Bifrost – Supporting Continuous Deployment with Automated Enactment of Multi-Phase Live Testing Strategies

Gerald Schermann, Dominik Schöni, Philipp Leitner, Harald C. Gall
University of Zurich, Department of Informatics, Switzerland
{schermann, leitner, gall}@ifi.uzh.ch
dominik.schoeni@uzh.ch

ABSTRACT

Live testing is used in the context of continuous delivery and deployment to test changes or new features in the production environment. This includes canary releases, dark launches, A/B tests, and gradual rollouts. Oftentimes, multiple of these live testing practices need to be combined (e.g., running an A/B test after a dark launch). Manually administering such multi-phase live testing strategies is a daunting task for developers or release engineers. In this paper, we introduce a formal model for multi-phase live testing, and present Bifrost as a Node.js based prototype implementation that allows developers to define and automatically enact complex live testing strategies. We extensively evaluate the runtime behavior of Bifrost in three rollout scenarios of a microservice-based case study application, and conclude that the performance overhead of our prototype is at or below 8 ms for most scenarios. Further, we show that more than 100 parallel strategies can be enacted even on cheap public cloud instances.

CCS Concepts

•**Computer systems organization** → *Distributed architectures;* •**Software and its engineering** → Domain specific languages;

Keywords

Release Engineering; Continuous Deployment; Canary Releases; A/B Testing; Microservices;

1. INTRODUCTION

The area of continuous delivery and deployment [10] is gaining more and more traction in cloud-based software engineering [5]. Continuous delivery is a DevOps practice "intended to shorten the time between a developer committing code to a repository and the code being deployed" [2]. Shortened release cycles are essential to a company's continuing success, especially in fast-growing and contested markets such as the Web. Not only allow shorter release cycles for faster innovation, they also allow for runtime techniques to verify how users adopt new features or ideas, e.g., canary releases [10], A/B testing [12], or dark launches [8]. These live testing techniques share the philosophy that new versions are initially released to a small sample of the user base, and are rigorously monitored for increases in runtime faults, performance regressions [1], or changes in business metrics (e.g., conversion rate). Depending on a feature's performance, more and more users are assigned to the newer version or traffic is rerouted to previous, stable versions in order to keep the impact of malfunctioning releases low.

Unfortunately, consistently implementing live testing in large-scale applications, where new releases are deployed by many distributed teams on a daily basis, is a daunting task for release engineers. Multiple versions of software services need to be operated in parallel, and it is hard to track which runtime entity (e.g., which cloud instance or Docker container) is running which code version. A/B testing requires clear separation between versions, so as to prevent confounding factors from influencing test results. A wide range of technical and business metrics need to be constantly monitored and compared to a known baseline for deviations. If runtime bugs, performance regressions, or unsatisfying A/B testing results are detected, a suitable fix (e.g., a rollback, or a hotfix) needs to be triggered, and if the metrics are positive, a further rollout should be considered. All of these factors make manually administering live testing often prohibitively expensive.

In this paper, we contribute to the state of the art with a formal model of live testing, which we then use as a basis for Bifrost, a prototype system for defining and automatically enacting live testing in a service-based system. Using Bifrost, release engineers can define sophisticated release strategies involving the specification of phases of canary releasing, A/B testing, dark launches, and combinations thereof, along with the associated metrics to be monitored, threshold values, and resulting actions. Release strategies are defined in a YAML-based domain-specific language [16] (DSL), and executed via an engine implemented in Node.js. Bifrost is non-intrusive in the sense that it does not require feature toggles or other code-level changes. Instead, the middleware assumes that new releases are available as new service instances. Live testing is then implemented via traffic routing functionality.

Middleware'16, December 12-16, 2016, Trento, Italy
© 2016 ACM. ISBN 978-1-4503-4300-8/16/12...$15.00
DOI: http://dx.doi.org/10.1145/2988336.2988348

Adopting Bifrost allows developers to formally specify how a change should be rolled out. This fosters formally or probabilistically reasoning about the strategy, e.g., in terms of expected rollout time, and enables version controlling, sharing, and reusing strategies between changes or teams. We evaluate the Bifrost approach based on a realistic microservice-based example application deployed to the Google Cloud Platform. In our experiments, Bifrost adds on average a small performance overhead of 8 ms when executing a multi-phase release strategy. This seems acceptable for many use cases, especially considering that Bifrost can be removed as soon as a change is rolled out to all users. Furthermore, our experiments show that the Bifrost middleware can support more than 100 release strategies in parallel without a significant performance degradation even when deployed to a low-end, single core cloud instance. Based on published information from industry leaders in continuous deployment, such as Facebook [24], we argue that this suggests that our approach scales to real-life release engineering scenarios.

The rest of this paper is structured as follows. Section 2 provides background information on live testing, and introduces a running example used in the remainder of the paper. A formal model for specifying live testing strategies is presented in Section 3, while the Bifrost middleware is introduced in Section 4. In Section 5, we present the results of comprehensive performance evaluation of our prototype. Finally, related previous work is covered in Section 6, and Section 7 concludes the paper by summarizing the main learnings, as well as discussing future work.

2. BACKGROUND

In cloud-based software engineering, practices such as DevOps, continuous delivery, and continuous deployment, have recently reached mainstream acceptance in the developer community. A common feature of these practices is that they provide means for software houses to further speed up their release processes and to get their products into the hands of their users faster [22]. For cloud-based Software-as-a-Service (SaaS) applications, this idea of "releasing faster" often comes in the form of wide-ranging automation, e.g., a deployment pipeline [2] that, fully automatedly, builds, tests, and pushes changes into production.

2.1 Microservice-Based Applications

As defined by Lewis and Fowler [15], the microservice architectural style is an approach for developing a single application as a suite of small services, having each running in its own process and communicating with lightweight mechanisms, typically an HTTP resource API. Single services are independent of each other, they do not necessarily share the technology stack with other services (e.g., programming language, data storage technology). The key advantage of service-based applications is their inherent scalability and deployment options in comparison to monolithic applications. Services are scaled on a fine-granular level instead of running multiple copies of a monolithic application. Moreover, services are deployed independently of each other, allowing replacing service versions without affecting other application parts. This architectural concept has its advantages for the adoption of live testing methods, as described in the following. It allows not only running multiple instances of a service, but also various versions of a service at the same time (e.g., canary and baseline version). Key requirement is a routing functionality ensuring that requests are correctly forwarded between the various service instances and versions. In the remainder of this paper, we will assume applications to follow this model. However, our fundamental concepts can also be implemented for other application models, for instance using feature toggles instead of dynamic traffic routing between services [2].

2.2 Live Testing

Moving fast in terms of releasing new features, while at the same time ensuring high quality, allows companies to take advantage of early customer feedback and faster time-to-market [4]. However, releasing more frequently and with a higher degree of automation also bears the risks of, ocassionally, rolling out defective versions. While functional problems are usually caught in testing, performance regressions are more likely to remain undetected, as they often only surface under production workloads [9]. To mitigate these risks, SaaS providers often make use of various live testing techniques, most importantly gradual rollouts, canary releases, dark launches, and A/B testing.

Canary Releases. Canary releases [10] entail the concept of releasing a new feature or version to a subset of customers only, while all other users continue using the stable, previous version of the application. The idea is to test a feature on a small sample of the user base, thus testing the new version in production, but at the same time limiting the scope of problems if things go wrong. Users are either selected as a random sample of all users, based on domain-specific properties (e.g., users that ordered a specific product), or a combination thereof.

Dark Launches. Dark, or shadow, launching [8, 24] is used to mitigate performance or reliability issues of new or redesigned functionality when facing production-like traffic. The functionality is deployed on production environments without being visible or activated for any end users. However, some or all production traffic is duplicated and applied to the "shadow" version as well. This allows the provider to observe how the new feature would be behaving in production, without impacting any users,

Gradual Rollouts. Gradual rollouts [10] are often combined with other live testing practices, such as canary releases or dark launches. The amount of users testing the newest feature or functionality is gradually increased (e.g., increase traffic to the new version in 5% steps) until the previous version is completely replaced.

A/B Testing. A/B testing [12] is technically similar to the other live testing techniques discussed here, but is mainly used for differing goals. While all the techniques so far are used to evaluate a new version with regard to a baseline (the presumably stable, previous version), A/B testing is often used to compare two new, alternative, implementations of the same functional requirement. These two versions are run in parallel, with 50% of all requests going to either version. Whereas it is common to select users with particular features for canary releases, A/B tests usually require a uniform sampling of the entire user demography for both alternatives. After a predefined experiment time, metrics (e.g., conversion rate) are statistically evaluated to decide which version fared better (or whether there was a statistically significant difference at all).

2.3 Example Live Testing Strategy

A core observation underlying this paper is that rollouts in practice often consist of multiple sequential phases of live testing. For instance, a concrete rollout strategy may consist of initial dark launching, followed, if successful, by a gradual rollout over a defined period of time. If no problems, are discovered, the new change may be A/B tested against an alternative implementation, which may have run through a similar live testing sequence.

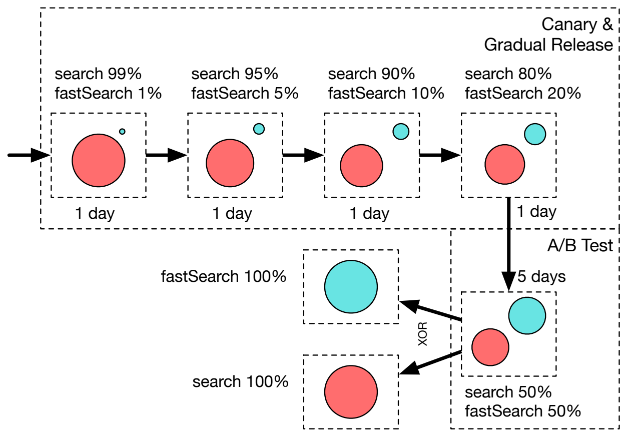

Figure 1: A simplified example of a live testing strategy with multiple phases. A change is gradually rolled out to more and more users, and subsequentially A/B tested.

A simple example live testing strategy, which will be used throughout the remainder of the paper as a running example, is given in Figure 1. Assume a company hosting a service-based web application selling consumer electronics. One of the integral services is the *search* service allowing customers to look for products they are interested in and to get an overview of the product catalog. The search service shall be redesigned and implement a new algorithm for delivering more accurate search results based on other users' search requests and their buying patterns. As replacing the previous slow, but working, search service by the new one is associated with risks, the service shall be canary tested first. Once the service performs as expected from a technical perspective, an A/B test should be conducted between the stable and canary variant. In case that the new implementation performs better according to a priori defined business metrics, a complete rollout should happen, otherwise a fallback to the stable version is conducted. The canary tested reimplementation *fastSearch* shall be rolled out to 1% of the US users first. Search and fastSearch are continuously monitored and collected metrics include response time, processing time (i.e., how long does the actual search algorithm take to get results), number of 404 requests, and the number of search requests per hour. Thresholds for fastSearch are set based on historic values collected for the stable search service, e.g., response time below 150ms. On a daily basis, and as long as the monitored metrics do not show any abnormalities, fastSearch shall be gradually rolled out to more and more users, first to 5%, then 10%, 20%, until at 50% the A/B test is conducted as shown in Figure 1. Besides more technical metrics, the A/B test focuses also on a business perspective (e.g., comparing the number of sold items on both variants), and is conducted for 5 days to capture enough data supporting statistical reasoning. State-of-the-art tools, such as the Configurator used by Facebook [24], require strategies such as this running example to be manually implemented by a human release engineer, for analyzing the data and the tweaking the rollout configuration after every step. This is labor-intensive, error-prone, and requires substantial experience in data science. In this paper, we propose BIFROST as an automated and more principled approach towards managing such release strategies.

3. A MODEL OF LIVE TESTING

Before explaining the implementation of the BIFROST middleware, we first introduce the fundamental ideas and characteristics that the system is based on, as well as the underlying formal model.

3.1 Basic Characteristics

After thorough analysis of live testing in general, and the practices discussed in Section 2 specifically, we have identified the following basic characteristics of a formal model for live testing.

Data-Driven. Live testing require extensive monitoring to decide on test outcomes or evaluate the current health state. This monitoring data is collected using existing tools in the application's landscape using Application Performance Monitoring, such as Kieker [26] or New Relic[1]. A model of live testing needs to support the inclusion of monitoring data into its runtime decision process.

Timed Execution. Live testing requires the collection, analysis, and processing of data in defined intervals. Gradual rollouts depend on timed increments to gradually introduce new versions or control the routed traffic. Depending on the concrete usage scenario, these methods may stretch over minutes, hours, or days.

Parallel Execution and Traffic Routing. All live testing practices require the parallel operation of multiple versions of a service, e.g., a stable previous version and an experimental new implementation for canary releases, or two alternative implementations for A/B testing. This also requires the correct routing of users to a specific version. For instance, canary releases are often targeted at specific user groups. For A/B tests, it is often important that the same user is directed to the same implementation across sessions.

Ordered Execution. Ordered execution is required to form live testing strategies consisting of chained phases of canary releasing, gradual rollouts, dark launches, and A/B tests. An example for such a live testing chain is given in Section 2.

3.2 Live Testing Model

Based on these identified characteristics, we derived a formal representation for live testing strategies. To begin with, a *strategy* $\mathcal{S}$ is modeled as a 2-tuple:

$$\mathcal{S} : \langle \mathcal{B}, \mathcal{A} \rangle$$

A strategy $\mathcal{S}$ consists of a set of *services* $\{b_1, \ldots, b_n\}$ and a deterministic finite automaton $\mathcal{A}$. In our model, services $b_i \in \mathcal{B}$ represent atomic architectural components, for instance services in a microservice-based system. Services b_i themselves are available in different versions (e.g., a stable previous search service version, and an experimental new

[1]https://newrelic.com

version) or as alternative implementations (e.g., for A/B testing). Whenever a change is rolled out, a new service version is launched. For a service b_i, this is modeled as a tuple $\langle v_1, \ldots, v_n \rangle$. Moreover, each of those versions v_i is associated with static configuration information sc_i, which holds a version's endpoint information (e.g., host name, IP address, and port). A user $u_i \in \mathcal{U}$ connected to the system is always using exactly one version of a service. However, this assignment may change during the execution of a release strategy (e.g., during a gradual rollout a user may be reassigned from a stable version to the canary version). Thus, this dynamic routing information, i.e., to which version v_j of a service b_i a user u_k is assigned to, modeled as a 3-tuple $\langle u_k, v_j, sticky \rangle$, represents an important part of a service's routing state. *Sticky* is a boolean flag specifying if a user's assignment is permanent for a certain state, thus whether a subsequent request by a user (e.g., search request) may be routed to a different version or not. Dark launches are different from all other live testing practices, in that they duplicate rather than reroute a traffic to a specific service version. This is modeled as a 3-tuple $\langle v_{i,j}, v_{k,l}, p \rangle$, where $v_{i,j}$ denotes the source version from which p percent of the traffic is duplicated and also routed to the target version $v_{k,l}$. Thus, the dynamic routing configuration dc_i of a service b_i is a 2-tuple $\langle \mathcal{M}, \Gamma \rangle$ being $\mathcal{M}$ a tuple of user mappings $\langle u_k, v_j, sticky \rangle$ and Γ a tuple of dark launch routing information $\langle v_{i,j}, v_{k,l}, p \rangle$.

The execution state of a release process is represented by an *automaton* $\mathcal{A}$, which is defined by a 5-tuple $\langle \Omega, S, s_1, \delta, F \rangle$. Ω represents the monitoring data a live testing strategy uses for decision making. Ω is modeled as tuple of metrics $\langle m_1, \ldots, m_n \rangle$, each m_i representing a time series $(t_0, \ldots, t_n)$ of metric values over time (t_0 to t_n). This data typically originates from external monitoring solutions. The automaton itself is defined as a set of states $\{s_1, \ldots, s_n\}$, $s_i \in S$, a starting state s_1, and set of final states F, where $F \subseteq S$. δ is a state transition function specifying the subsequent state depending on the current state and the outcome of a state's associated checks ($e \in \mathbb{Z}$), formally defined as $\delta : S \times \mathbb{Z} \rightarrow S$. States and transitions represent the concept of ordered execution, in which multiple states form distinctive phases during the live testing process.

A state s_i is defined as a 5-tuple $\langle \mathcal{C}, \mathcal{T}, \mathcal{W}, \Phi, \eta \rangle$ including checks $\mathcal{C}$, thresholds $\mathcal{T}$, weights $\mathcal{W}$, configurations Φ, and a user selection function η. In a state, multiple checks modeled as a tuple $\langle c_1, \ldots, c_n \rangle$, $c_i \in \mathcal{C}$, are executed at the same time, thus matching the characteristic of parallel execution. A check may for instance represent monitoring a specific metric (e.g., service response time). The outcomes of each individual check are combined as a weighted linear combination. The resulting outcome value serves as input for the state transition function δ. For each check c_i, there is a weighting factor $w_i \in \mathcal{W}$, thus formally, weights are modeled as a tuple $\langle w_1, \ldots, w_n \rangle$.

Each state is associated with specific services' dynamic routing configurations Φ, modeled as a tuple $\langle dc_j, \ldots, dc_k \rangle$ containing all configurations of services relevant for a state. The user mappings $\mathcal{M}$ of those dynamic routing configurations are built and controlled by the state's function η, formally $\eta : \mathcal{U} \rightarrow \mathcal{V}$, which assigns a specific user u_i to a version v_j of service s_k. This allows fine-grained routing and filtering functionality, e.g., assign 5% of US users to the *fastSearch* canary. Our approach is agnostic to how this selection and filtering is implemented. For instance, our approach is compatible to the user selection and sampling approach used in Configurator [24]. Once the execution of a release strategy enters a state s_i, the dynamic routing configurations of the services associated to this state are evaluated and executed.

An example state machine for the running example introduced in Section 2 is given in Figure 2. In state b, the stable *search* service is assigned to 95% of the users, while the *canary tested*, newly designed reimplementation *fastSearch* is used by 5% of the users. Depending on the outcome of the various checks in each state and their weighting, a numerical outcome value is generated in each state. This outcome value is compared against defined *thresholds*, leading to a state transition. For instance, in state b, a transition either happens directly to state d because of the canary's good performance (outcome > 4), to state c in which the traffic is only slowly increased (outcome $= 4$), or a rollback happens transitioning to state g (outcome mapped ≤ 3).

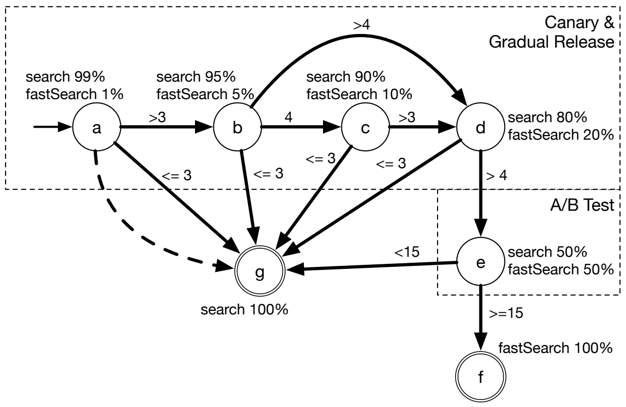

Figure 2: A visualization of the state machine of the running example. Every state executes checks for a specified amount of time, leading to a numerical outcome value. State transitions are based on this outcome. State "g" represents a rollback of the release. The dashed arrow in state "a" represents an "exception" that allows to jump directly to the rollback state "g" if a serious problem is detected in "a".

Formally, the state transition function δ takes the current state s_i and the state's aggregated and weighted outcome value $e \in \mathbb{Z}$ as input. For each state, an ordered tuple of thresholds $\langle t_1, ..., t_n \rangle$, $t_i \in \mathcal{T}$, is specified containing at least one value. A tuple of thresholds with n values forms $n + 1$ disjoint ranges, e.g., thresholds $\langle 2, 4 \rangle$ form the ranges $-\infty < x \leq 2$, $2 < x \leq 4$, and $4 < x \leq \infty$. In the state transition function δ, to each range of a state s_i, a state s_j is assigned, representing the automaton's subsequent state if the aggregated outcome value e falls into the range. In Figure 2, state a has exactly one threshold (i.e., 3), thus forming exactly two possible state outcomes, while state b has two thresholds (i.e., $\langle 3, 4 \rangle$) and thus three outgoing transitions.

In the following, we will elaborate step by step how a state's outcome value is determined and when state transitions are triggered. A single state executes multiple checks at the same time. A check c_i is defined as a 3-tuple $\langle f_{c_i}, \Omega^i, \tau \rangle$ consisting of a metric evaluating function $f_{c_i} : \Omega^i \rightarrow \{0, 1\}$, monitoring data Ω^i, and a timer τ.

In detail, a check c_i's function f_{c_i} takes a subset of the monitoring data $\Omega^i \subseteq \Omega$ as input and returns 0 or 1, i.e., a check is either successful or not. Such evaluation functions could be of varying complexity, they might check only for a single service version's metric (e.g., response time < 150ms), a combination of multiple metrics of a version, or evaluate even metrics across multiple services and versions (e.g., for the purpose of A/B testing).

In order to reason about a service's behavior, it is necessary to continuously evaluate the adherence to specified metrics, thus a check's evaluation function may be executed multiple times. This is achieved by introducing a timer mechanism τ, controlling when and how often single checks execute. Functions evaluating monitoring data are executed independently of each other. This is illustrated in Figure 3 showcasing the timed (re-)execution of the functions associated with three checks using different execution intervals. As the outcome of a single function execution is either 0 or 1, the outcome of a check is determined by aggregating (i.e., summing up) the outcome values of each execution (i.e., 1 to n) during the course of time controlled by τ leading to an outcome value $e \in \mathbb{Z}$.

$$\begin{aligned} f^{\tau}_{c_i}(\Omega^i) &: f^{1}_{c_i}(\Omega^i) + \ldots + f^{n}_{c_i}(\Omega^i) \\ &= \sum_{j=1}^{n} f^{j}_{c_i}(\Omega^i) \rightarrow e \in \mathbb{Z} \end{aligned}$$

The model distinguishes between two types of checks: *basic* checks and *exception* checks. While for basic checks the single execution results are only evaluated at the end, single execution results of exception checks trigger state transitions whenever their evaluation function returns 0. The intuition here is that for basic checks, individual tests may fail (e.g., even if a change performs as expected, there may be a small number of individual checks for a change for which the error rate slightly increased due to expected stochastic variations). However, if things are going very badly (e.g., a 100% or higher increase in the error rate), exception checks allow developers to immediately roll back a release without having to wait to the end of the current state. In Figure 3, such state changes could happen at t_0, t_1, t_2, and t_3. State a in Figure 2 contains an exception check leading to state g.

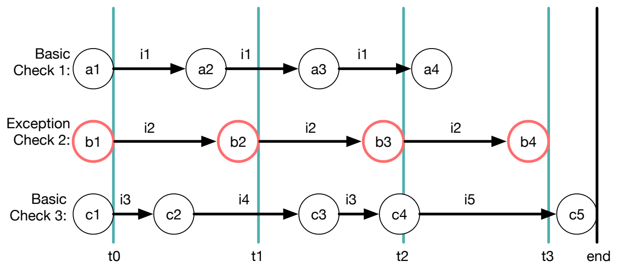

Figure 3: An illustration of the time-based execution of multiple checks.

Formally, an exception check c_i is a 4-tuple $\langle f_{c_i}, \Omega^i, \tau, s_j \rangle$ consisting of a metric evaluating function f_{c_i}, monitoring data Ω^i, a timer τ, and a fallback state $s_j \in S$ to which the automaton switches if the evaluating function returns 0 during its timed (re-)execution. If all n function executions are successful, the aggregated outcome value of an exception check equals n.

A basic check c_i is a 5-tuple $\langle f_{c_i}, \Omega^i, \tau, T_{c_i}, Out_{c_i} \rangle$ of a metric evaluating function f_{c_i}, monitoring data Ω^i, a timer τ, an ordered tuple of thresholds $\langle t_1, ..., t_n \rangle$, $t_i \in T_{c_i}$, and an output mapping Out_{c_i}.

$$\begin{aligned} Out_{c_i} : \{ & (t_i, t_{i+1}, r_i) \mid t_i < t_{i+1}, r \in \mathbb{Z}, \\ & t \in T_{c_i}, i \in \{1, \ldots, n\} \} \end{aligned}$$

Similar to a state's outcome in the state transition function δ, the aggregated outcome of a basic check e is compared to thresholds forming disjoint ranges, and based on the check's outcome mappings Out_{c_i}, mapped to an integer value r_i.

$$e \xrightarrow{Out_{c_i}} \{ r_i \mid t_{i-1} < e \leq t_i, (t_{i-1}, t_i, r_i) \in Out_{c_i} \}$$

Thresholds are used to cope with varying monitoring data, e.g., the response time of the monitored *fastSearch* service may vary, thus the outcomes of the evaluation function may vary as well. Outcome mappings allow mapping those different outcome values onto a normalized integer outcome value. For example, assume a basic check for controlling *fastSearch*'s response time in state b in Figure 2. The check is executed 100 times in intervals of 10 minutes. The response time check's thresholds are 75 and 95, thus forming ranges $x \leq 75$, $75 < x \leq 95$, and $x > 95$. The corresponding mappings are $(-\infty, 75, -5)$, $(75, 95, 4)$, and $(95, \infty, 5)$. This means that if the check fails more than 24 times, the mapping returns -5, if the aggregated value is between 75 and 95, it returns 4, otherwise 5.

Once we have the results of the single checks of a state, the final step is to aggregate those results as a weighted linear combination, and consider their weighting factors in order to determine the state's outcome.

$$\sum_{i=1}^{n} f^{\tau}_{c_i}(\Omega^i) * w_i \rightarrow e \in \mathbb{Z}$$

Given the current state s_i, this final result e is the input for the state transition function δ, resulting in either a state change, or staying in the current state. In this case, the state is re-executed, with all timers and thresholds reset. This concept of multiple outgoing paths allows (1) continuing the rollout strategy if the tested services behave as expected, (2) staying in a certain state if results are not definite and require reexecution, or (3) switching to a fallback state if new functionality does not behave as expected and to keep its impact low. Moreover, the concept of exception checks allow state changes (i.e., roll backs) at any time during the execution.

4. BIFROST

In this section, the Bifrost middleware is presented. The system is a Node.js based prototype implementation of our live testing model. Our prototype specifically targets microservice-based applications.

4.1 System Overview

As visualized in Figure 4, the two main components of the Bifrost middleware are the Bifrost engine and Bifrost proxies. The middleware acts on top of the application's services, ensuring that routing instrumentation specified in the release strategy is adhered to.

Conceptually, there is exactly one Bifrost proxy for each service that is part of the applied live testing method. This

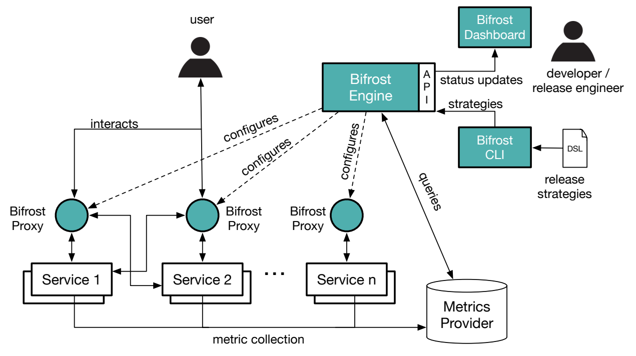

Figure 4: High-level architectural overview of the BIFROST middleware.

one-proxy-per-service concept prevents traffic bottlenecks and keeps services decoupled. A service acting behind by a proxy may run in multiple instances and multiple versions at the same time. BIFROST proxies facilitate live testing via implementing dynamic traffic routing. For instance in case of an A/B test, 50% of all traffic is routed transparently to two different versions of a service. A key advantage of this design is that the middleware is easy to integrate into existing applications, without altering or rewriting functionality. Thus routing and rollout logic is not part of the services' code bases, as would be the case for feature toggles [2]. The middleware supports any web-based service including databases and external services accessed through HTTP. BIFROST proxies are lightweight. Each instance of the proxy is basically another service added to the application, and proxies work in combination with load balancers, auto-scaling functionality, reverse proxies or request gateways. The BIFROST engine has the main responsibility to orchestrate and properly configure the deployed proxies in the system. Basically, the engine executes the state machine of the formal release model. It interprets the release strategies specified in a domain-specific language, and continuously queries and observes monitoring data collected by metrics providers or external services in order to evaluate the rules specified in the release strategies and enact appropriate actions (i.e., state changes). Whenever a state change happens during the rollout process (e.g., entering a new phase in the specified strategy), the engine updates the affected proxies.

Besides the middleware components, BIFROST comprises two additional tools, the BIFROST command-line interface (CLI) and BIFROST dashboard. The CLI connects to the BIFROST engine and allows scheduling and executing release strategies remotely or as part of release scripts (e.g., build automation using Jenkins). The BIFROST dashboard visualizes the current execution state of release strategies providing detailed information such as the outcome of executed checks (e.g., metric below threshold).

4.2 Implementation

We now discuss how this high-level design has been realized in BIFROST.

4.2.1 Technology Stack.

The BIFROST middleware has been developed mainly in JavaScript utilizing Node.js as the server-side JavaScript runtime, in combination with Babel[2], which is a backwards-compatible JavaScript transpiler. 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 release process (e.g., checks running in parallel with different timer configurations). The communication between the middleware's components is handled through RESTful HTTP APIs that make use of ExpressJS[3]. Moreover, Socket.IO is used implementing the WebSocket protocol providing full-duplex communication channels. This is necessary for updating the BIFROST CLI and dashboard with real-time information. Finally, the proxy functionality has been implemented using node-http-proxy[4].

4.2.2 Domain-Specific Language

To simplify the specification of release strategies and thus to avoid specifying every single state of the underlying formal model, the BIFROST domain-specific language (DSL) was designed. Besides fostering simplicity, the text-based DSL aims to be version-controlled, thus supporting transparency and traceability of a company's release strategies. The DSL was built as an internal DSL on top of YAML as a host language. YAML is a data serialization language designed to be readable by humans. In the following, we will present implementation details of and design decisions for the engine based on small DSL code snippets showcasing specific elements of a rollout strategy. However, a more detailed description of the DSL is out of the scope of this work, but example strategies formalized in the DSL that have been used throughout the evaluation of BIFROST are part of our online appendix.

Data-Driven Execution. Collected and aggregated monitoring data is the essential ingredient for the engine's runtime decisions. The BIFROST engine is designed to support multiple data sources. However, currently, the engine's prototype implementation is primarily built for Prometheus[5]. Listing 1 shows an example how a *basic check* is implemented in the BIFROST DSL in form of a *metric* element.

```
1   - metric:
2     providers:
3       - prometheus:
4           name: search_error
5           query: request_errors
6                 {instance=”search:80”}
7     intervalTime: 5
8     intervalLimit: 12
9     threshold: 12
10    validator: ”<5”
```

Listing 1: Example Metric

Lines 2 to 6 specify the data retrieval, i.e., to which provider to connect to and which query to be executed. The metric providers' access information (i.e., IP, port) is specified in a configuration file loaded at the engine's start-up. In this concrete example, the query retrieves the amount of request errors associated with the service instance *search* from Prometheus. BIFROST supports retrieving an arbitrary number of metrics from different data providers in the context of a check. The retrieved data is then associated to the provided name and can be used inside the scope of the check for validation purposes.

[2]https://babeljs.io/
[3]http://expressjs.com/
[4]https://github.com/nodejitsu/node-http-proxy
[5]https://prometheus.io/

Timed Execution. Each basic check in our model has a metric evaluating function, which operates on a set of metrics, and its execution is controlled by a timer. In the previous step, we have already shown how the engine collects metrics. Line 10 of Listing 1 shows a simple function evaluating the collected metrics. In this case, a single metric is retrieved and compared to a scalar value. The check is re-executed every 5 seconds and 12 times in total. The current implementation of the DSL represents a simplified version of the release model discussed in Section 3.2. Each check has exactly one threshold value, thus the aggregation of the result of a check's timed-execution can be mapped to either true or false. In line 9, the threshold is set to 12, which means that the check returns only true if all 12 executions evaluate to true.

Rollouts. The parallel execution of checks and their aggregated outcomes may lead to state changes, which then influence how traffic is routed through the system, thus changing dynamic routing configurations dc_i of services b_i. The basic instrument for specifying such rollouts is the *route* directive in the BIFROST DSL. An example for a route supporting dark launches is provided in Listing 2.

```
1  - route:
2      from: search
3      to: fastSearch
4      filters:
5        - traffic:
6            percentage: 100
7            shadow: true
8      intervalTime: 60
```

Listing 2: Dark Launch

The example specifies that all traffic (line 6) routed to the *search* service within the next 60 seconds (line 8) shall be duplicated (line 7) and also routed to the *fastSearch* service. This allows dark launching a service, thus assessing amongst others whether the tested service scales correctly.

In order to support such mechanisms, BIFROST proxies intercept incoming connections, and depending on their configuration, they route requests accordingly. BIFROST supports two types of routing: header-based and cookie-based. The former inspects a request's header fields (specified in RFC 2616), which could include custom-named header fields as well. For header-based traffic filtering, the proxy itself does not decide to which service instance a request is routed, it acts solely on its configuration received from the engine. Thus, the concrete header field has to be injected somewhere else in the process, e.g., by an external service called at the user's login controlling which users are in which group of a conducted A/B test. This is different for the second option, cookie-based filtering, where, for example in case of A/B tests, the proxy decides into which bucket a request is put into. Listing 2 shows an example for such a cookie-based filtering variant. In addition, this concept is used for applying general random traffic filtering such that a certain percentage of users is assigned to a specific version. However, depending on the type of the conducted release practice, it may be important that requests from the same users are always routed to the same service instance (e.g., A/B testing). This behavior is generally called sticky sessions. The proxy accomplishes this by setting a cookie on the client using the *Set-Cookie Header* in its response. The cookie contains a RFC-compliant UUID that is used to re-identify the client in subsequent requests. Depending on whether sticky sessions are used or not, the proxy either stores the set cookie to re-identify users, or the subsequent request is again running through the proxy's decision process.

Deployment Configuration. Evidently, the engine needs to be aware of which services exist in the system, and where the proxies are located. This corresponds to the static routing information modeled in the formal release model. In the BIFROST DSL, this is covered by the DSL's deployment part, while the specification of the previous code snippets where all in the DSL's strategy part. The former takes a list of key-value pairs mapping host names of services to host names of corresponding BIFROST proxy instances. This simple mechanism allows the tool to work in different deployment setups. The middleware per se is not responsible for the deployment of the various components. However the DSL and engine are designed in such a way to be extended and make use of deployment management tools, such as Chef or Puppet, in future versions.

5. EVALUATION

The BIFROST toolkit provides developers with a flexible approach to introduce various rollout practices into their release process. However, the feasibility of this approach is influenced by the middleware's performance impact and how well the approach scales, both conceptually and technically. Thus, in the following section we specifically take a look on how the BIFROST middleware performs in realistic settings. We look at two different scenarios, evaluating the performance overhead introduced by the BIFROST proxies for the end user as well as the scalability of the BIFROST middleware itself, in terms of parallel strategies and checks. A replication package for our study is available in the online appendix.

5.1 Evaluation of End-User Overhead

We firstly address the question whether using BIFROST degrades end user performance.

5.1.1 Case Study Application

To address this question, a case study application simulating a generic microservices application was necessary. Unfortunately, few suitable open source microservice-based applications exist. Hence, we developed a custom Node.js based case study application specifically to run performance tests against for the purpose of evaluating the middleware. The implementation of this case study is available in the online appendix.

This application simulates a generic e-commerce website selling consumer electronics. It was kept simple in order to provide a testbed for the performance evaluation and demonstration of the capabilities of the BIFROST middleware. The application consists of 7 services in total: a HTML/JavaScript *frontend*, and three RESTful HTTP services, *product*, *search*, and *auth*. The *product* service allows browsing the product catalog and placing buy orders, the *search* service is used for executing text-based product search queries, and *auth* service authenticates and authorizes users based on their provided e-mail and password, and validates tokens. In addition, there is a *MongoDB* database for storing products and users, an instance of *Prometheus*, which collects container and low-level performance metrics as well as business metrics from services that expose them, and finally *nginx*[6]. Nginx is a reverse-proxy used as a central entry-point to the application for users. It proxies incoming requests to either the frontend service or to the product service. An overview of the case study application architecture is provided in Figure 5. Connections between the services and Prometheus were omitted for clarity reasons.

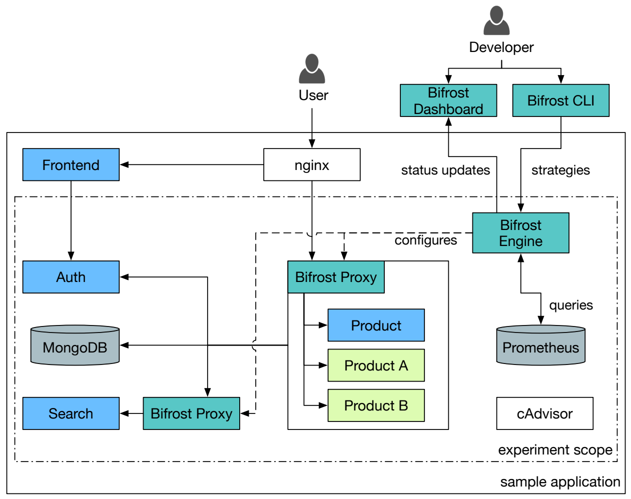

Figure 5: Architecture of a microservice-based case study application, consisting of 7 microservices.

5.1.2 *Experiment Setup*

We now discuss how we have set up the case study application and experiment.

Case Study Application Deployment. We deployed the case study application on 12 virtual machines forming a Docker Swarm[7] on the Google Cloud Platform[8]. We used virtual machines of type `n1-standard-1` in Google's `us-central1-a` region. Consequently, each virtual machine had a single virtual CPU implemented as a single hardware hyper-thread on a 2.6 GHz Intel Xeon E5 and 3.75 GB memory. Experiments were conducted between May 1st and May 19th, 2016.

The first node acted as Swarm-Master. Docker Swarm allows clustering a pool of Docker hosts into a single virtual Docker host supporting the execution of Docker Compose, which simplifies application deployment and in our case replication as well. Every service of the case study application resides in its own Docker container. Moreover, to ensure that a single container's performance does not influence other containers, in this setup, all containers were running on their own virtual machine. Besides the services of our case study application, the middleware components were deployed as Docker containers as well, i.e., one VM hosting the Bifrost engine, and two VMs hosting proxies for the `search` and `product` service. In addition, to automate the evaluation process, the Bifrost CLI was put into a dedicated container as well. As the `auth` service is not relevant for the executed live testing strategy, it does not use a Bifrost proxy. This simulates the case of a stable service for which currently no live testing strategy is executed. To collect the containers' performance metrics (e.g., CPU utilization, memory consumption) cAdvisor[9] was used pushing the collected data to Prometheus, which further increased the number of containers and VMs in our experiment setup by two. Finally, to simulate production traffic, we used another Docker container and VM of the same type for hosting an instance of Apache JMeter as load generator.

Test Setup. The goal of this experiment was to show the performance impact of the Bifrost middleware in a more complex release cycle consisting of the execution of a release strategy involving multiple live testing methods. In this scenario, the product service shall be replaced and two new alternatives were implemented for this purpose, `product A` and `product B`. The specified release strategy introduces both alternatives to the running system, runs a set of live testing methods making sure that they perform as expected and depending on the outcome of those tests, one of the newly implemented product services shall be gradually rolled out to all users. The release strategy involves the following phases.

1. Canary Launch: Tests `product A` and `product B` service while monitoring for errors, i.e., HTTP status code 500 responses. 5% of the traffic to the stable product service gets redirected to A and B respectively, and an aggregated error count from Prometheus is monitored. This phase lasts for 60 seconds, and is implemented using cookie-based routing without sticky sessions. This phase corresponds to a single state in the formal model with two checks running in parallel, which are re-executed every 12 seconds.
2. Dark Launch: `Product A` and `product B` receive 100% of all original traffic to the product service for a duration of 60 seconds. This represents a single state in the formal model. We refrained from our initial checks on the services' CPU utilization as this would have led, in certain cases, to automatic rollbacks during our load test.
3. A/B Test: Routes 50% of the product traffic to `product A` and the remaining 50% to `product B`. As a test metric the sales performance is monitored over 60 seconds. The test uses sticky sessions and cookie-based routing. After completion, the traffic distribution is reverted to the original product service. This live test corresponds to a single state in the model, with one check executed at the end.
4. Gradual Rollout: Rolls out the winner from the previous A/B Test starting with 5% traffic up to 100%, increasing traffic 5% every 10 seconds, for 200 seconds duration in total. Corresponds to 20 states in the model.

Note that, in order to compress the total duration of the experiment to 380 seconds, we chose extremely short execution times for each phase. Obviously, in practice, developers would typically choose longer durations for each phase.

We initiated the execution of the live testing strategy after a ramp up period of 30 seconds to slowly increase the load and after an additional 60 seconds for health checking the deployed services. After the ramp up, a steady traffic of 35 requests per second was simulated using a JMeter test suite. The test suite targeted the product service and consisted of 4 different requests that touched different parts of the system:

[6]https://www.nginx.com
[7]https://docs.docker.com/swarm/
[8]https://cloud.google.com

[9]https://github.com/google/cadvisor

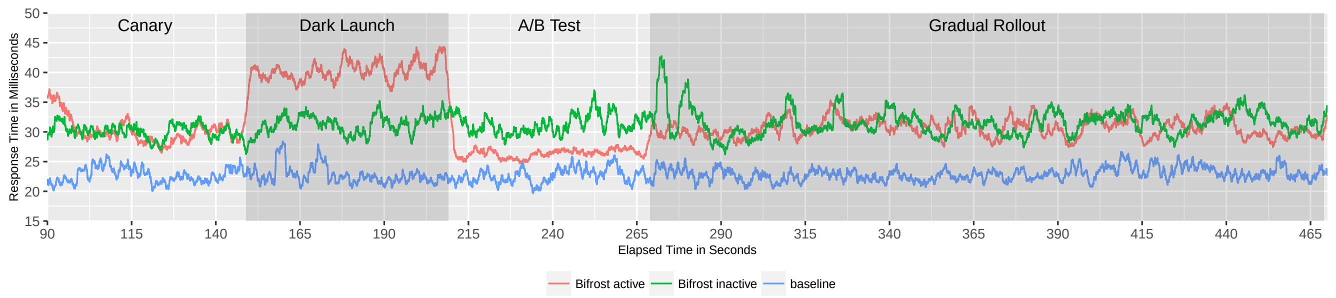

Figure 6: 3-second moving average of response times as monitored in the JMeter load generator over the duration of the experiment. Baseline is the response time without BIFROST, inactive represents BIFROST, and specifically the routing proxies being installed but without any active strategy, and active is the case when a live testing strategy is being executed.

	Canary			**Dark Launch**			**A/B Test**			**Gradual Rollout**		
	baseline	inactive	active	baseline	inactive	active	baseline	inactive	active	baseline	inactive	active
mean	22.75	30.04	30.28	22.68	31.34	40.23	22.64	31.30	26.52	22.93	31.59	30.68
min	20.04	26.27	26.47	20.42	27.95	31.67	19.65	27.86	24.65	20.35	26.20	27.43
max	26.24	32.79	38.58	28.44	35.26	44.35	26.05	37.03	31.67	26.66	42.76	35.34
sd	1.26	1.21	2.22	1.53	1.53	1.70	1.23	1.58	1.00	1.07	2.18	1.55
median	22.58	30.08	29.77	22.36	31.51	40.11	22.59	30.98	26.49	22.85	31.43	30.53

Table 1: Basic statistics of response times in milliseconds for all release phases.

- Buy: A HTTP POST request to the `product` service, which writes to the database. No response body is sent back.
- Details: A HTTP GET request to the `product` service, which returns information about a single product. The request only requires a read operation in the database, and returns a small response body.
- Products: A HTTP GET request to the `product` service, returning a list of all products including their buyers. Requires a read operation in the database as well, but returns a large response body.
- Search: A HTTP GET request to the `product` service, which in turn invokes the `search service`. Requires another read operation in the database, and returns a small response body.

All requests require authorization via the `auth service`. We conducted test runs in three different variations: (1) baseline, i.e., running the load test without the middleware and proxies deployed, (2) BIFROST inactive, running the load test with the middleware and proxies deployed but without executing any strategy, and (3) BIFROST active, running the load test with the middleware and proxies deployed and executing a strategy. For each of those three variations we collected the average response time in 5 test runs and used a moving average with a window size of 3 seconds for aggregation.

Test Results. Figure 6 plots the average end user response time as measured by the JMeter load generator during the release in the described phases (canary launch, dark launch, A/B test, gradual rollout). The single release phases are highlighted for better readability. We observe that, in general, BIFROST introduces a constant small overhead to service invocations. For gradual releases and canary tests, this overhead is approximately 8 ms in our tests (see also Table 1 for detailed numbers), which we consider acceptable for many production settings. Further, it should be noted that our Node.js based prototype implementation is not optimized for speed, and a more efficient implementation would likely be feasible. Further, our evaluation setup made use of cookie-based routing, which is generally slower than a header-based routing would be. Finally, this case study application and all components have been deployed on low-end cloud instance types. More powerful instance types, or dedicated server hardware, would likely reduce the overhead further. However, even with this prototype implementation we have shown that our underlying concept seems feasible for real-world usage. Another observation from Figure 6 is that response times are stable within phases. That is, there is no middleware-induced change in the overhead during tests, which is particularly important for A/B testing. The scenarios when BIFROST is inactive and active did not lead to statistically significant response times for canary releases and gradual rollouts, indicating that the execution of a single strategy is cheap. This will be researched in more detail in Section 5.2.

Two phases need more explanation, specifically the A/B test and the dark launch. For the A/B test (third phase in the figure), we observe that the average response time decreases in comparison to when BIFROST is inactive. This is a side-effect of the load balancing effect of A/B testing, i.e., in this phase invocations are by definition split between two services, leading to reduced load on both of them. In effect, this reduces the overhead to approximately 4 ms. For the dark launch, we observed the opposite effect. As this live testing strategy requires duplication of traffic, the overhead induced by the middleware is increased as well, leading to an overall higher response time and an increased overhead of 18 ms. This is because in our test setting three requests need to be shadowed (requests to the authentication service, the product service, and the database). Thus, in contrast to other live testing methods, dark launching requires a certain level of caution (e.g., making sure that the proxy runs on machine able to handle the load), especially if, as in our setup, 100% of the traffic is duplicated.

5.2 Evaluation of Engine Performance

The previous performance test focused on the overall application's performance. However, as we executed only a single release strategy, we now want to study how the Bifrost middleware behaves under load created by (1) executing multiple release strategies at the same time (simulating the case of a large organization with many teams, all independently releasing new versions), and (2) executing complex release strategies with an increasing amount of parallel checks.

5.2.1 Executing Multiple Release Strategies

This test studies how many parallel live experiments can be conducted at the same time, and, thus, whether our middleware is capable of being used in a broader context in a company having various different product teams launching rollout experiments independently from each other.

Case Study Application Deployment. We used a cluster of 4 virtual machines with the same specification as described before forming a Docker Swarm on the Google Cloud Platform. We used the *product* and *product A* service of our sample application running in their own containers as target of all executed release strategies. To collect performance metrics (e.g., CPU utilization, memory consumption), containers hosting cAdvisor and Prometheus were deployed. Moreover, a MongoDB container complemented the deployment setup. While the engine and the proxy had their own VMs, cAdvisor and Prometheus shared the third VM, and the remaining containers shared the fourth VM.

Test Setup. For this experiment the application itself was irrelevant as as long as we could simulate typical engine-to-proxy communication and show the middleware's scalability. Hence, there was no simulated load targeting the case study services during this experiment.

To execute multiple release strategies, we used a slightly modified version of the release strategy presented in Section 5.1. The strategy consisted again of 4 phases (canary, dark launch, A/B test, phased rollout) with a duration of 280 seconds in total. The checks and routing instrumentation for `product B` were not relevant for this experiment and were consequently removed. The duration of the final phase was decreased by 100 seconds.

In order to evaluate the scalability of Bifrost with regards to parallel strategies, we increased the number of executed release strategies in a stepwise manner from 1 over 5 to 10, and then for each additional step by 10 until 200 strategies. Our goal was to observe the load on the Docker container running the Bifrost engine, which is responsible for enacting the defined release strategies. A single test run was repeated 5 times, including the collection of CPU and memory utilization data, and the raw duration of each strategy execution, i.e., end time – start time.

Test Results. Figure 7 shows the engine's CPU utilization when running multiple strategies in parallel. CPU utilization is the driving factor as both the engine's and the proxy's memory consumption was on a stable, but increasing level. Even though executed on a cheap cloud instance with a single core CPU, the engine is able to handle more than 100 strategies executed in parallel. When considering that even industry leaders in continuous deployment, such as Facebook [20, 24], deploy between 100 and 1000 times a day, this is a good indication that our middleware is able to handle realistic concurrent deployment numbers even on low-end public cloud resources.

This is also supported by looking at how long it takes Bifrost to enact each of those strategies. This is visualized in Figure 8. Up to 80 parallel strategies, there is a small,

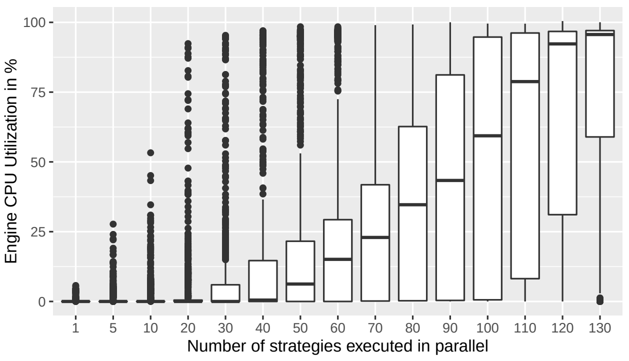

Figure 7: Boxplots of CPU utilization on the Docker container running the Bifrost engine. Even with more than 100 strategies being executed in parallel, the instance is rarely fully utilized.

linear increase in delay for each additional strategy. From this point onwards, the engine slowly starts to become overloaded, hence the standard deviation of delays increases and the delay rises with each additional strategy substantially.

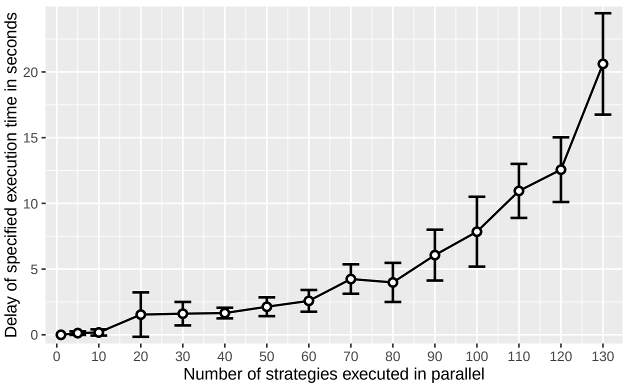

Figure 8: Delay in enacting a release strategy when running multiple strategies in parallel. Error bars represent ± one standard deviation.

It should be noted that our experiment represents a worst case for the Bifrost engine, as all strategies in the experiment were executed at the same time and with identical configuration, thus the periodic reexecution of the checks happened at the same time as well. However, because of the single core environment, execution at the same time is not possible and thus, a slight delay is introduced for each strategy. However, even in this setting, a delay of 8 seconds in the mean for enacting 100 releases at the same time is usually negligible in practice, as realistic live testing phases usually span hours or days.

5.2.2 Executing Release Strategies With Many Checks

In this experiment we study the upper bound of parallel checks the Bifrost engine can handle.

Case Study Application Deployment. We launched a cluster of 3 virtual machines forming a Docker Swarm on the Google Cloud Platform, with the same specification as before. Similar to the previous experiment, we focused on the engine's behavior. Hence, no load for the case study application was produced. Besides the engine, we used containers for the *product* and *product A* services, a container hosting a single BIFROST proxy instance, and a container for MongoDB. Moreover, to collect performance metrics (e.g., CPU utilization, memory consumption), containers hosting cAdvisor and Prometheus were deployed. The engine and the proxy instance were deployed on separate VMs, while the remaining 5 containers shared the third VM.

Test Setup. In this experiment, we stressed the engine with a single release strategy, but using an increasing number of parallel checks. Our goal was to identify an upper bound at which the engine is unable to handle the accumulating load. The strategy we used was trivial, consisting only of two identical phases, each running 60 seconds. Each phase contained $8 * n$ checks, where n denotes the current step (stepsize = 10). Out of those 8 checks, 3 target the availability of the product service, and the remaining 5 checks query data from Prometheus. For simplicity, in each step during the experiment, we duplicated the same 8 checks. The engine itself does not cache requests or queries, thus there is no difference whether we would have, for each (re-)execution, queried for different metrics. We repeated each step in our experiment 5 times, and collected CPU and memory utilization data, as well as the raw duration of the strategy's enactment.

Results. As can be seen in Figure 9, we were not able to identify an upper limit of checks executed in parallel with our experimental setup. The engine's CPU utilization is slowly increasing for each step. However, even for 1600 checks executed in parallel, we did not reach full utilization. Given the slight increase for each step and the fact that we executed those checks on a single core machine, this indicates that in a more realistic context with more powerful resources, the engine could even handle higher amounts of checks and thus should be able to cover all realistic monitoring requirements.

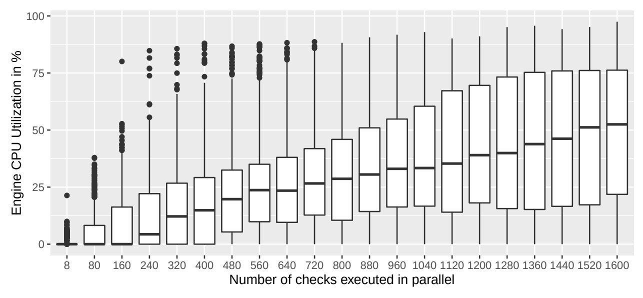

Figure 9: Boxplots of CPU utilization when executing an increasing number of checks in parallel for a single strategy.

As in the previous experiment, executing checks at the same time on a single core machine introduces a delay in the enactment of the strategy. This delay increases with the amount of checks executed in parallel and is depicted in Figure 10. When executing 1600 checks in parallel, this delay is roughly 50 seconds, which is, given the specified execution time of 120 seconds, quite high. Thus, the delay needs to be taken into account when defining a live testing strategy that uses a very high number of parallel checks. However, arguably, for most practical scenarios a much lower number of checks will be sufficient. In addition, and as before, deploying the engine to a larger cloud instance, specifically one with more virtual CPUs, is likely to mitigate this problem.

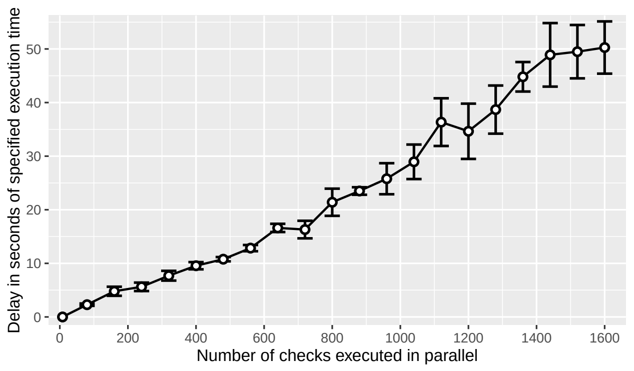

Figure 10: Delay in enacting a single release strategy with an increasing number of checks. Error bars represent ± one standard deviation.

5.3 Evaluation Summary and Limitations

Our experiments have shown a promising runtime behavior for BIFROST. We have shown that the overhead introduced by using BIFROST for live testing is only around 8 ms for most live testing strategies, even on low-end cloud instances. However, users need to keep in mind that specifically dark launches can substantially increase this overhead due to traffic duplication. We have also shown that our concept and prototype implementation is able to scale to very high numbers of parallel releases as well as parallel checks, indicating that our approach is suitable even for large companies with many parallel rollouts.

The main limitation of our study is that we have only conducted experiments on a single case study application. Hence, we cannot eliminate the possibility that our approach will have higher overhead or scale worse for other applications. Further, while realistic, our case study application was designed specifically for this experiment, and is not a real production application. Secondly, we have conducted our experiments in a virtualized environment (the Google Cloud Platform). It is possible that the performance variations inherent in public clouds [14] have influenced the results of our study. To mitigate this risk, we have repeated each experiment 5 times, and report the observed deviations.

6. RELATED WORK

BIFROST as a middleware for automated enactment of live testing strategies in microservice-based systems is strongly related to a number of ongoing trends and developments in modern software development. In an earlier paper, we have already argued for the importance of microservices in modern systems engineering [21]. Cito et al. [5] discuss that DevOps [2] and data-driven runtime decision making is a core factor in the development of state-of-the-art cloud applications. This has also been confirmed by Begel and Zimmermann [3], as well as by Kim et al. [11], who argue that data science is increasingly becoming a central element of the software development and release engineering process. Generally, this newfound interest in data and analytics is related to the current hype surrounding Big Data [17], as well as to the idea of continuous delivery and deployment [10]. Whereas continuous delivery primarily deals with shortened release cycles, as discussed for instance by Feitelson et al. for

Facebook [8], continuous deployment goes one step further and largely automates the deployment process [19]. Rahman et al. [18] have studied practices for continuous delivery, and also identified the practies we use (gradual rollouts, dark launches, canary releases, and A/B testing) as central. In our previous work, we have also identified continuous deployment as a prerequisite for live testing [22]. Conversely, being able to make use of live testing is an important payoff that motivates companies to widely automate their deployment process.

As discussed in Section 3.1, a core property of BIFROST is that release decisions are driven by runtime data. Hence, our proposed middleware can build on previous research on application performance management, such as Kieker [26] or our own previous work on the monitoring and management of Web application performance [6, 7]. Bakshy and Frachtenberg [1] have recently presented work on statistical methods to identify performance regressions in scale-out cloud systems, based on their experience at Facebook. Another contribution from the Facebook domain [24] describes how the company implements gradual rollouts and canary releases. Our basic model of live testing, as discussed in Section 3.2, is largely aligned with this description of real-life canary releasing. However, alternative approaches for canary testing, such as CanaryAdvisor [25], are also available. Another live testing approach for which substantial previous research is existing is A/B testing. Most importantly, Kohavi et al. have proposed a basic model as well as concrete guidelines [12, 13] on how to conduct statistically rigorous A/B tests for cloud applications. Tamburrelli and Margara [23] have rephrased A/B testing as a search-based software engineering problem, which they solve using a combination of aspect-oriented programming and genetic algorithms. BIFROST gives developers a structured way to conduct dark launches, canary releases, or A/B tests, and is fully compatible to the practices described in those earlier works.

In addition, our work is also related to some well-known open source toolkits related to CD and live testing. For instance, the Ruby-based Scientist! framework[10] is a simple library that allows a developer to encode A/B tests directly in code. The disadvantage of this model is that this way live testing code is tangled with the production code base. Further, adapting the configuration (e.g., going from one A/B test to another) requires changes in the application code. A non-intrusive tool that, similarly to BIFROST, builds on top of a microservice architecture to implement A/B testing and canary testing, is Vamp[11]. Unlike our work, Vamp does not support shadow launches or multi-phase rollouts. Another related tool is ION-Roller[12], which focuses on deployment using Docker images. It allows multi-phase rollouts, but only for simple Blue/Green deployment setups. Canary launches require manual monitoring, as it features rollback capabilities upon manual intervention. ION-Roller is a service consisting of an API, web app and CLI tool that orchestrates Amazon's Elastic Beanstalk to provide safe immutable deployment, health checks, traffic redirection and more. The main advantage of the BIFROST middleware over these existing systems is that it provides developers with a structured way and domain-specific language to arrange and automatically enact multi-phase live testing strategies, a principle that is as of yet largely unexplored.

[10]https://github.com/github/scientist
[11]http://vamp.io/
[12]https://github.com/gilt/ionroller

7. CONCLUSIONS

In this work, we proposed a formal model for defining live testing strategies covering four previously-identified methods of live testing (canary releases, dark launches, A/B tests, and gradual rollouts). On top of that, we provided a prototype implementation automatically enacting and executing multi-phase release strategies defined in a YAML-based domain-specific language. We evaluated our prototype in three experiments covering (1) the performance overhead introduced to systems when the BIFROST middleware is deployed, and identifying BIFROST's scaling capabilities when confronted with (2) a large number of multi-phase release strategies executed in parallel and (3) release strategies with a large set of continuously evaluated metrics and health checks. Even though our experiments were conducted on cheap public cloud instances, we have shown that the BIFROST middleware adds on average only 8 ms performance overhead when executing a multi-phase release strategy in comparison to a baseline application without BIFROST deployed. The BIFROST's engine is able to handle more than 100 release strategies at the same time on a single core machine and can cope with more than 1000 checks executed in parallel. Hence, we conclude that our approach can be used even in the scale of current-day industry leaders in continuous deployment. Our approach has a number of distinct advantages. Most importantly, formalizing release strategies in a DSL fosters transparency, and allows strategies to be shared, reused, and versioned. Further, additional verification and validation tools can be built on top of our work. While out of scope in this paper, this will be part of our future work.

Additionally, our future work needs to address a number of limitations of the current model and implementation. Most importantly, we are currently not modeling dependencies between services and versions. Similarly, we currently assume that all changes are forward and backward compatible, especially in terms of data schemas. Previous work [22] has shown that this is not necessarily the case. Finally, we currently assume that provisioning and load balancing service instances is handled outside of BIFROST. Future versions of the tool will be able to instantiate versions themselves, by interfacing with Infrastructure-as-Code tools such as Vagrant or Chef.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610802 (CloudWave), and from the Swiss National Science Foundation (SNF) under project Whiteboard (no. 149450).

Online Appendix

We provide additional material to this paper, including links to the source code of BIFROST, the case study application used in the evaluation, and a replication package for our study in an online appendix:

http://www.ifi.uzh.ch/seal/people/schermann/projects/bifrost.html

8. REFERENCES

[1] E. Bakshy and E. Frachtenberg. Design and Analysis of Benchmarking Experiments for Distributed Internet Services. In *Proceedings of the 24th International Conference on World Wide Web (WWW)*, pages 108–118, 2015.

[2] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, jun 2015.

[3] A. Begel and T. Zimmermann. Analyze This! 145 Questions for Data Scientists in Software Engineering. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 12–23, New York, NY, USA, 2014. ACM.

[4] L. Chen. Continuous Delivery: Huge Benefits, but Challenges Too. *Software, IEEE*, 32(2):50–54, Mar 2015.

[5] J. Cito, P. Leitner, T. Fritz, and H. C. Gall. The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 393–403, New York, NY, USA, 2015. ACM.

[6] J. Cito, P. Leitner, H. C. Gall, A. Dadashi, A. Keller, and A. Roth. Runtime Metric Meets Developer - Building Better Cloud Applications Using Feedback. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2015)*, New York, NY, USA, 2015. ACM.

[7] J. Cito, D. Suljoti, P. Leitner, and S. Dustdar. *Web Engineering: 14th International Conference, ICWE 2014, Toulouse, France, July 1-4, 2014. Proceedings*, chapter Identifying Root Causes of Web Performance Degradation Using Changepoint Analysis, pages 181–199. 2014.

[8] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and Deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[9] K. C. Foo, Z. M. J. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora. An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 159–168, Piscataway, NJ, USA, 2015. IEEE Press.

[10] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[11] M. Kim, T. Zimmermann, R. DeLine, and A. Begel. The Emerging Role of Data Scientists on Software Development Teams. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, pages 96–107, New York, NY, USA, 2016. ACM.

[12] R. Kohavi, A. Deng, B. Frasca, T. Walker, Y. Xu, and N. Pohlmann. Online Controlled Experiments at Large Scale. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*.

[13] R. Kohavi, R. M. Henne, and D. Sommerfield. Practical guide to controlled experiments on the web: Listen to your customers not to the hippo. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 959–967, New York, NY, USA, 2007. ACM.

[14] P. Leitner and J. Cito. Patterns in the Chaos – a Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Transactions on Internet Technology*, 2016.

[15] J. Lewis and M. Fowler. Microservices. http://martinfowler.com/articles/microservices.html, Mar. 2014.

[16] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37(4):316–344, Dec. 2005.

[17] F. Provost and T. Fawcett. Data Science and its Relationship to Big Data and Data-Driven Decision Making. *Big Data*, 1(1):51–59, 2013.

[18] A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin. Synthesizing Continuous Deployment Practices Used in Software Development. In *Proceedings of the Agile Conference (AGILE)*, pages 1–10, Aug 2015.

[19] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner, and M. Oivo. Continuous Deployment of Software Intensive Products and Services: A Systematic Mapping Study. *Journal of Systems and Software*, 2016. To appear.

[20] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm. Continuous Deployment at Facebook and OANDA. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE'16)*, pages 21–30, New York, NY, USA, 2016. ACM.

[21] G. Schermann, J. Cito, and P. Leitner. All the Services Large and Micro: Revisiting Industrial Practice in Services Computing. In *Proceedings of the 11th International Workshop on Engineering Service Oriented Applications (WESOA'15)*, 2015.

[22] G. Schermann, J. Cito, P. Leitner, U. Zdun, and H. Gall. An Empirical Study on Principles and Practices of Continuous Delivery and Deployment. PeerJ Preprints 4:e1889v1 https://doi.org/10.7287/peerj.preprints.1889v1, 2015.

[23] G. Tamburrelli and A. Margara. Towards Automated A/B Testing. In *Proceedings of the 6th International Symposium on Search-Based Software Engineering (SSBSE)*, volume 8636 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2014.

[24] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. 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.

[25] A. Tarvo, P. F. Sweeney, N. Mitchell, V. Rajan, M. Arnold, and I. Baldini. CanaryAdvisor: A Statistical-based Tool for Canary Testing (Demo). In *Proceedings of the 2015 International Symposium on*

Software Testing and Analysis (ISSTA), pages 418–422, New York, NY, USA, 2015. ACM.

[26] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE'12)*, pages 247–248, New York, NY, USA, 2012. ACM.