnet.com/article/a-few-words-about-microservices/`, 02 2015. (Accessed on 03/14/2016).

[MKA+13] Mika V Mantyla, Foutse Khomh, Bram Adams, Emelie Engstrom, and Kim Petersen. On rapid releases and software testing. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 20–29. IEEE, 2013.

[ND14] Sneps-sneppe Manfred Namiot Dmitry. On micro-services architecture. *International Journal of Open Information Technologies*, 2, 2014.

[New15] Sam Newman. *Building Microservices*. " O'Reilly Media, Inc.", 2015.

[Nie94] Jakob Nielsen. *Usability engineering*. Elsevier, 1994.

[NS13] S. Neely and S. Stolt. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *Agile Conference (AGILE), 2013*, pages 121–128, Aug 2013.

[OAB12] Helena Holmström Olsson, Hiva Alahyari, and Jan Bosch. Climbing the" stairway to heaven"–a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 392–399. IEEE, 2012.

[Par16] Parse. Parse | faq. `https://parse.com/faq`, 2016. (Accessed on 03/25/2016).

[Pul13] Ville Pulkkinen. Continuous deployment of software. *Cloud-Based Software Engineering*, pages 46–52, 2013.

[RHWP15] A.A.U. Rahman, E. Helms, L. Williams, and C. Parnin. Synthesizing continuous deployment practices used in software development. In *Agile Conference (AGILE), 2015*, pages 1–10, Aug 2015.

[Rig14] RightScale. Continuous integration and delivery in the cloud: How rightscale does it. `http://www.rightscale.com/blog/cloud-management-best-practices/continuous-integration-and-delivery-cloud-how-rightscale-does-it`, 10 2014. (Accessed on 03/21/2016).

[San15] Peter Sankauskas. The dos and don'ts of blue/green deployment - cloudnative. `http://cloudnative.io/blog/2015/02/the-dos-and-donts-of-bluegreen-deployment/`, 04 2015. (Accessed on 03/02/2016).

[SB14] Daniel Staahl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48 – 59, 2014.

[SCL15] Gerald Schermann, Jürgen Cito, and Philipp Leitner. All the services large and micro: Revisiting industrial practice in services computing. *PeerJ PrePrints*, 3:e1588, 8 2015.

[Ser14] Serverfault. web server - how many requests should my webserver be able to handle? - server fault. `http://serverfault.com/questions/8149/how-many-requests-should-my-webserver-be-able-to-handle`, 03 2014. (Accessed on 03/25/2016).

[Sev14] Doug Seven. Knightmare: A devops cautionary tale - doug seven. `http://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/`, 04 2014. (Accessed on 03/01/2016).

[SLCG14] J. Scheuner, P. Leitner, J. Cito, and H. Gall. Cloud work bench – infrastructure-as-code based cloud benchmarking. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 246–253, Dec 2014.

[Sni16] Ed Snible. Active deploy & canary advisor. `https://www-304.ibm.com/events/tools/interconnect/2016ems/REST/presentations/PDF/InterConnect2016_7283.pdf`, 02 2016. (Accessed on 03/25/2016).

[Tho15] J. Thoenes. Microservices. *IEEE Software*, 32(1):116–116, Jan 2015.

[TKV+15] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 328–343, New York, NY, USA, 2015. ACM.

[Tof12] Giovanni Toffetti. Web engineering for cloud computing. In *Current Trends in Web Engineering*, pages 5–19. Springer, 2012.

[TSM+15] Alexander Tarvo, Peter F. Sweeney, Nick Mitchell, V.T. Rajan, Matthew Arnold, and Ioana Baldini. Canaryadvisor: A statistical-based tool for canary testing (demo). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 418–422, New York, NY, USA, 2015. ACM.

[VM01] Aad Van Moorsel. Metrics for the internet age: Quality of experience and quality of business. In *Fifth International Workshop on Performability Modeling of Computer and Communication Systems, Arbeitsberichte des Instituts für Informatik, Universität Erlangen-Nürnberg, Germany*, volume 34, pages 26–31. Citeseer, 2001.

[Wir96] Niklaus Wirth. Extended backus-naur form (ebnf). *ISO/IEC*, 14977:2996, 1996.

# Glossary

**API** Application Programming Interface

**DNS** Domain Name System

**DSL** Domain Specific Language

**EBNF** Extended Backus–Naur Form

**CD** Continuous Delivery, Continuous Deployment

**CI** Continuous Integration

**CLI** Command Line Interface

**GUI** Graphical User Interface

**HTTP** Hypertext Transfer Protocol

**HHVM** HipHop Virtual Machine

**IaaS** Infrastructure as a Service

**JS** JavaScript

**JSON** JavaScript Object Notation

**PaaS** Platform as a Service

**QoS** Quality of Service

**REST** Representational State Transfer

**SaaS** Software as a Service

**UI** User Interface

**URL** Uniform Resource Locator

**VCS** Version Control System

# Appendix

# Bifrost Sample Release Strategy

```
---
name: Webshop Redesign A/B Test
deployment:
 ref: mapping

 services:
  - ref: frontend
  - ref: products
  - ref: products_a
  - ref: products_b
  - ref: auth

strategies:
 - name: Health Check
   actions:
    - request:
        url: "http://products/"
        status: 404
        intervalTime: 5
        intervalLimit: 12
        threshold: 12

 - name: Canary Launch
   actions:
    - AND:
        actions:
          - route:
              from: products
              to: products_a
              intervalTime: 60
              filters:
                - traffic:
                    percentage: 5

          - metric:
              providers:
                - prometheus:
                    name: products_a_error
                    query: request_errors{instance="products_a:80"}
              intervalTime: 5
              intervalLimit: 12
              threshold: 12
              validator: "<5"

          - route:
              from: products
              to: products_b
              intervalTime: 60
              filters:
                - traffic:
                    percentage: 5

          - metric:
              providers:
                - prometheus:
                    name: products_b_error
                    query: request_errors{instance="products_b:80"}
              intervalTime: 5
              intervalLimit: 12
              threshold: 12
```

```
              validator: "<5"

          onTrue: Load Test
          onFalse: rollback

  - name: Load Test
    actions:
      - route:
          from: products
          to: products_a
          filters:
            - traffic:
                percentage: 100
                shadow: true
          intervalTime: 60

      - route:
          from: products
          to: products_b
          filters:
            - traffic:
                percentage: 100
                shadow: true
          intervalTime: 60

      - metric:
          providers:
            - prometheus:
                name: b_cpu_load
                query:
avg_over_time(container_cpu_system_seconds_total{name="bifrostcomposesamp
le_products_b_1"}[60s])

          delay: 60
          validator: <80
          onTrue: ab_test
          onFalse: rollback

      - metric:
          providers:
            - prometheus:
                name: a_cpu_load
                query:
avg_over_time(container_cpu_system_seconds_total{name="bifrostcomposesamp
le_products_a_1"}[60s])

          delay: 60
          validator: <80
          onTrue: A/B Test
          onFalse: rollback

  - name: A/B Test
    actions:
      - route:
          from: products
          to: products_a
          filters:
            - traffic:
                sticky: true
                percentage: 50
```

```
          intervalTime: 60

      - route:
          from: products
          to: products_b
          filters:
            - traffic:
                sticky: true
                percentage: 50
          intervalTime: 60

      - metric:
          providers:
            - prometheus:
                name: a_sold
                query:
avg_over_time(products_sold{instance="products_a:80"}[60s])
            - prometheus:
                name: b_sold
                query:
avg_over_time(products_sold{instance="products_b:80"}[60s])

          delay: 60
          validator: "a_sold>=b_sold"
          onTrue: Phased Rollout A
          onFalse: Phased Rollout B

  - name: Phased Rollout A
    actions:
       - route:
           from: products
           to: products_a
           filters:
             - traffic:
                 percentage: 5
           intervalTime: 10
           intervalLimit: 20
           threshold: 20
           onTrue: finish

  - name: Phased Rollout B
    actions:
       - route:
           from: products
           to: products_b
           filters:
             - traffic:
                 percentage: 5
           intervalTime: 10
           intervalLimit: 20
           threshold: 20
           onTrue: finish
```

# Performance Evaluation Filter Configurations

# Canary Launch

```
---
name: Webshop Redesign A/B Test
deployment:
 ref: mapping

 services:
  - ref: frontend
  - ref: products
  - ref: products_a
  - ref: products_b
  - ref: auth

strategies:
 - name: filterHeader
   actions:
    - route:
        from: products
        to: products_a
        persistance: true
        filters:
          - header:
              field: X-User-Group
              value: canary
```

# Shadow Launch

```
---
name: Webshop Redesign A/B Test
deployment:
 ref: mapping

 services:
  - ref: frontend
  - ref: products
  - ref: products_a
  - ref: products_b
  - ref: auth

strategies:
 - name: filterShadow
   actions:
    - route:
        from: products
        to: products_a
        persistance: true
        filters:
          - traffic:
              percentage: 100
              shadow: true
```

# Fixed A/B Test

```
---
name: Webshop Redesign A/B Test
deployment:
 ref: mapping

 services:
  - ref: frontend
  - ref: products
  - ref: products_a
  - ref: products_b
  - ref: auth

strategies:
 - name: filterSticky
   actions:
    - route:
        from: products
        to: products_a
        persistance: true
        filters:
          - traffic:
              sticky: true
              percentage: 50
```

# Random A/B Test

```
---
name: Webshop Redesign A/B Test
deployment:
 ref: mapping

 services:
  - ref: frontend
  - ref: products
  - ref: products_a
  - ref: products_b
  - ref: auth

strategies:
 - name: filterTraffic
   actions:
    - route:
        from: products
        to: products_a
        persistance: true
        filters:
          - traffic:
              percentage: 50
```

# Bifrost Engine Installation Guide

# Bifrost Engine

The Bifrost Engine is the central component of the Bifrost Toolkit, that consists of the Bifrost Proxy

## Setup for Development

### Requirements

- NodeJS > 4.2.*
- Gulp

### Setup

After cloning the project, make sure you have installed gulp in your global npm (`npm install -g gulp`).

1. `npm install` to install all dependencies.
2. `gulp` to transpile the sources.
3. `npm start` to run the engine.

The engine will listen on `localhost:9090`

### Gulp

There are a number of various gulp tasks that are able to help you during development:

- `gulp clean`: Removes the /dist folder and all transpiled files
- `gulp babel`: Transpiles the code to backwards compatible ECMAScript2015-compliant JavaScript
- `gulp test`: Runs the mocha test suite.
- `gulp serve`: Starts the Bifrost Engine using nodemon, automatically restarting the process upon code changes. Perfect for development!

# Bifrost Toolkit Integration Guide

# Integrating the Bifrost Toolkit

This guide will help you to setup the Bifrost Toolkit in your own application to leverage its data-driven release capabilities.

## 1. Deploying the Engine

There are currently two choices to deploy the Bifrost Engine.

### Docker

If you are using Docker, you may deploy the Bifrost Engine as follows. First, either build the docker image locally or simply use the prebuilt image using:

```
docker run -e NODE_ENV=production -e PROMETHEUS=[URL] -d --name bifrost-engine -t
dschoeni/bifrost-engine
```

Replace the Prometheus-URL with the correct full URL (for example `http://prometheus:9090`) or the Engine will be unable to properly access get Prometheus API.

Make sure that you either `--link` (deprecated) existing containers you would like to use during a release process (such as instances of Bifrost Proxy or Prometheus) or deploy them into the same Docker-Network using `--net` (preferably).

### Node.js

You can choose on how to deploy Bifrost Engine by yourself. The Engine can be deployed manually, using:

```
npm start
```

Make sure that your `NODE_ENV` environment variable is set to production, as this greatly influences the performance.

## 2. Proxying Services

In order to use the routing capabilities during releases, each service has to be proxied by an instance of Bifrost Proxy. To run a proxy for a service, simply use:

```
docker run -e NODE_ENV=production -e PORT=[PORT] -e HOST=[HOSTNAME] -d -t
dschoeni/bifrost-proxy
```

Please make sure to replace `PORT` and `HOSTNAME` with the apporiate values of the service you would like to deploy the proxy for.

Make sure that you either `--link` (deprecated) the proxied service container or deploy them into the same Docker-Network using `--net` (preferably).

## 3. Release!

If all your containers are properly running, there is only one thing left: Install the Bifrost CLI to easily schedule releases.

# Bifrost Microservices Sample Application Guide

# Bifrost Microservices Sample Application

This application has been built to demonstrate the release and live-testing capabilities of the Bifrost Toolkit.

## Launching the Application

### Requirements

- Docker >= 1.9.1
- Docker-Compose >= 1.9.1

On Windows, add the following entries into your `hosts`-file:

```
localhost auth.bifrost.com
localhost products.bifrost.com
localhost frontend.bifrost.com
```

If you're using `docker-machine`, replace localhost with the correspondant machine IP (default: `192.168.99.100`).

### Application-Only

To launch the version without Bifrost deployed, simply use:

```
docker-compose up -d
```

You should now be able to open `http://frontend.bifrost.com/` in a browser.

### Application using the Bifrost Toolkit

To launch the version using the Bifrost Toolkit, simply use:

```
docker-compose -f docker-compose-bifrost.yml up -d
```

You should now be able to open `http://frontend.bifrost.com/` in a browser.

### Generating Demo-Data

You can generate some users and data. Note that the name of the containers depends on the folder you have cloned the project into.

```
docker exec -ti bifrostmicroservicessampleapplication_auth_1 node seed.js
docker exec -ti bifrostmicroservicessampleapplication_products_1 node seed.js
```

You should now be able to open `http://frontend.bifrost.com/` in a browser and login using `demouser@demo.ch` and password `test`.

## Testing Bifrost

A sample release is provided in `/strategies/bifrost.yml`. To test this release and rollout a "modified" service, you can use the Bifrost CLI:

### Switch Services

To showcase the routing capabilities of Bifrost, one can switch the traffic to different frontend services as follows:

```
bifrost --engine [ENGINE-IP]:8181 strategies/switch_frontend_redesigned.yml
```

### Simulating a Full Release

This should launch a sample strategy using Bifrost.

```
bifrost --engine [ENGINE-IP]:8181 strategies/bifrost.yml
```