ITER8: Online Experimentation in the Cloud

Mert Toslali
Boston University
toslali@bu.edu

Srinivasan Parthasarathy
IBM Research
spartha@us.ibm.com

Fabio Oliveira
IBM Research
fabolive@us.ibm.com

Hai Huang
IBM Research
haih@us.ibm.com

Ayse K. Coskun
Boston University
acoskun@bu.edu

ABSTRACT

Online experimentation is an agile software development practice that plays an essential role in enabling rapid innovation. Existing solutions for online experimentation in Web and mobile applications are unsuitable for cloud applications. There is a need for rethinking online experimentation in the cloud to advance the state-of-the-art by considering the unique challenges posed by cloud environments.

In this paper, we introduce ITER8, an open-source system that enables practitioners to deliver code changes to cloud applications in an agile manner while minimizing risk. ITER8 embodies our novel mathematical formulation built on online Bayesian learning and multi-armed bandit algorithms to enable online experimentation tailored for the cloud, considering both SLOs and business concerns, unlike existing solutions. Using ITER8, practitioners can safely and rapidly orchestrate various types of online experiments, gain key insights into the behavior of cloud applications, and roll out the optimal versions in an automated and statistically rigorous manner.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Software testing and debugging**; **Empirical software validation**; **Software evolution**.

KEYWORDS

performance, distributed systems, online experimentation, multi-armed bandit

ACM Reference Format:
Mert Toslali, Srinivasan Parthasarathy, Fabio Oliveira, Hai Huang, and Ayse K. Coskun. 2021. ITER8: Online Experimentation in the Cloud. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3472883.3486984

1 INTRODUCTION

Motivation. In our digital economy, organizations across all industries must deliver new software faster and with better quality to adapt to new demands and opportunities so as to remain competitive. Not surprisingly, we have been witnessing an increasing adoption of agile software development practices, including *continuous deployment* [8, 33, 58, 63] and *online experimentation* [1, 2, 5, 11, 22, 26, 31, 49, 55, 69].

In the traditional sense, online experimentation is a technique employed to test assumptions about or gain insights into the value delivered by new application versions exposed to users in production, aiming to identify the version that produces the best user experience or the highest revenue. Concretely, to solve this problem, practitioners use A/B or A/B/n tests, which entail: (1) deploying two or more competing versions of an application, with different features or configurations; (2) splitting users across versions; (3) collecting metrics related to user engagement and/or business results (e.g., conversion rate); and (4) determining, based on the data, which version is the best. Online experimentation of Web and mobile applications (i.e., code running on browsers or mobile devices) is widely adopted, supported by mature solutions [22, 55, 69], and has been extensively studied [1, 2, 22, 25, 26, 31, 49, 51, 55, 59, 60, 75].

Unfortunately, no study to date has investigated online experimentation of cloud applications (i.e., code running in the cloud), even though cloud applications directly affect user experience and business results. For example, Amazon reported that every 100 ms of latency costs them 1% in sales [19], and Google reported early on that an extra 500 ms in search results generation had significantly decreased user satisfaction, dropping user traffic by 20% [52]. Cloud applications' behavior is inherently volatile due to performance bugs, resource contention, and infrastructure-related failures, all of which can conflate with user experience and business results [4, 19, 24, 35, 38, 52].

We argue that existing solutions (and their underlying mathematical formulation) for online experimentation in the Web and mobile domains are unsuitable for cloud applications. Furthermore, and perhaps as a result, the deployment of cloud applications is an art, when contrasted with the scientifically inspired approach adopted in the Web and mobile domains. In particular, when deploying to the cloud, organizations aiming to be more agile resort to automation solutions that focus on the narrow problem of progressive rollout of a new application version, such as Flagger [23] and Argo Rollouts [6]. These systems, popular in the cloud-native computing community, gradually expose a newly deployed cloud application version to more user traffic while checking (in an ad hoc manner) if it satisfies acceptance criteria based on performance and correctness metrics—typically, latency and error rate. Although these systems can prevent a malfunctioning or slow version from being fully deployed by rolling it back if needed, they lack the mathematical sophistication necessary to assess and compare application versions, let alone optimize for business-oriented metrics.

The number of organizations (across many industries) releasing cloud applications at least once a week has been increasing [13], yet practitioners lack proper solutions to plan and automate new code releases methodically to optimize for business metrics under acceptable performance behavior. Thus, there is a need to rethink online experimentation for the cloud era, study it, and provide a practical solution to this timely problem.

Our work. In this paper, we introduce Iter8 [37], a system for online experimentation of cloud applications. Critically, Iter8 embodies our novel mathematical formulation devised specifically to tackle four key challenges posed by online experimentation in the cloud domain. First, cloud applications are expected to satisfy Service-level Objectives (SLOs), typically expressed in terms of performance and correctness metrics, such as tail latency and error rate. Therefore, in addition to considering a business metric to compare competing versions, as done in the Web and mobile domains, an experimentation system comparing multiple cloud application versions needs to factor in the SLOs promised to the users as well. Iter8 enables experiments with the following type of goal: *Among all versions satisfying SLOs, find the one that maximizes revenue (or user engagement).* Iter8's online Bayesian learning algorithm convolves the Key Performance Indicators (KPIs) of interest (from the SLO specification) and a business reward metric to be maximized when comparing and assessing competing versions of a cloud application.

Second, the cloud domain provides a challenging opportunity for an online experimentation system: APIs for changing programmatically how user traffic is split across cloud application versions. In traditional experiments in the Web and mobile domains, the distribution of users across competing versions is decided a priori by a practitioner and fixed for the entire experiment duration. Differently, Iter8 automatically adjusts the user traffic across competing cloud application versions, as more data becomes available for version assessments. As a result, a Iter8 practitioner can specify an experiment with the following goal: *Among all versions satisfying SLOs, gradually roll out the winner version, while discovering it, so that revenue (or user engagement) is maximized in the process.* Iter8's novel multi-armed bandit algorithm—PBR (Probabilist Bayesian Routing)—intelligently adjusts the user traffic split at each iteration of an experiment so that competing versions are explored with the goal of maximizing business reward *during the experiment*, while satisfying SLOs. This approach can be thought of as "earning as you learn about your versions." Furthermore, a variant of our PBR algorithm (PBR-split) enables practitioners to have Iter8 find the best version more quickly, when the goal is ending as fast as possible with the best version identified.

Third, practices not directly related to business metrics, such as canary releases (with or without progressive rollout) and dark launches, are common in the cloud domain. We frame such practices as online experiments where the goal revolves around validating experimental versions against SLOs. For instance, in Iter8, a canary release experiment with progressive rollout has two competing versions—the current version and the canary version. Iter8 assesses the canary version based on the SLOs and progressively exposes more users to it as more data becomes available, provided the canary consistently meets the SLOs. In the end, all users will see the canary version if it is validated; otherwise, all users will see the current version. Practitioners benefit from Iter8's statistically rigorous analysis (Bayesian learning) of the KPIs of interest during the experiment, as opposed to settling for the ad hoc approaches used by popular solutions in the cloud-native community.

Fourth, a cloud experimentation system must exhibit two fundamental properties: accuracy and repeatability. Accuracy means producing the correct experiment outcome. Satisfying both properties requires statistical rigor to (1) interpret the data properly, (2) be resilient to high variance due to cloud noise, and (3) avoid deciding prematurely on a winner version. Iter8's version assessments and traffic split decisions are based on statistically rigorous analysis of data accumulated throughout an entire experiment.

Our evaluation starts by studying Iter8's accuracy in the context of canary releases. Overall, we show that canary releases orchestrated by Iter8 result in correct outcomes 93% of the time, with an F1-score of 0.95. In contrast, the most popular systems for cloud-native canary releases and progressive rollouts, Flagger [23] and Argo Rollouts [6], achieve an accuracy of 55% and 53%, respectively. We then study Iter8's ability to maximize business reward (in A/B and A/B/n rollouts) by comparing our PBR algorithm with a state-of-the-art multi-armed bandit algorithm named Exp3 [16], as well as with the most popular strategy used in traditional A/B and A/B/n tests, represented by an algorithm that splits the traffic uniformly—UNIF [39, 59]. Our overall results reveal that Iter8 outperforms Exp3 and UNIF in terms of cumulative average reward by 6% and 8%, respectively. More importantly, Iter8 keeps the user traffic to the optimal version 93% of the time cumulatively, whereas Exp3 does so only 33% of the time. We also investigate Iter8's behavior when the online experiment goal is identifying the best competing version as quickly as possible, rather than maximizing business reward. To do so, we compare our PBR-split algorithm against Exp3 and UNIF and observe that Iter8 finds the best version with significantly fewer requests (served by the cloud application versions): Iter8 delivers a speed-up (in the number of requests) of 56% over Exp3 and 59% over UNIF to reach a 99% confidence level.

Summary. Our contributions are as follows:

- We design and implement Iter8, a first-of-its-kind system for online experimentation of cloud applications.
- We devise and propose a model that enables practitioners to craft and perform a wide variety of online experiments of cloud applications with a broad range of goals.
- We propose a novel mathematical formulation built on online Bayesian learning and our new multi-armed bandit algorithms (PBR and PBR-split) to enable online experimentation tailored for the cloud, considering both SLOs and business concerns, unlike existing solutions.
- We present extensive results evaluating Iter8.
- We make Iter8[1] available as open source to the cloud community.

2 BACKGROUND AND RELATED WORK

Online experimentation on Web and Mobile. The literature on online experimentation of Web and mobile applications (represented by A/B testing) is vast. Some of these works have studied A/B testing and reported on its benefits to business results and development agility from firsthand experience [21, 44, 45, 73], whereas many others proposed new techniques. Many of these techniques rely on statistical hypothesis testing and multi-armed bandit algorithms to dynamically change how users are split across competing versions [1, 2, 22, 25, 26, 31, 49, 51, 55, 59, 60, 69, 75].

However, no technique previously proposed takes into account SLOs when comparing versions and adjusting the user distribution dynamically during an experiment. In contrast, Iter8 enables experimenters to express SLOs that must be met, in addition to a business reward metric to be maximized, while it changes the user distribution to *explore* and *exploit* competing versions. Addressing the common need for cloud applications to meet SLOs is critical; failing to do so can hurt an organization's reputation, affect users' experience, and reduce revenue [19, 44, 52]. In our novel mathematical formulation, SLOs are stochastic feasibility constraints.

Agile practices and live testing in the cloud. Over the past two decades, enterprises have embraced a cultural shift to deliver code faster, as they move to the cloud and agile software development methodologies and tools mature [8, 33]. At the same time, a myriad of companies were born in the cloud era and helped accelerate the creation of an ecosystem of tools for agile *cloud-native* development [6, 14, 23, 68, 79]. The tools from the cloud-native computing community closest to our work are Flagger and Argo Rollouts, used for automating the progressive rollout of cloud applications. Flagger and Argo Rollouts assess a new version as it is rolled out, but do so in an ad hoc manner. In our extensive evaluation, we compare Iter8 with these tools (§6.1).

A few scientific publications have studied how practitioners use agile practices and live testing (A/B testing, canary releases, and dark launches) for cloud applications in the field [12, 58, 63, 66]. The cloud literature and practitioners equate A/B testing with mechanisms for splitting users across versions. However, as done in the Web and mobile domains, A/B testing encompasses both splitting users and comparing the competing versions to identify the one that maximizes business reward. Our work solves the A/B testing (and A/B rollout) problem for cloud applications.

Other prior works have proposed solutions to improve live testing [20, 48, 64, 74]. CanaryAdvisor [74] validates canary releases by checking performance and correctness metrics, whereas the work of Ernst *et al.* [20] focused on reducing the interference of canaries in the production infrastructure. More broadly, Schermann *et al.* [64] introduced Bifrost, a Domain-specific Language to model the specification of live testing strategies and a runtime for automating them. Similar to Flagger and Argo Rollouts, and unlike Iter8, these works lack statistical rigor for comparing and assessing competing versions accurately. WSMeter [48], on the other hand, is a statistical model to evaluate the performance of a collection of live heterogeneous jobs running on a massive production hosting environment. WSMeter defines a holistic performance metric to capture the overall performance of the entire environment, considering a wide variety of unevenly

[1]Software available from https://iter8.tools.

distributed jobs. WSMeter's approach and target problem are orthogonal to Iter8's.

Machine Learning (ML) model selection. Like cloud applications, ML models evolve and are versioned; hence, the emergence of systems for serving ML models in the cloud and research on model selection is not surprising. For example, Clipper [16] uses a multi-armed bandit algorithm named Exp3 [7] to dynamically change the user distribution across competing model versions while identifying the best version. Given Exp3's theoretical guarantees and practicality, we use it as one of our baselines for comparison with Iter8's algorithms (§6.2 and §6.3). Unlike Iter8, Exp3's underlying mathematical formulation does not consider SLOs.

Data-driven software development. The emerging role of data scientists in product teams [41] helps foster a data-driven development culture, which underlies online experimentation. Begel and Zimmermann [10] conducted surveys to understand software engineers' view on how data scientists can help them, and Cito *et al.* [12] interviewed software engineers to study development and operations changes created by cloud applications. These studies revealed that developers focus mostly on operational metrics (e.g., latency) to measure the quality of cloud applications, and are increasingly looking at business metrics (related to revenue and user engagement); however, the difficulty of interpreting the data to make informed decisions is an overarching concern. The surveyed engineers also expressed the importance of testing cloud applications in production without compromising performance by rolling back functionality if needed.

Iter8 addresses these concerns. It enables developers to automate the process of (1) interpreting performance and business metrics combined, (2) assessing and comparing competing cloud application versions, (3) identifying the best one, and (4) gradually rolling it out, all the while maximizing business outcomes and satisfying performance SLOs.

Offline testing. Some works advocate solutions for assessing the quality of cloud applications in offline test environments, isolated from production, by introducing automated benchmarking stages in continuous delivery pipelines and attempting to generalize the creation of benchmarks [27, 28]. Others propose solutions to emulate the complexity of large production environments to assess and improve its overall reliability (e.g., [50]). These techniques are orthogonal to and can coexist with Iter8. Online experimentation is not intended to replace offline tests (e.g., unit and integration tests). Instead, we see Iter8 experiments as additional safeguards for deploying new versions to production, while ensuring that SLOs are satisfied and reward is maximized. In addition, Iter8 can work with traffic engineering mechanisms, like traffic mirroring (shadowing), enabling users to assess versions in a staging environment using live traffic [37].

3 ITER8

In this section, we describe how Iter8 orchestrates online experiments for cloud applications, the types of experiments it supports, and its design and implementation.

As shown in Figure 1, a practitioner starts an experiment by creating an *experiment spec*, which declares the application and competing versions, SLOs (e.g., limits on tail latency and error rate), a reward metric (e.g., mean revenue), and the experiment's duration. Iter8 configures the initial user traffic split across versions (e.g., 5% to $v1$ and 95% to $v2$) using the underlying cloud platform's API. As the versions receive user traffic, observations of KPIs (e.g., latency and error rate) and reward are collected and stored in a telemetry database. Iter8 periodically queries the database to build belief distributions for version-KPI and version-reward pairs (§4). As the experiment proceeds and more data is available, the belief distributions converge to the true values of the KPIs and reward. At each experiment iteration, Iter8 adjusts the user split across the competing versions using its traffic shifting algorithm (§5), taking into account the belief distributions and SLOs. At each iteration, if a version does not meet all SLOs, its *effective reward* is zero; conversely, versions satisfying all SLOs have an *effective reward* equal to the observed value of the reward metric. Thus, the fraction of users exposed to non-compliant versions decreases over time as the fraction exposed to the most rewarding version increases (§5). The experiment continues for its defined duration or until its goal has been achieved (e.g., winner version found with 95% confidence), whichever comes first.

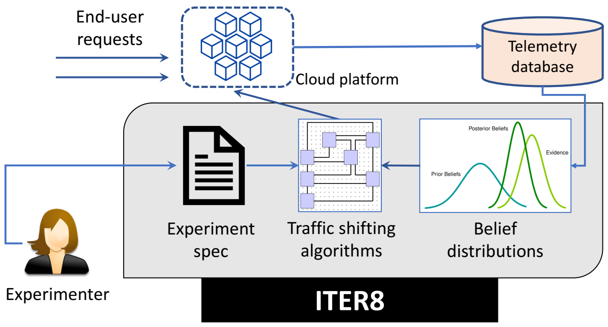


Figure 1: Iter8 overview.

3.1 Experiment spec

We now summarize the key elements of an experiment spec. The inset below shows a simplified sample spec in the YAML format for an **A/B rollout** experiment. That spec identifies the target cloud application *my-app* in the namespace *my-team* and two competing versions, named *my-app-v1* and *my-app-v2*. The spec declares two SLOs to be satisfied—the

95th latency percentile must be below 100 ms and the error rate must be below 1%— and a business reward metric (mean revenue) to be maximized. The experiment is configured to run for 100 iterations of 30 seconds each. Under *trafficControl*, the spec indicates a strategy that corresponds to one of Iter8's two traffic shifting algorithms (§5) and sets the parameter *maxIncrement* to 10 so that Iter8 does not increase the user traffic to a version by more than 10% at each iteration. Following this spec, Iter8 will progressively shift the user traffic towards the version that has the highest mean revenue and that satisfies the SLOs on the 95th percentile of latency and error rate.

```
spec:
 type: A/B
 target: my-team/my-app
  baseline: my-app-v1
  candidates:
  - my-app-v2
 criteria:
  objectives:
  - metric: 95th-latency
   upperLimit: 100
  - metric: error-rate
   upperLimit: 0.01
  rewards:
  - metric: mean-revenue
   preferredDirection: High
 duration:
  intervalSeconds: 30
  iterations: 100
 trafficControl:
  strategy: reward-maximization
  maxIncrement: 10
```

Iter8 supports two patterns for splitting the user traffic across competing versions: **progressive** or **fixed**. In the progressive pattern, shown in the sample, the experiment spec indicates a traffic control strategy that translates into one of Iter8's traffic shifting algorithms. In these scenarios, Iter8 has full control and progressively shifts the user traffic towards the winner version over multiple iterations. In the fixed pattern, the experiment spec provides an explicit user traffic split that remains unchanged for the entire experiment.

Next, we describe the experiment types enabled by Iter8 using our terminology for online experimentation of cloud applications.

Canary release. This experiment involves a baseline version, a candidate version (the canary), and SLOs. If the candidate is validated (i.e., it meets the SLOs), Iter8 will declare it the winner. A canary release experiment can use either the progressive or the fixed traffic control pattern. In the progressive case, the traffic will shift towards the candidate if it is validated, or towards the baseline version otherwise.

A/B rollout. An A/B rollout involves a baseline version, a candidate version, a reward metric, SLOs (optional), and the progressive traffic control pattern. If both versions are validated, the version that optimizes the reward is the winner. If only a single version is validated, this version is the winner. If no version is validated, there is no winner. The traffic will shift towards the winner, or the baseline if there is no winner.

A/B/n rollout. An A/B/n rollout involves a baseline version, two or more candidate versions, a reward metric, SLOs (optional), and the progressive traffic control pattern. The winner of the experiment is the version that optimizes the reward among the subset of versions that are validated. The traffic will progressively shift towards the winner, or the baseline version if there is no winner.

A/B test. Unlike an A/B rollout, an A/B test relies on the fixed traffic control pattern.

A/B/n test. Unlike an A/B/n rollout, an A/B/n test relies on the fixed traffic control pattern.

Conformance test. It involves a single version (baseline) and SLOs. The goal is to check if that version meets the SLOs.

Traffic engineering features. Iter8 enables three traffic engineering techniques. First, experimenters can perform **user segmentation**, a technique to select a specific segment of the user population for an experiment, leaving the remaining users unaffected. Second, during A/B, A/B/n, or canary experiments, **session affinity** is often necessary to ensure that the version to which a particular user's request is routed remains consistent during the experiment. Third, the technique of **traffic mirroring**—replicating real user traffic—enables experimenting with a *dark-launched* version with no impact on users. Metrics are collected and evaluated for the dark version, but end users do not see that version.

3.2 Design and implementation

Iter8 enables online experimentation of applications running on any cloud based on Kubernetes [47], which is the technology used by many public and private cloud offerings. Many platforms have been built on top of Kubernetes to augment Kubernetes' mechanisms for metrics collection and extend its APIs with high-level abstractions for developers. Relevant are APIs for traffic management, which Iter8 uses to distribute users across competing versions and effect the traffic engineering techniques explained above. Iter8 currently supports the following underlying platforms: (1) Istio [36], the current most popular service mesh; (2) Knative [42], a platform for serverless containers and event-driven applications; and (3) KFServing [40], a platform for serving ML models on arbitrary frameworks. Architecturally, Iter8 has two key components, which we describe next.

Iter8-controller. This component orchestrates all ongoing experiments. It keeps track of experiments' status, duration, and iterations. At each iteration, Iter8-controller calls Iter8-analytics to get an update on a particular experiment—its status, version assessments, and the new user traffic split. It then applies the user traffic split recommended by Iter8-analytics using the underlying platform API. This control loop continues until the experiment ends.

Since Iter8 is designed for Kubernetes-based clouds, we implemented Iter8-controller as a Kubernetes controller in Golang, following the controller pattern [15]. We extended Kubernetes with a Custom Resource Definition (CRD) [17] named *Experiment* whose structure matches our declarative experiment spec (§3.1). To start an experiment, a Iter8 user first creates a YAML file representing an *Experiment* CRD instance and then submits it to the target Kubernetes cloud. After Kubernetes creates the resource, ITER8-controller starts the corresponding experiment.

Setting and updating the user traffic split is a platform-dependent action. For example, ITER8 manipulates Istio's VirtualService and DestinationRules [76] in the Istio service mesh, and Knative Service [77] in Knative. We designed ITER8 for multi-platform extensibility.

ITER8-analytics. This component encapsulates our Bayesian online learning (§4) and multi-armed bandit traffic shifting algorithms (§5). The former builds belief distributions for KPIs and rewards, and assesses each competing version against SLOs, whereas the latter takes the belief distributions and SLOs as inputs and decides how to adjust the user traffic split across versions.

We designed ITER8-analytics as a REST API server written in Python. At each experiment iteration, ITER8-controller makes an HTTP call to ITER8-analytics to get updated information. To build the belief distributions, ITER8-analytics queries the cloud platform's telemetry database. Any telemetry database that enables HTTP queries can be used by ITER8. ITER8-analytics converts the data from a telemetry-specific format to a common internal format used by the ITER8-controller. Currently, ITER8 supports Prometheus [56] (a popular time-series data store), Sysdig [70], and New Relic [53].

Many metrics are automatically collected by the underlying platform and stored in a chosen telemetry database. For example, the Istio service mesh collects data on latency and error rate and can export it to Prometheus. When ITER8 is installed, it comes with configuration for the default metrics of the target platform. We have extended Kubernetes with a ITER8 metric CRD to make it easier for users to enable ITER8 to work with their own custom metrics.

4 ONLINE LEARNING IN ITER8

We now present our online Bayesian learning mathematical formulation. We first describe how ITER8 derives a multivariate utility function (§4.1) to convolve the KPIs (from the SLOs) and a reward metric, which is the basis for *effective reward* (§3). We then explain how ITER8 progressively updates belief distributions for KPIs and reward to learn the utility function during an experiment (§4.2 and §4.3).

4.1 Multivariate utility function

The experiment spec is represented in ITER8 as a multivariate utility function. This serves two purposes. First, it combines multiple KPIs into a single utility function, which is learned and optimized online during the experiment. Second, it enables ranking the competing versions, with the winner being the version with the maximum utility.

In order to describe this function, we use the following mathematical notation: Let $X_0[p]$ denote the reward KPI for version p, let $X_1[p], \ldots, X_k[p]$ denote the KPIs for version p, and let $\ell_1, \ldots, \ell_k$ denote their respective SLOs. For each $j \in 0, \ldots, k$; $X_j[p]$ is the random variable denoting the value of version-level KPI; $\mathbb{E}[X_j[p]]$ denotes the expected value of this version-level KPI. This is generally unknown to ITER8 at the experiment start and is learned online through KPI observations. Here, ℓ_j is a fixed constant which is specified in the experiment spec. For example, in the inset in §3.1, $X_0[p]$ denotes the `mean-revenue` KPI, $X_1[p]$ denotes the latency KPI of version p, and $\ell_1 = 100$ is the SLO on the latency.

We first define *feasibility*; we say that version p is *feasible* if it satisfies all the SLOs provided in the experiment spec. More formally, feasibility is defined in Eq. (1) below:

$$\forall j \in 1, \ldots, k : \mathbb{E}[X_j[p]] \leq \ell_j. \tag{1}$$

ITER8 then derives the *utility* function ($h_a(p)$) for version p. The utility function is presented in Eq. (2) below, where $\mathcal{F}$ denotes the set of feasible versions.

$$h_a(p) = \mathbb{E}[X_0[p]]\mathbf{1}_{p \in \mathcal{F}} \tag{2}$$

The intuition behind the multivariate utility function is as follows. If the expected KPIs of a version p are within their respective limits, $h_a(p)$ equals the expected reward of p; otherwise, if even one of p's expected KPIs violates its limit, then $h_a(p)$ becomes 0. We emphasize that, like the expected values $\mathbb{E}[X_j[p]]$ of version-level KPIs, the version-level utilities $h_a(p)$ are *fixed but unknown quantities*. ITER8 progressively estimates these quantities using KPI observations.

4.2 Belief update

ITER8 associates a belief (probability) distribution with every version-KPI pair ($[p, j]$) and updates these distributions periodically at the end of each iteration of an experiment based on the new batch of telemetry data. The batched updates are a practical necessity since telemetry updates are not instantaneous in the real world but often happen with delays and over batches of requests [56].

We now describe the mechanics behind belief updates using the following idealized scenario. Suppose we have a coin, which turns up heads with probability q. Further, suppose we do not know the true value of q, which equals 0.65. In Bayesian inference, q as a random variable. We can model our uncertainty about q using a prior belief distribution. $Beta(1, 1)$ is a reasonable prior when we start with no information about q as it distributes q uniformly in $[0, 1]$. If we toss the coin 100 times and observe 70 heads in the sample, we can justifiably conclude the 'true' value of q is likely to be close to 0.7. Formally, we use Bayes' theorem to update our belief for q to obtain a *posterior distribution* for q, which is now $Beta(1 + 70, 1 + 30)$. This is an instance of the classic Beta-Binomial belief update model illustrated in Figure 2.

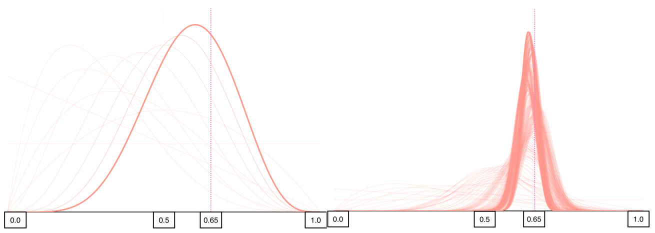


Figure 2: Beta-Binomial belief update. The true value of q is 0.65. The beta belief distributions on the left and right correspond to 10 and 400 observations, respectively. On the right we see less variance and a mean closer to the true value.

Many KPIs (e.g., error rate and click-through rate) are averages of 0/1 binary indicator random variables (e.g., 1 if the user clicked on a link). We refer to these KPIs as indicator KPIs. Let experiment iteration be indexed $1, 2, \ldots$. Let $Beta_e(\alpha_e[p, j], \beta_e[p, j])$ denote the belief distribution associated with $[p, j]$ at the start of iteration e. Iter8 uses the Beta-Binomial belief update for indicator KPIs.

Other KPIs, such as average revenue or average latency, may not be averages of 0/1 random variables and require a more general approach. We refer to these KPIs as continuous KPIs. Let j be a continuous KPI. Let a_j and b_j represent the minimum and maximum possible values that can be observed for KPI j. The distribution $Beta_e(\alpha_e[p, j], \beta_e[p, j])$ captures the uncertainty about the *normalized expectation* $\frac{\mathbb{E}[X_j[p]]-a_j}{b_j-a_j}$. At the start of the experiment, this distribution is initialized to $Beta(1, 1)$. Let $Z^+_{e-1}[p, j]$ denote the sum of all values observed for version p and KPI j, and let $W^+_{e-1}[p]$ denote the total number of requests routed through version p until the end of iteration $e - 1$. Then, the update equations for KPI j for version p for any iteration $e > 1$ are as follows:

$$\alpha_e[p, j] = 1 + \frac{Z^+_{e-1}[p, j] - a_j W^+_{e-1}[p]}{b_j - a_j} \tag{3}$$

$$\beta_e[v, j] = 1 + \frac{b_j W^+_{e-1}[p] - Z^+_{e-1}[p, j]}{b_j - a_j} \tag{4}$$

Statistical intuition behind belief update. The update Equations (3) and (4) account for several important theoretical and practical considerations.

1) It is easy to verify that the Beta-Binomial updates for indicator KPIs can be recovered as a special case of (3) and (4) by setting a_j and b_j to 0 and 1 respectively. This is not surprising, considering the fact that indicator KPIs are a special case of continuous KPIs.

2) The *posterior mean* (i.e., the mean of the updated Beta belief distribution) for the version-KPI pair $[p, j]$ equals $\frac{a_j+b_j+Z^+_{e-1}[p,j]}{2+W^+_{e-1}[p]}$. By the strong law of large numbers [18, 65], this value approaches the true expectation $\mathbb{E}[X_j[p]]$ almost surely as more requests are routed through p, i.e., as the *effective sample size* $2 + W^+_{e-1}[p] \to \infty$.

3) The *variance* of a Beta distribution with parameters α and β can be computed analytically as $\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$ [18, 65]. It follows that the posterior variance of the version-level KPI $[p, j]$ equals $(b_j - a_j)^2 \frac{\alpha_e[p,j]\beta_e[p,j]}{(\alpha_e[p,j]+\beta_e[p,j])^2(\alpha_e[p,j]+\beta_e[p,j]+1)}$. It is also possible to verify that this variance goes to 0 as the effective sample size $2 + W^+_{e-1}[p] \to \infty$.

4) Iter8 can easily incorporate new version(s) deployed in the middle of an experiment using the belief update mechanisms. It accomplishes this by initializing and updating the belief distributions associated with the new version(s) as in Eq. (3) and (4). The belief distributions for the new version(s) will exhibit greater variance initially, and will converge to their true expectations as more requests are routed, and KPI observations are collected for these versions.

5) By the same token as **4)** above, Iter8 can also easily incorporate historical data on versions that were running before the experiment began.

6) The update Equations (3) and (4) do not rely on individual request-level KPI observations but on aggregated KPI observations batched for each iteration. This enables Iter8 to generalize to heterogeneous cloud environments where gathering instantaneous KPI observations is not an option, but batched over windows of time ([34, 56]).

The belief updates in Eq. (3) and (4) are examples of approximate Bayesian computation that bypass the likelihood functions for the KPIs (i.e., the distributions of the random variables $X_j[p]$) and directly estimate expectations of the KPIs (i.e., $\mathbb{E}[X_j[p]]$). Several approximate Bayesian computation methods have been proposed in the literature [9, 78]. We choose the belief update in Eq. (3) and (4) due to its simplicity and the fact that similar update methods have been recently shown to have provably good convergence properties [3].

4.3 Monte Carlo sampling

Given the KPI observations until now, how can we quantify the probability of a specific version p being the optimal version? In this section, we describe how Iter8 handles these questions via Monto Carlo sampling [18, 65]. We start with the procedure for creating a single utility sample for a given version p, a key subroutine for other estimation tasks.

Utility sample for p. Recall that $Beta_e(\alpha_e[p, j], \beta_e[p, j])$ is associated with version-KPI pair $[p, j]$ and represents the posterior belief distribution for the normalized expectation $\frac{\mathbb{E}[X_j[p]]-a_j}{b_j-a_j}$. We can sample a value $\hat{y}_j[p]$ from

$Beta_e(\alpha_e[p, j], \beta_e[p, j])$, and compute the value $\hat{x}_j[p] = a_j + (b_j - a_j)\hat{y}_j[p]$, which is our sample for $\mathbb{E}[X_j[p]]$. Similarly, we can sample $\hat{x}_0[p], \ldots \hat{x}_k[p]$ for KPIs $0, \ldots, k$ respectively, substitute them in (2) in place of the corresponding $\mathbb{E}[X_j[p]]$ values, and obtain a single utility sample $\hat{h}_a[p]$ for version p.
Posterior probability of p. This quantity represents the probability of version p being the optimal. Using the above procedure, we create a sample utility vector that contains a single utility sample for each version in the experiment. Iter8 estimates the posterior probability of p by sampling a large number of such utility vectors (e.g., 10000), and computing the fraction of vectors in which p emerged as the version with the maximum utility.

Based on the above formulation, Iter8 surfaces a variety of useful insights from Monte Carlo sampling and Bayesian assessment during an experiment and after its termination. These insights include the probability of a version being the winner (posterior probability of p), the probability of a version improving over the baseline with respect to a given metric, the probability of a version being the best version with respect to a given metric, and the range of likely values for the KPI of a version (i.e., the credible interval).

5 ITER8'S DECISION ALGORITHMS

Online experimentation presents a fundamental tradeoff between exploration of available options versus exploitation of the option currently considered the best one. The multi-armed bandit problem well exemplifies this tradeoff. Consider a gambler who faces a slot machine with multiple arms, each of which produces a random payout when pulled. The gambler wishes to maximize the total payout through a sequence of arm pulls. Should the gambler aggressively exploit an arm, which is the current best arm, or should the gambler broadly explore the set of available arms (and risk losing payout)? In Iter8, the experimenter (gambler) wishes to maximize reward (payout) among various versions (arms).

Thompson Sampling, which chooses the arms probabilistically, is a heuristic for such explore/exploit problems [75]. The Thompson Sampling algorithm provides assorted benefits over classical multivariate hypothesis tests or A/B experiments, both of which uniformly evaluate competing variants [59, 61]. The algorithm has strong theoretical guarantees on how quickly it will converge to an optimal solution [3, 60].

The algorithms we develop in this paper, PBR and PBR-split, significantly generalize the classic Thompson Sampling [59, 75]. They incorporate stochastic feasibility constraints (SLOs), a multivariate utility function, and approximation Bayesian computation (§4). In particular, they sample a large number of utility vectors for each competing version using Monte Carlo sampling (§4.3). For each vector, the algorithm (PBR) picks the best (or two best in PBR-split) version according to sample utility, and marks this version as a candidate. In the traffic policy, the weight given to a version p is equal to the fraction of vectors in which p emerged as the candidate. We now discuss the behavior of our algorithms.
Reward maximization (PBR): PBR creates an adaptive and iterative traffic policy where traffic is routed to a version proportional to its posterior probability. As the experiment progresses, the posterior probability of suboptimal versions converge to 0, PBR progressively shifts traffic towards the optimal version as desired, hence maximizing the reward during the experiment.
Best-version identification (PBR-split): PBR-split creates a traffic policy where traffic is progressively shifted towards the two best versions ranked by the utility. The idea behind PBR-split is to aggressively support Iter8's quest to separate the truly optimal version from the second-best version (and, therefore, all the rest). By doing so, PBR-split sharpens posterior probabilities, hence enabling quicker identification of the best variant in an experiment.

6 EXPERIMENTAL EVALUATION

We now present a quantitative comparative analysis of Iter8. We first assess Iter8's accuracy, its ability to solve the reward maximization problem, and its ability to identify the best competing version as quickly as possible. We conduct these experiments running two types of benchmark applications in a public cloud—a distributed microservice-based application and an ML model serving system. Our evaluation concludes with simulations to study our algorithms with randomized KPIs and an increasing number of competing versions.

We summarize our public cloud setup below. Comparison baselines and benchmark applications are detailed in later sections (§6.1 and §6.2).
Public cloud setup. We conduct our experiments on a 14-node Kubernetes cluster (version 1.17.11) [47] in a multi-tenant public cloud. Each Kubernetes worker node is a VM. Our cluster has various worker node flavors to represent a heterogeneous and realistic production environment. The distributed microservice-based benchmark application runs on 13 nodes, 4 of which have 8 CPU cores and 64 GB of memory; the remaining nodes have 4 CPU cores and 16 GB of memory. The ML model serving system runs on a GPU pool with 2 Tesla P100 physical cards. Each card has 2 GPUs, 32 CPU cores, and 384 GB of memory. We use the Istio service mesh [36] for traffic management and Prometheus [56] as the telemetry database.
Load generation. We exercise the benchmark applications using the hey HTTP load generator [30]. The nature of the generated load varies depending on the experiment. We use constant, diurnal, and unpredictable loads with spikes in user demand similar to the methodology used in FIRM [57].

6.1 Accuracy

In this section, we evaluate Iter8's accuracy. We define ***accuracy*** as the ratio of correct experiment decisions over the total number of experiments. In this definition, the term experiment refers to online cloud experiments, such as canary releases and A/B rollouts. For example, in a canary release where the canary version is supposed to meet the SLOs, the correct decision is to validate the canary version instead of rolling back to the baseline.

Benchmark application. In our accuracy evaluation, we use the Train Ticket [80] benchmark application, which is widely used and the largest publicly available microservice-based application. Train Ticket is modeled after a travel ticket booking system that enables users to query the trains' schedule, reserve train tickets, and pay for them. It was designed and implemented as a distributed application comprising 41 microservices spanning 4 programming languages, using MySQL and MongoDB for data persistency. We chose one microservice, named *travel-service*, to be subjected to online experimentation. Among the 41 microservices, *travel-service* makes the largest number of downstream calls.

Comparison baselines & methodology. We compare Iter8 against Flagger [23] and Argo Rollouts [6], which are the most popular solutions for automating canary releases with progressive rollouts in the cloud-native computing community. To analyze the accuracy of Iter8, Flagger, and Argo Rollouts, we perform canary releases on *travel-service* under a variety of different situations. To create two versions (baseline and canary) of *travel-service* for this set of experiments, we modified the original code as follows. We increased the canary's latency based on a normal distribution with a mean 10 ms, and we also made the canary version return HTTP errors randomly with a probability of 0.025, while the baseline version returns no errors. The latency and error rate metrics are collected by the Istio service mesh; no code instrumentation is needed. Table 1 shows the mean latency and error rate for the baseline and canary versions of *travel-service*.

Naturally, when comparing Iter8 against Flagger and Argo Rollouts, we use the same SLOs specification. However, Flagger and Argo Rollouts require an additional *threshold* parameter to control the total number of SLO violations that the canary is allowed to make before they decide to roll back to the baseline version. We use their default value of 5 for all experiments, except in those where we vary that parameter. Furthermore, Flagger and Argo Rollouts require the user to explicitly set the traffic percentage increase to the canary version at each experiment iteration. In our experiments, we configure Flagger and ArgoRollouts to shift 10% of the user traffic to the canary at each iteration. In Iter8, traffic-shifting decisions are made automatically, but the user can limit the maximum traffic increase per iteration. We configure Iter8 so that the traffic shift to the canary version is no more than 10% per iteration. All canary release experiments are configured to last for 10 iterations. (Note that we use the words step and iteration interchangeably.)

Table 1: KPIs for *travel-service* baseline and canary versions.

name	latency (ms)	error-rate
baseline	110	0
canary	120	0.025

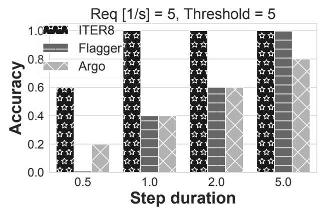
Quantitative results. In Figure 3, we summarize the results of our comparative accuracy analysis. Figure 3a shows the accuracy for different values of step (i.e., iteration) duration: 0.5, 1, 2, and 5 minutes. We generate a load of 5 requests/s for a total of 150, 300, 600, and 1500 requests per step. In these experiments, we specify that the canary release needs to satisfy an SLO on the mean latency KPI ($\leq \ell_{lat} = 140ms$). As mentioned earlier, the violation threshold for Flagger and Argo Rollouts is set to the **default** value (if the number of violations $\geq$ 5, they roll back to the baseline version).

Figure 3a reveals that Flagger and Argo Rollouts accuracy suffers for short iteration durations. These systems' lack of statistical rigor is compounded by their considering KPI observations on a per iteration basis. Shorter iterations translate into fewer observations. In contrast, Iter8 not only applies a statistically rigorous online Bayesian learning algorithm, but it also uses KPI observations accumulated for the entire experiment duration to learn the belief distributions.

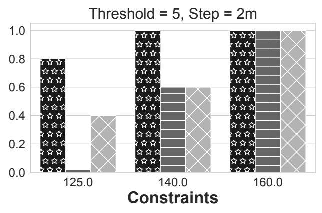
Practitioners need to consider the number of data points available to an online experimentation system and how it affects its accuracy. For Flagger and Argo Rollouts, that number is a function of load and step duration; for Iter8, it depends on the load and the entire experiment duration. Note, however, that Iter8 can end an experiment as soon as the desired level of statistical confidence is achieved, which can happen before the configured experiment duration.

In the experiments reported in Figure 3b, we vary the SLO on mean latency by setting the limit to 125, 140, and 160 ms. As shown, tighter SLOs severely impact Flagger and Argo Rollouts. This result stems from their reliance on the *threshold* parameter to decide when to roll back, as opposed to Iter8's approach of making decisions based on the learned KPI belief distributions. Figure 3c shows the impact of an additional SLO on the error rate KPI ($\leq \ell_{err} = 0.03$). Additional SLOs further impact Flagger and Argo Rollouts. Although their *threshold* parameter is defined on a per KPI basis, the more KPIs, the higher the likelihood that these systems will make a decision based on a threshold violation.

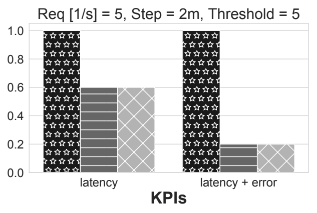
Next, we increase Flagger and Argo Rollouts threshold parameter values. In Figure 3d, we show that a more permissive



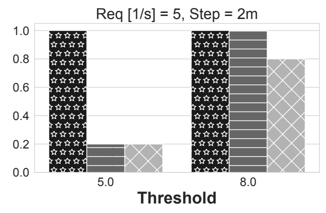
(a) Varying step durations



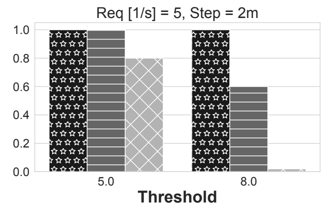
(b) Varying SLOs



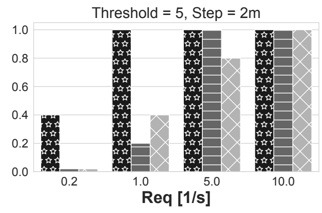
(c) Varying KPIs



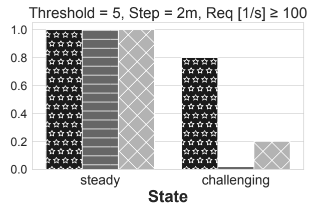
(d) Varying thresholds (ok canary)



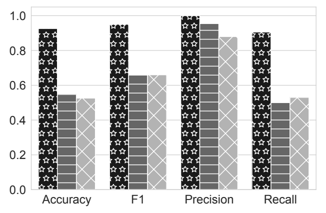
(e) Thresholds (bad canary)



(f) Varying load



(g) Steady vs challenging



(h) Overall evaluation

Figure 3: Accuracy in canary releases. The accuracy results are summarized for varying step durations, SLOs, number of KPIs, thresholds, request load, and patterns; ITER8 (in black '⋆'), Flagger (in grey '-'), and Argo (in light-grey 'x').

threshold benefits these systems' accuracy when the canary version is supposed to be validated. However, when the canary code is not compatible with the SLO ($\leq \ell_{err} = 0.02$), higher threshold values lead to less accuracy, as shown in Figure 3e. It is hard for practitioners to decide on the threshold, which limits the trustworthiness of Flagger and Argo Rollouts. Instead of relying on ad hoc threshold parameters, ITER8 uses our Bayesian approach to analyze the KPIs and make decisions, thereby guaranteeing statistical rigor and reducing the users' cognitive load.

Figure 3f shows results for different load intensities. Fewer requests/s translate into fewer KPI observations. Therefore, as expected, the results in Figures 3f and 3a exhibit the same trend for the same reasons.

In Figure 3g, we study the impact of high load and high variance. The results grouped for the label *steady* were obtained with a constant load of 100 requests/s and permissive SLOs ($\ell_{lat} = 140ms$ and $\ell_{err} = 0.04$). We then show the impact of high variance under the label *challenging*. To achieve high variance, we increase the original latency of the canary version. Instead of adding a constant amount of latency, we add latency that follows a normal distribution with $\mu = 100ms$ and $\sigma = 30$ (similar to [57]). That brings the canary's mean latency to 220 ms. We also generate load with a diurnal pattern and random spikes, and make the SLOs more challenging ($\ell_{lat} = 225ms$ and$\leq \ell_{err} = 0.03$). The high variance, unpredictable load, and tight SLOs lead to a low accuracy of 20% for Flagger and Argo Rollouts, while ITER8 can deliver an accuracy of 80%.

Summary. Finally, the overall results (Figure 3h) from all runs above reveal that canary releases orchestrated by ITER8 result in correct outcomes 93% of the time, with an F1-score[2] of 0.95. In contrast, Flagger and Argo Rollouts achieve an accuracy of 55% and 53%, respectively.

6.2 Reward maximization

We now evaluate ITER8's ability to maximize business reward during an experiment, which is the goal of A/B and A/B/n rollouts. Neither Flagger nor Argo Rollouts supports these scenarios. They do not even have the concept of reward. Thus, we chose a different benchmark compatible with relevant comparison baselines.

Benchmark application. The benchmark we use for assessing ITER8's reward maximization capability is an ML model serving system using TensorFlow Serving [54] to serve ML models. We use five representative and complex deep-learning models, namely VGG-16 [67], MobileNetV2 [62], EfficientNet-B0 [72], InceptionV3 [71], and ResNet50 [29] (see Table 2). A front-end HTTP server receives requests, which get routed to one of the models in the backend. An incoming HTTP request asks a model to classify an object from the CIFAR-100 ML dataset [46].

Comparison baselines & methodology. Not all Reinforcement Learning algorithms are suitable for experimentation with SLO criteria. Clipper [16] casts the problem of ML

[2]F1 is defined as the harmonic mean of precision and recall. Precision is the ratio of true positives to the number of all canary rollouts. The recall is the ratio of true positives to the number of all actual positives in the data set.

model selection as a multi-armed bandit problem and uses the Exp3 [7] algorithm to select the best among multiple competing ML models. We, therefore, compare Iter8's algorithms against Exp3, as well as with the most popular strategy used in traditional A/B and A/B/n tests, represented by an algorithm that splits the traffic uniformly—UNIF [39, 59]. We implement these two algorithms in Iter8 to conduct our comparative analysis.

Exp3. This algorithm associates a weight $s_i = 1$ for each of k models and then randomly selects a model with distribution (d) $p_i = s_i / \sum_{j=1}^{k} s_j$. It observes the reward (r_i) per variant, and updates weights according to $s_i := s_i * \exp(r_i/d(i) * \gamma/k)$. However, Exp3 merely considers reward; it does not take into account SLOs. Therefore, we modify Exp3 to support SLOs and, accordingly augment Exp3 with the concept of *effective reward*, which replaces the reward in the original heuristic outlined above. The value of the *effective reward* is 0 when the SLOs are not satisfied up until the current iteration, preventing Exp3 from exploiting versions not satisfying SLOs.

UNIF. UNIF [39, 59] is one of the most common strategies for A/B and A/B/n testing in practice. It splits user traffic equally across all competing versions to evaluate them using statistical hypothesis testing.

Methodology. We compare Iter8's algorithms against Exp3 and UNIF in the context of A/B/n rollout. In these experiments, we have 5 competing versions represented by the models listed in Table 2. The front-end HTTP server acting as a proxy for the 5 models in the backend relies on an oracle that knows the correct response to each request it receives from the CIFAR-100 data set. When a model returns the correct response, that model's reward is incremented; otherwise, it remains the same. The front-end service reports the reward for each model to the Prometheus telemetry database. Table 2 shows the models' mean reward and mean latency. The latency metric is reported by the Istio service mesh without any additional instrumentation.

Iter8 can use any application-specific metric available in a database for reward maximization or SLO validation. In the following ML model serving experiments (§6.2 and §6.3), we use model accuracy as the reward metric purely for demonstration purposes, and for comparison with exp3 [7], which used model loss in its evaluation that is similar in spirit. In the real world, we expect a business metric like conversion rate or user engagement to be used as the reward. Note that each of the 5 ML models is just like competing versions of cloud applications. Iter8 is not meant to solve the problem of hyperparameter tuning.

We define an SLO on mean latency ($\leq \ell_{lat} = 75ms$). As you can see in Table 2, some models do not meet that SLO. The goal of our A/B/n rollout experiments is as follows:

Table 2: Set of deep-learning models used in the single model selection experiments.

	model	reward	latency (ms)
v1	VGG-16	0.68	133
v2	ResNet50	0.77	82
v3	InceptionV3	0.79	90
v4	MobileNetV2	0.78	66
v5	EfficientNetB0	0.82	71

Among all versions satisfying SLOs, gradually roll out the winner version so that reward is maximized in the process. We generate a load of 50 requests/s combined with an additional diurnal load with random spikes. Each experiment results in more than 50000 requests. We run 3 types of experiments, each with different step durations: 10, 20, and 40 minutes. All experiments have the same duration but different numbers of steps. All data points shown in the next set of graphs correspond to the average of 5 runs. Unlike the experiments described in §6.1, here we do not set Iter8's *maxIncrement* parameter that limits the amount of traffic shift at each iteration.

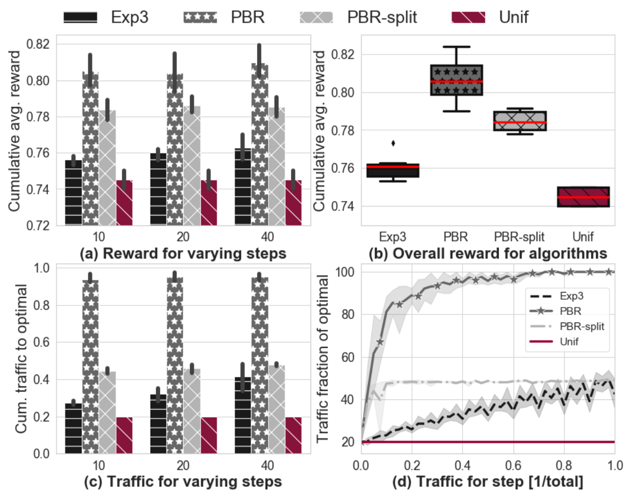


Figure 4: Reward maximization. Cumulative mean reward, and traffic towards optimal are shown for PBR, PBR-split, Exp3, and UNIF. Note that (a) and (b) are visualized zoomed-in to better demonstrate the difference between algorithms.

Quantitative results. Figure 4a shows the cumulative average reward obtained by Iter8's two algorithms (PBR and PBR-split), Exp3, and UNIF, and Figure 4b summarizes the overall average rewards from all these runs. Our overall results reveal that our PBR algorithm, whose purpose is to solve the reward maximization problem, outperforms Exp3 and UNIF in terms of cumulative average reward by 6% and 8%, respectively. Furthermore, PBR keeps the user traffic to the optimal version 93% of the time cumulatively, whereas Exp3 does so only 33% of the time, as shown in Figure 4c.

Critically, Figure 4d demonstrates that PBR converges quickly to the optimal version.

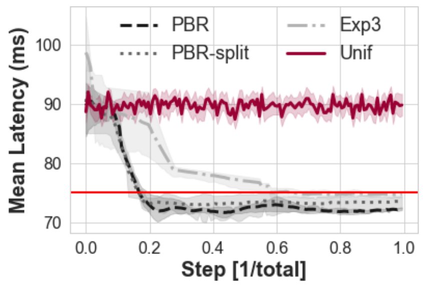


Figure 5: SLO violations. The overall mean latency from ML models are shown for PBR, PBR-split, Exp3, and UNIF. Upper limit on latency (SLO) is depicted by the red horizontal line.

Finally, Figure 5 shows the overall mean latency across all experiments for each of the algorithms. The horizontal red line represents the upper limit on mean latency as specified in the SLO. PBR not only yields the highest reward during the course of the experiments, but it also significantly outperforms Exp3 and UNIF in meeting the SLO. Note that Exp3 does not take SLOs into account in its original form (we extended it to consider SLOs), and neither does any other multi-armed bandit algorithm or online experimentation system available today.

6.3 Best-version identification

An experimenter might have the goal of confidently identifying the best version (while rolling it out) without worrying about maximizing a reward. Solving the problem of best-version identification enables shorter experimentation and, consequently, the release of the best version to production as quickly as possible. Using the same benchmark application and comparison baselines used in §6.2, we evaluate, comparatively, Iter8's ability to identify the best version as quickly as possible. The goal of Iter8's PBR-split algorithm is to solve the best-version identification problem.

Methodology. The experiment settings are the same we used in §6.2. We study how long (in number of requests) it takes each of the algorithms to identify the best version with varying levels of statistical confidence. Once the desired confidence level is reached, the experiment finishes.

Quantitative results. Figure 6 shows the number of requests needed by each algorithm to reach 3 confidence levels—90%, 95%, and 99%. Our PBR-split algorithm identifies the best version significantly faster than the others. It reduces the total number of requests by 56% compared to Exp3 and 59% compared to UNIF to reach a 99% confidence level. Moreover, it does so while observing SLOs (see Figure 5), which no other multi-armed bandit algorithm or online experimentation system supports today.

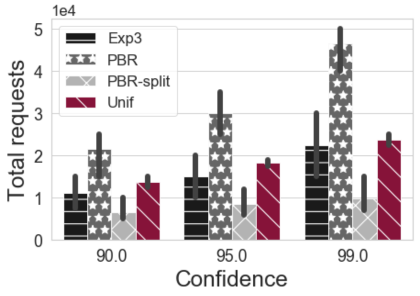


Figure 6: Best-version identification. The number of requests to reach various confidence levels are shown for PBR, PBR-split, Exp3, and UNIF.

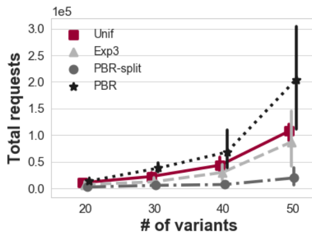
As we explained in §5, PBR-split's strategy prioritizes differentiating the best version from the second best and, by extension, the rest. Differently PBR, exploits the current best version in most of the steps at the expense of refining its knowledge of other competing versions.

6.4 Algorithmic analysis

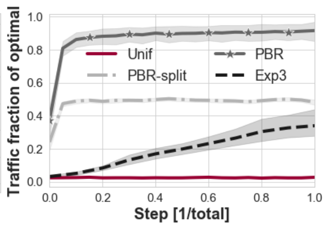
We ran all previous experiments in a public multi-tenant cloud on Kubernetes and Istio. In this section, we evaluate our algorithms using a Python simulation environment. These simulations enable us to investigate our algorithms' behavior with randomized KPI values and an increasing number of competing versions.

Methodology. We simulate KPIs for competing versions according to randomly generated distributions. The simulation process enables us to randomize KPIs, vary the number of competing versions, and evaluate the robustness of our algorithms more comprehensively.

We run 20 experiments for each given number of variants ([20, 30, 40, 50]) with 100 requests per step, and at each run, we randomize the KPI values. At the beginning of each experiment, the mean reward ($\mathbb{E}[X_0[p]]$) is sampled from the uniform distribution (U[0,1]), then the reward follows a Bernoulli distribution, i.e., $\Pr[X_0 = 1] = \mathbb{E}[X_0[p]]$. The mean latency ($\mathbb{E}[X_1[p]]$) is sampled from the uniform distribution (U[100,500]), and the latency of each simulated version follows a normal distribution (i.e., $X_1 \sim \mathcal{N}(\mu = \mathbb{E}[X_1[p]], \sigma^2 = 20)$). We specify an SLO on the mean latency as $\leq \ell_{lat} = 300ms$.



(a) Best-version identification



(b) Reward maximization

Figure 7: Comprhensive analysis of algorithms. Analysis of algorithms for a various numbers of competing versions with randomized KPIs.

Quantitative results. Figure 7 summarizes the overall results for reward maximization and best-version identification. Figure 7a shows the number of requests needed by each algorithm to reach a 95% level of confidence for a varying number of competing versions. PBR-split outperforms all other algorithms. In particular, when the number of competing versions increases, it becomes more challenging for the other algorithms to identify the best version. This is reflected by the substantial increase in the number of requests to reach the given level of confidence. In contrast, PBR-split quickly eliminates the underperforming versions with statistical confidence, resulting in quicker assessments. Thus, it scales better to a large number of versions.

Figure 7b shows the traffic split to the optimal version in the reward-maximization experiments. We observe that, even with a large number of versions and randomized KPIs, PBR significantly outperforms the alternatives, converging to the optimal version quickly, thereby maximizing the traffic to it and the reward.

6.5 Overhead

Iter8-analytics uses sophisticated statistical methods when deciding how to split user traffic across competing versions. This section evaluates if the overhead of running these statistical methods in real-time can impact their practicality. We perform these measurements on a machine with 4 CPUs (2.8 GHz) and 16 GB of memory by varying the number of versions in an experiment. Taking the average of 100 runs, we observe that the mean response time of the analytics service is 32ms, 40ms, 192ms, and 219ms for 3, 5, 20, and 50 competing versions, respectively. Given that experiments and their step duration are usually in the order of minutes and hours, sub-second delays are negligible. Further, Iter8-analytics and controller resource usage (which is minimal) is shared across experiments and users within a cluster.

7 DISCUSSION

Iter8 vs. resource allocation modules. Iter8 treats experimentation, auto-scaling, and hardware configuration as related but distinct problems. Iter8 experiments are declaratively specified, and experiment specifications co-exist with other specifications such as Horizontal Pod Autoscaler ([32]) configurations in Kubernetes or auto-scaling annotations in Knative ([43]). In our experiments, we provided the same auto-scaling and hardware configuration for each version, which ensures fairness. In particular, if the candidate version is found to be performant, and starts receiving a higher share of traffic, it would auto-scale in the same manner as the baseline version. This is the recommended practice for experimentation with Iter8.

Experimentation cost. Users can set up a maximum experiment duration which Iter8 will honor. If a winner cannot be found within this duration, Iter8 will end the experiment and provide the details of its statistical analysis to help users understand any trends discovered by it (§4.3). Users can also configure experiments so that a candidate version is rolled back immediately if it seriously misbehaves (e.g., due to a major performance regression) [37]. Another safeguard Iter8 supports is traffic segmentation, where a new version can be gradually rolled out only to a selected segment of the end-users [37].

8 CONCLUSION

In this paper, we propose Iter8, an online experimentation system for cloud applications. Our system embodies our novel mathematical formulation built on online Bayesian learning and our new multi-armed bandit algorithms (PBR and PBR-split) to enable online experimentation tailored for the cloud, considering both SLOs and business concerns, unlike existing solutions. Through extensive experiments in a public cloud, we compare Iter8 against popular alternatives from the cloud-native computing community, as well as state-of-the-art algorithms. We demonstrate that Iter8 is not only more general, but it also outperforms the existing alternatives.

Acknowledgements. We thank our shepherd, Zhihao Jia, and the anonymous reviewers for their invaluable feedback. This research is partially supported by IBM Thomas J. Watson Research Center.

REFERENCES

[1] Deepak Agarwal. 2013. Computational Advertising: The Linkedin Way. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management* (San Francisco, California, USA) *(CIKM '13)*. Association for Computing Machinery, New York, NY, USA, 1585–1586.

[2] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. 2014. LASER: A Scalable Response Prediction Platform for

Online Advertising. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining* (New York, New York, USA) *(WSDM '14)*. Association for Computing Machinery, New York, NY, USA, 173–182.

[3] Shipra Agrawal and Navin Goyal. 2017. Near-Optimal Regret Bounds for Thompson Sampling. *J. ACM* 64, 5, Article 30 (Sept. 2017), 24 pages. https://doi.org/10.1145/3088510

[4] Akamai. 2017. Akamai Online Retail Performance Report: Milliseconds Are Critical. https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp.

[5] Xavier Amatriain and Deepak Agarwal. 2016. Tutorial: Lessons Learned from Building Real-Life Recommender Systems. In *Proceedings of the 10th ACM Conference on Recommender Systems* (Boston, Massachusetts, USA) *(RecSys '16)*. Association for Computing Machinery, New York, NY, USA, 433. https://doi.org/10.1145/2959100.2959194

[6] Argo Rollouts. 2021. https://argoproj.github.io/argo-rollouts/.

[7] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. 2003. The Nonstochastic Multiarmed Bandit Problem. *SIAM J. Comput.* 32, 1 (Jan. 2003), 48–77. https://doi.org/10.1137/S0097539701398375

[8] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A Software Architect's Perspective* (1st ed.). Addison-Wesley Professional.

[9] Mark A. Beaumont, Wenyang Zhang, and David J. Balding. 2002. Approximate Bayesian Computation in Population Genetics. *Genetics* 162, 4 (2002), 2025–2035.

[10] Andrew Begel and Thomas Zimmermann. 2014. Analyze This! 145 Questions for Data Scientists in Software Engineering. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/2568225.2568233

[11] Lucas Bernardi, Themistoklis Mavridis, and Pablo Estevez. 2019. 150 Successful Machine Learning Models: 6 Lessons Learned at Booking.Com. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery; Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 1743–1751. https://doi.org/10.1145/3292500.3330744

[12] Jürgen Cito, Philipp Leitner, Thomas Fritz, and Harald C. Gall. 2015. 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* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 393–403. https://doi.org/10.1145/2786805.2786826

[13] Cloud Native Computing Foundation (CNCF). 2020. CNCF Survey. https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf.

[14] Cloud Native Computing Foundation (CNCF). 2021. CNCF. https://www.cncf.io.

[15] Controller Pattern, Kubernetes. 2021. https://kubernetes.io/docs/concepts/architecture/controller/.

[16] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 613–627.

[17] Custom Resources, Kubernetes. 2021. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

[18] Morris H. DeGroot and Mark J. Schervish. 2002. *Probability and Statistics* (3 ed.). Addison-Wesley.

[19] Yoav Einav. 2019. Amazon found every 100ms of latency cost them 1% in sales. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/.

[20] D. Ernst, A. Becker, and S. Tai. 2019. Rapid Canary Assessment Through Proxying and Two-Stage Load Balancing. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 116–122.

[21] Aleksander Fabijan, Pavel Dmitriev, Helena Holmström Olsson, and Jan Bosch. 2017. The Evolution of Continuous Experimentation in Software Product Development: From Data to a Data-Driven Organization at Scale. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 770–780. https://doi.org/10.1109/ICSE.2017.76

[22] Firebase. 2021. https://firebase.google.com.

[23] Flagger. 2021. https://docs.flagger.app.

[24] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 19–33. https://doi.org/10.1145/3297858.3304004

[25] Aditya Gopalan, Shie Mannor, and Yishay Mansour. 2014. Thompson Sampling for Complex Online Problems. In *Proceedings of the 31st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 32)*, Eric P. Xing and Tony Jebara (Eds.). PMLR, Bejing, China, 100–108.

[26] Thore Graepel, Joaquin Quiñonero Candela, Thomas Borchert, and Ralf Herbrich. 2010. Web-Scale Bayesian Click-through Rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine. In *Proceedings of the 27th International Conference on International Conference on Machine Learning* (Haifa, Israel) *(ICML'10)*. Omnipress, Madison, WI, USA, 13–20.

[27] M. Grambow, F. Lehmann, and D. Bermbach. 2019. Continuous Benchmarking: Using System Benchmarking in Build Pipelines. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. 241–246.

[28] Martin Grambow, Lukas Meusel, Erik Wittern, and David Bermbach. 2020. Benchmarking microservice performance: a pattern-based approach. *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (2020).

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]

[30] Hey-workload. 2016. https://github.com/rakyll/hey.

[31] Daniel N. Hill, Houssam Nassif, Yi Liu, Anand Iyer, and S.V.N. Vishwanathan. 2017. An Efficient Bandit Algorithm for Realtime Multivariate Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Halifax, NS, Canada) *(KDD '17)*. Association for Computing Machinery, New York, NY, USA, 1813–1821.

[32] Horizontal Pod Autoscaler. 2021. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[33] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (1st ed.). Addison-Wesley Professional.

[34] InfluxDB. 2013. https://www.influxdata.com/products/influxdb-overview/.

[35] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. 2011. On the Performance Variability of Production Cloud Services. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '11)*. IEEE Computer Society, Washington, DC, USA, 104–113.

[36] Istio. 2021. https://istio.io/docs/concepts/what-is-istio/.

[37] Iter8. 2020. https://iter8.tools/.

[38] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, and et al. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 34–50. https://doi.org/10.1145/3132747.3132749

[39] Julian Katz-Samuels and Clay Scott. 2018. Feasible Arm Identification *(Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, Stockholmsmässan, Stockholm Sweden, 2535–2543. http://proceedings.mlr.press/v80/katz-samuels18a.html

[40] KFServing. 2021. https://github.com/kubeflow/kfserving/.

[41] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2016. The Emerging Role of Data Scientists on Software Development Teams. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/2884781.2884783

[42] Knative. 2021. https://knative.dev/.

[43] KNative Autoscalers. 2021. https://knative.dev/docs/serving/autoscaling/autoscaler-types/.

[44] Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. 2013. Online Controlled Experiments at Large Scale. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Chicago, Illinois, USA) *(KDD '13)*. Association for Computing Machinery, New York, NY, USA, 1168–1176. https://doi.org/10.1145/2487575.2488217

[45] Ron Kohavi, Randal M. Henne, and Dan Sommerfield. 2007. 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* (San Jose, California, USA) *(KDD '07)*. Association for Computing Machinery, New York, NY, USA, 959–967. https://doi.org/10.1145/1281192.1281295

[46] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images.* Technical Report.

[47] Kubernetes. 2021. https://kubernetes.io/.

[48] Jaewon Lee, Changkyu Kim, Kun Lin, Liqun Cheng, Rama Govindaraju, and Jangwoo Kim. 2018. WSMeter: A Performance Evaluation Methodology for Google's Production Warehouse-Scale Computers. *SIGPLAN Not.* 53, 2 (March 2018), 549–563. https://doi.org/10.1145/3296957.3173196

[49] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. 2010. A Contextual-Bandit Approach to Personalized News Article Recommendation. In *Proceedings of the 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. Association for Computing Machinery, New York, NY, USA, 661–670.

[50] Hongqiang Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. CrystalNet: Faithfully Emulating Large Production Networks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[51] Benedict C. May, Nathan Korda, Anthony Lee, and David S. Leslie. 2012. Optimistic Bayesian Sampling in Contextual-bandit Problems. *J. Mach. Learn. Res.* 13 (June 2012), 2069–2106.

[52] Marissa Mayer. 2006. What Google Knows. Presentation delivered at the Web 2.0 Summit. Summary by Greg Linden is here: http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html.

[53] New Relic. 2021. https://newrelic.com.

[54] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. In *Workshop on ML Systems at NIPS 2017*.

[55] Optimizely. 2021. https://www.optimizely.com.

[56] Prometheus. 2021. https://prometheus.io/docs/introduction/overview/.

[57] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 805–825. https://www.usenix.org/conference/osdi20/presentation/qiu

[58] A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin. 2015. Synthesizing Continuous Deployment Practices Used in Software Development. In *2015 Agile Conference*. 1–10.

[59] Daniel Russo. 2016. Simple Bayesian Algorithms for Best Arm Identification. In *29th Annual Conference on Learning Theory (Proceedings of Machine Learning Research, Vol. 49)*, Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir (Eds.). PMLR, Columbia University, New York, New York, USA, 1417–1418. http://proceedings.mlr.press/v49/russo16.html

[60] Daniel Russo and Benjamin Van Roy. 2016. An Information-theoretic Analysis of Thompson Sampling. *J. Mach. Learn. Res.* 17, 1 (Jan. 2016), 2442–2471.

[61] Daniel J. Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. 2018. A Tutorial on Thompson Sampling. *Found. Trends Mach. Learn.* 11, 1 (July 2018), 1–96. https://doi.org/10.1561/2200000070

[62] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2019. MobileNetV2: Inverted Residuals and Linear Bottlenecks. arXiv:1801.04381 [cs.CV]

[63] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous Deployment at Facebook and OANDA. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16 Companion)*. ACM, New York, NY, USA, 21–30.

[64] 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* (Trento, Italy) *(Middleware '16)*. ACM, New York, NY, USA, Article 12, 14 pages.

[65] Mark J. Schervish. 1995. *Theory of Statistics*. Springer-Verlag.

[66] Mojtaba Shahin, Mansooreh Zahedi, Muhammad Ali Babar, and Liming Zhu. 2018. An empirical study of architecting for continuous delivery and deployment. *Empirical Software Engineering* (09 2018). https://doi.org/10.1007/s10664-018-9651-4

[67] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV]

[68] Spinnaker. 2021. https://www.spinnaker.io/.

[69] Split. 2021. https://www.split.io.

[70] Sysdig. 2021. https://sysdig.com.

[71] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2015. Rethinking the Inception Architecture for Computer Vision. arXiv:1512.00567 [cs.CV]

[72] Mingxing Tan and Quoc V. Le. 2020. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. arXiv:1905.11946 [cs.LG]

[73] 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* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/2815400.2815401

[74] 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 (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 418–422. https://doi.org/10.1145/2771783.2784770

[75] William R. Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3-4 (12 1933), 285–294. arXiv:http://oup.prod.sis.lan/biomet/article-pdf/25/3-4/285/513725/25-3-4-285.pdf

[76] Traffic Management, Istio. 2021. https://istio.io/latest/docs/concepts/traffic-management/.

[77] Traffic Splitting, Knative Serving. 2021. https://knative.dev/docs/serving/samples/traffic-splitting/.

[78] Brandon M. Turner and Trisha [Van Zandt]. 2012. A tutorial on approximate Bayesian computation. *Journal of Mathematical Psychology* 56, 2 (2012), 69 – 85.

[79] Vamp. 2021. https://vamp.io/.

[80] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. 2018. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2887384