Continuous Experimentation for Software Developers

Schermann, Gerald

DOI: https://doi.org/10.1145/3152688.3152691

Continuous Experimentation for Software Developers

Gerald Schermann
Software Evolution and Architecture Lab
University of Zurich, Switzerland
schermann@ifi.uzh.ch

Abstract

Recent advances in build, test, and deployment automation not only enable companies shipping new functionality faster to their users, but also provide them the ability to experiment with functionality on small fractions of the user base first. These experiments involve techniques such as A/B testing, canary releases, or dark launches. However, neither managing multiple experiments in parallel (i.e., operating and monitoring multiple versions), nor specifying parameters for experiments (e.g., to avoid that they negatively impact each other) is a trivial task. In my research, I want to support developers and release engineers conducting experiments in an automated and data-driven way.

CCS Concepts • **Computer systems organization** → *Distributed architectures*; • **Software and its engineering** → *Software notations and tools*;

Keywords Continuous Experimentation, Release Engineering

ACM Reference format:
Gerald Schermann. 2017. Continuous Experimentation for Software Developers. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 4 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Problem Statement

The trend towards continuous deployment and delivery (CD) [9] enables companies, and especially Web-based companies, shipping new functionality faster and more frequently to their users, while at the same time keeping risks manageable [16]. Supported by a high degree of automation (e.g., build, test, and deployment), CD practices allow companies to take advantage of early customer feedback and reduced time to market [3]. However, shipping new functionality more frequently bears the risk that occasionally defective changes are released, or changes which do not satisfy the users' demands. Those problems have in common that they are likely to remain undetected in traditional testing environments (e.g., users' reactions to a new UI, performance regressions) as they only hit surface when facing production workloads [7]. Consequently, companies are making use of so called continuous experimentation techniques testing new functionality on small fractions of the user base in the production environment first. These experimentation techniques including A/B testing [11, 12], canary releases [9], dark launches [6, 19], and gradual releases [9] guide development activities based on data collected on a subset of the users and support companies in their release decisions, i.e., whether to roll back or continue rolling out new functionality to a larger user base.

The field of continuous experimentation is widely driven by well-known industry leaders such as Facebook, or Microsoft. Unfortunately, our knowledge is primarily based on the peculiarities and needs of those innovation leaders, excluding other companies and the challenges they face. As our exploratory, empirical study revealed [17], many companies conduct experiments. However, release monitoring, experiment design, and experiment result interpretation is mainly based on "gut feeling" rather than on solid, traceable processes. For instance, what metrics to collect to reason about a new release's health state is often based on intuition and experience. Many study participants did not have the data science knowledge for both experiment specification and result interpretation, or the resources to develop tooling for experimentation.

Conference'17, Washington, DC, USA
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

2 Proposed Research

In my research, I want to pave the way for companies of all sizes and various domains developing service-based applications, and fill this "knowledge gap" by providing tooling for conducting experiments and help release engineers and developers specifying them. The underlying hypothesis of my research is the following:

An explicit and formally defined model of continuous experimentation allows us to support software developers and release engineers conducting experiments by recommending experiment parameters (e.g., user group, duration) and verifying the feasibility of experiments (e.g., they do not negatively impact each other).

Having such a model would support companies and their developers and release engineers transitioning from "gut feeling-based" experimentation to explicit data-driven experimentation. In the following, I will motivate my research questions that are used to validate my hypothesis and discuss how they contribute to support the specification, verification, and execution of experiments.

In a first step, I need to understand the underlying characteristics of experiments. This includes getting an idea of the commonalities of the different experimentation techniques and how we can make use of them. Having a deeper understanding of both runtime and non-runtime aspects of continuous experimentation allows me to create a conceptual model of experimentation, which will serve as a common basis for both executing and verifying experiments.

RQ 1: What are the common characteristics of continuous experiments and how can we explicitly model them?

In addition, experimentation requires operating multiple versions of an application in parallel (e.g., a stable *checkout* service and an experimental, new implementation of it *fastCheckout*). During the course of the experiment, and based on the continuously collected monitoring data, user assignments to specific versions may change dynamically (e.g., further rollout of *fastCheckout*, or rollback to *checkout* because it does not behave as expected). Manually observing (i.e., services' health states, technical and business metrics) and administering (e.g., operating multiple versions in parallel, user reassignments) continuous experiments is a daunting task, especially for companies with multiple distributed teams independently working on their services. Those teams might even run their own experiments, which makes it also hard to keep track of

what experiments are executed, in which parts of the application, by whom, for how long, and at which scope. Therefore, in **RQ 2**, I plan to take the conceptual model of **RQ 1** as a basis and provide tooling for the automated, data-driven execution of experiments:

RQ 2: How can we support software developers and release engineers in conducting automated and data-driven experiments?

RQ 2a: How can we ensure that our approach scales and also works for large organizations with multiple distributed teams?

RQ 2b: How can we make experimentation explicit by using a domain-specific language, and thus foster awareness and traceability of experiments?

Once we are able to support the automated execution of defined experiments, we still lack solutions to help developers or release engineers to actually verify whether or not the experiment she or he intends to run is feasible. I want to investigate whether we can verify prior launching an experiment whether its underlying service configuration (i.e., the services and versions being part of the experiment) are actually compatible to each other. Moreover, different data sources (e.g., historic traffic, past experiments, current assignments) are available which might provide valuable resources for the recommendation of experiments on different levels of complexity (e.g., recommending single parameters, recommending entire experiments). Consequently, **RQ 3** is formulated as follows:

RQ 3: How can we verify and recommend experimentation?

Each of those research questions are directly related to the hypothesis. Finding a conceptual model of continuous experimentation in **RQ 1** is essential for Research Questions 2 and 3 as it lays the foundations for both the execution and the verification steps.

3 RQ1: Characteristics and Model of Experimentation

To address **RQ 1**, I plan to investigate the state of experimentation in companies of different sizes and across multiple domains.

3.1 The State of Experimentation

We conducted a study [17] to gain insights on how companies make use of experimentation practices. This includes investigating for which types of changes they conduct experiments, how they interpret the collected data of which metrics, for how long they typically run experiments, how they select user groups for experiments, how they implement experimentation in their application ecosystem, and who is responsible for the experiments. In addition, we wanted to shed light on the obstacles companies face when conducting experiments, or what prevents them from conducting experiments.

We investigated those questions in a mixed-method study consisting of qualitative interviews combined with a qualitative survey. We interviewed 31 software developers and release engineers of 27 companies and attracted 187 complete responses in an online survey. The outcome and observations of this study led to the underlying idea of this thesis. Two of the key findings which mainly influenced my thesis are:

"Gut Feeling-based" Experimentation: Aside from the aforementioned innovation leaders, our study has shown that experimentation is mainly driven by intuition rather than following a rigorous formal process. We observed that many release engineers are mostly going by their gut feeling and previous experience when defining metrics and thresholds to evaluate the success of experiments. Which features to conduct experiments on, or which users to consider for experiments is rarely based on sound statistical or empirical basis.

Technical Debt due to Feature Toggles: Feature toggles [8] are a common way of handling multiple versions in the same code base. However, interview participants stated that this increases complexity (e.g., maintenance) and might lead to technical debt. This is also confirmed by recent findings of Rahman et al. [15].

3.2 Conceptual Model of Continuous Experimentation

By asking how companies conduct experiments we learned about characteristics of continuous experimentation. This led to a first conceptual model of experimentation, which we presented in [18]. Besides experiments being *data-driven*, experimentation requires *timed*, *parallel*, and *ordered* execution. The conceptual model covers the services and the users being part of the experiment and maps to a state machine. Every single state of the state machine represents specific user assignments, e.g., which users are assigned to the *fastCheckout* service. In each state, a set of so-called *checks* is executed ensuring that the services under experimentation behave as expected. The outcome of checks then determines the subsequent state. This could even include "fallback" states in order to immediately react to problems with the tested service(s), e.g., reassign all users to the stable, previous version.

This abstract view on experimentation, having states and transitions, allows us to combine and chain multiple experimentation techniques to form, as we call it, *multi-phase experimentation strategies*. This initial conceptual model was used to implement our middleware Bifrost [18] to automatically execute such experimentation strategies (see **RQ 2**). However, this model covers just a small fraction of experimentation, it only considers those dimensions, i.e., services and user groups and their respective traffic, which are part of the executed experimentation strategy, thus the runtime aspect. Even though this is sufficient to execute the specified experiment, it does not prevent developers or release engineers from running multiple experiments at the same time that might overlap and negatively influence each other, e.g., the same user groups are part of multiple experiments which make it hard to interpret results and draw valid conclusions. Therefore, in order to tackle **RQ 3**, I also plan to investigate non-runtime aspects of experiments.

4 RQ2: Conducting Automated and Data-Driven Experiments

To support the automated, data-driven execution of continuous experiments we developed Bifrost [18]. Bifrost is a middleware implemented on top of the conceptual model presented in **RQ 1**. However, as formally specifying every experiment as a state machine is not feasible for developers or release engineers, we developed a domain-specific language (DSL) to make experiments explicit (as we call it *experimentation-as-code*), thus addressing **RQ 2b**. The Bifrost DSL is a YAML-based language, allowing developers and release engineers to version control experiments, and thus, to keep track of changes regarding those experiments, as well as, to easily reuse (parts of) them. The DSL describes the single phases of an experiment, i.e., what is monitored within the single phases and how the traffic is routed (i.e., user assignments) within those phases,

and which phases are executed when (i.e., start of phase based on conditions of the previous phase's outcome).

The Bifrost *engine* as the core component of the middleware depicted in Figure 1 takes the specified experiment submitted via the Bifrost command line interface as input, translates it into a state machine (see **RQ 1**), and executes it. During the execution it continuously monitors the specified metrics by querying multiple data sources (i.e., metrics providers) and (re-)evaluates its current state of execution.

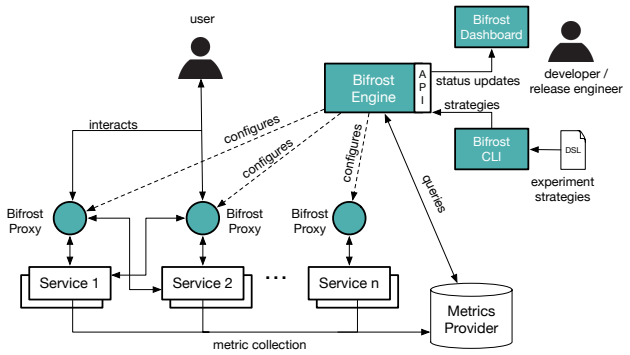

Figure 1. High level overview of the Bifrost middleware (Figure taken from [18]).

The *engine* configures the Bifrost *proxies*, lightweight components which are placed in front of the services which are part of the experiment. *Proxies* intercept each incoming HTTP request (either from users or other services), and depending on their configuration, requests are forwarded to a certain instance of a service version. The usage of *proxies* as instrument to route traffic allows us to move the routing logic from the code to our DSL and thus, avoids the usage of feature toggles and their downsides regarding complexity and maintenance.

To address **RQ 2a**, we demonstrated in our paper [18] that our approach can be used even on the scale of industry leaders in continuous deployment. We looked at (1) the performance overhead introduced to systems when the Bifrost prototype is deployed, and identified Bifrost's scaling capabilities when confronted with (2) a large number of multi-phase experimentation strategies executed in parallel and (3) strategies with a large set of continuously evaluated metrics and health checks. Even though we evaluated Bifrost on cheap public cloud instances, we have shown that the middleware adds on average only 8 ms performance overhead when executing a multi-phase strategy in comparison to a baseline application without Bifrost deployed. Bifrost is able to handle more than 100 experiments at the same time on a single core machine and can cope with more than 1000 checks executed in parallel.

5 RQ3: Recommending and Verifying Experiments

One cannot assume that engineers are trained data scientists, therefore I want to come up with approaches that support developers and release engineers in specifying experiments on a sound statistical basis. Running multiple experiments in parallel involves the risk that experiments influence each other and skew the results. I want to provide solutions to avoid such situations and provide appropriate tooling for developers and release engineers with limited data science skills. Given a set of experimentation strategies that are currently executed or scheduled to be executed and new functionality which should be tested, in my research I want to address *service compatibility* and *user group recommendation*.

5.1 Service Compatibility

Services being part of an experiment might not be compatible with each other. For instance, a newly introduced functionality might require (breaking) changes on other services. Consequently, it is important to verify service configurations (i.e., what services are part of the experiment and which version of each service) prior to launching an experiment, making sure that only valid configurations are deployed. I plan to investigate concepts known from software variability research, especially software product lines [4]. My idea is to transfer ideas of *feature models* [2] to the service domain and make use of SAT solving techniques [14] applied on these models. This would allow me to help the developer specifying valid service configurations and in case of incompatible services and versions, provide the developer recommendations how the configuration can be fixed to meet the constraints defined in the model (i.e., the dependencies of the various services and versions). Therefore, I would need to add a dimension of service dependencies to my conceptual model of experimentation. A technical evaluation of my approach will be based on multiple configurations and dependency models (i.e., number of services and their dependencies) of varying complexity to prove its functionality.

5.2 User Group Recommendation

Having a valid service configuration to host a new experiment, the next step is to select a proper user group to experiment with. Ongoing and scheduled experiments need to be taken into account making sure that parallel experiments do not negatively impact each other and a chosen user group provides enough data to reason about. Based on common statistical approaches (e.g., Kohavi et al. [12]) to determine a (minimum) sample size for experiments, both historic and current traffic, and other (scheduled) experiments, I want to programmatically determine and recommend suitable user groups for experiments. I want to define a fitness function taking these constraints into account and consider, for example, the application of genetic algorithms to identify a suitable user group allowing us to reason about a new functionality in minimum time with a certain level of confidence. The evaluation of this approach should mainly be based on simulation (e.g., different numbers of services, traffic, experiments in parallel).

6 Related Work

There exist multiple studies on the challenges companies face when adopting CD and experimentation. Those include experience reports from the perspective of single organizations on their way to CD (e.g., Chen [3]), but also studies involving multiple companies and their technical and organizational challenges (e.g., Leppanen et al. [13]). In our own work, we derived a model based on the trade-off between release confidence (i.e., the effort companies put into the quality gates in their development process) and release velocity (i.e., the pace with which they release new versions).

Moreover, there exist academic publications discussing how well-known industry companies conduct continuous experiments. Those include reports of Microsoft [10, 11] and Google [20]. Similarly, Fabijan et al. [5] derived a model detailing technical, organizational, and business evolution to provide a guidance towards data-driven

experimentation based on an investigation of the evolution of the experimentation process at Microsoft.

From a technical perspective, Tang et al. [19] provide details on how Facebook manages multiple versions running in parallel (e.g., using A/B testing) with a sophisticated configuration-as-code approach. Bakshy et al. [1] give insights how Facebook uses a DSL to separate experimentation design from application code. Recently, Veeraraghavan et al. [22] described Facebook's approach called *Kraken* to manage (i.e. route) live user traffic on various levels (i.e., data center, server) for identifying and resolving (performance) bottlenecks across their application ecosystem. Tarvo et al. [21] proposed a tool for automated canary testing incorporating data collection and analysis. Non-academic work includes open source tools such as Vamp[1] and Spinnaker[2]. Similar to our own tooling Bifrost, Vamp offers functionality to automate experiments specified in a DSL. Unlike Bifrost, Vamp does not support chaining multiple experimentation techniques to form multi-phased experiments. Spinnaker is tooling created to step in after the CI stage in a deployment process, and allows the creation of custom workflows including stages for experimentation and different strategies (e.g., canary rollout followed by a blue/green deployment).

7 Conclusion and Further Research

In my thesis, I want to support software developers and release engineers conducting continuous experiments by recommending experiment parameters (e.g., user groups) and verifying the feasibility of experiments (e.g., making sure that they do not impact each other). I want to support companies transitioning from "gut feeling-based" experimentation to explicit, automated, and data-driven experimentation. The expected contributions of my research include (1) a conceptual model of continuous experimentation incorporating both runtime aspects of ongoing and scheduled experiments and non-runtime aspects (e.g., service dependency dimension), (2) a proof-of-concept implementation to support the automated, data-driven execution of experimentation techniques combined to multi-phase strategies, and finally, (3) a proof-of-concept implementation to support the verification and recommendation of service configurations and experiment parameters (i.e., user group recommendation). While (1) and (2) are mainly covered, my current research focuses on the latter, the verification and recommendation of experiments.

A further vision is to combine these tools and set focus on the IDE as well, making the developer aware of on which parts of an application's code base experiments are executed (and under which configuration), and give them the chance to trigger experiments directly from within the IDE and get immediate feedback how a code change performs in the production environment (e.g., on a dedicated "dark launched" service instance for experiments receiving duplicated traffic).

Acknowledgments

I would like to thank Harald Gall and Phillip Leitner for supervision and support. The research leading to these results has received funding from the Swiss National Science Foundation (SNF) under project Whiteboard (no. 149450).

[1] http://vamp.io/
[2] https://www.spinnaker.io/

References

[1] Eytan Bakshy, Dean Eckles, and Michael S. Bernstein. 2014. Designing and Deploying Online Field Experiments. In *Proceedings of the 23rd International Conference on World Wide Web (WWW '14)*. ACM, New York, NY, USA, 283–292.
[2] Don Batory. 2005. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*. Springer, 7–20.
[3] Lianping Chen. 2015. Continuous Delivery: Huge Benefits, but Challenges Too. *Software, IEEE* 32, 2 (Mar 2015), 50–54.
[4] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional.
[5] Aleksander Fabijan, Pavel Dmitriev, Helena Holmström Olsson, and Jan Bosch. 2017. The Evolution of Continuous Experimentation in Software Product Development. In *International Conference on Software Engineering (ICSE)*. Buenos Aires.
[6] Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. 2013. Development and Deployment at Facebook. *IEEE Internet Computing* 17, 4 (2013), 8–17.
[7] King Chun Foo, Zhen Ming (Jack) Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. 2015. 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)*. IEEE Press, Piscataway, NJ, USA, 159–168.
[8] Pete Hodgson. 2016. Feature Toggles. http://martinfowler.com/articles/feature-toggles.html. (Jan. 2016).
[9] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional.
[10] Katja Kevic, Brendan Murphy, Laurie Williams, and Jennifer Beckmann. 2017. Characterizing Experimentation in Continuous Deployment: a Case Study on Bing. In *International Conference on Software Engineering, Software Engineering in Practice (ICSE SEIP)*. Buenos Aires.
[11] Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. Online Controlled Experiments at Large Scale. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.
[12] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M. Henne. 2009. Controlled experiments on the web: survey and practical guide. *Data Mining and Knowledge Discovery* 18, 1 (2009), 140–181.
[13] M. Leppanen, S. Makinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M.V. Mantyla, and T. Mannisto. 2015. The Highways and Country Roads to Continuous Deployment. *IEEE Software* 32, 2 (Mar 2015), 64–72.
[14] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*. Carnegie Mellon University, 231–240.
[15] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. 2016. Feature Toggles: Practitioner Practices and a Case Study. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 201–211. DOI: https://doi.org/10.1145/2901739.2901745
[16] Gerald Schermann, Jürgen Cito, Philipp Leitner, and Harald C Gall. 2016. Towards Quality Gates in Continuous Delivery and Deployment. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 1–4.
[17] Gerald Schermann, Jürgen Cito, Philipp Leitner, Uwe Zdun, and Harald C. Gall. 2017. We're Doing It Live: An Empirical Study on Continuous Experimentation. *Journal of Information and Software Technology* (2017). Under submission.
[18] Gerald Schermann, Dominik Schöni, Philipp Leitner, and Harald C. Gall. 2016. Bifrost: Supporting Continuous Deployment with Automated Enactment of Multi-Phase Live Testing Strategies. In *Proceedings of the 17th International Middleware Conference*. ACM, New York, NY, USA, Article 12, 14 pages.
[19] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, USA, 328–343.
[20] Diane Tang, Ashish Agarwal, Deirdre O'Brien, and Mike Meyer. 2010. Overlapping Experiment Infrastructure: More, Better, Faster Experimentation. In *Proceedings 16th Conference on Knowledge Discovery and Data Mining*. Washington, DC, 17–26.
[21] Alexander Tarvo, Peter F. Sweeney, Nick Mitchell, V.T. Rajan, Matthew Arnold, and Ioana Baldini. 2015. CanaryAdvisor: A Statistical-based Tool for Canary Testing (Demo). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, New York, NY, USA, 418–422.
[22] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. 2016. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 635–650.