ServiceLab: Preventing Tiny Performance Regressions at Hyperscale through Pre-Production Testing

Mike Chow, *Meta Platforms;* Yang Wang, *Meta Platforms and The Ohio State University;* William Wang, Ayichew Hailu, Rohan Bopardikar, Bin Zhang, Jialiang Qu, David Meisner, Santosh Sonawane, Yunqi Zhang, Rodrigo Paim, Mack Ward, Ivor Huang, Matt McNally, Daniel Hodges, Zoltan Farkas, Caner Gocmen, Elvis Huang, and Chunqiang Tang, *Meta Platforms*

https://www.usenix.org/conference/osdi24/presentation/chow

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

ServiceLab: Preventing Tiny Performance Regressions at Hyperscale through Pre-Production Testing

Mike Chow[1], Yang Wang[1 †], William Wang[1], Ayichew Hailu[1], Rohan Bopardikar[1], Bin Zhang[1], Jialiang Qu[1], David Meisner[1], Santosh Sonawane[1], Yunqi Zhang[1], Rodrigo Paim[1], Mack Ward[1], Ivor Huang[1], Matt McNally[1], Daniel Hodges[1], Zoltan Farkas[1], Caner Gocmen[1], Elvis Huang[1], and Chunqiang Tang[1]

[1] Meta Platforms
[†] The Ohio State University

Abstract

This paper presents ServiceLab, a large-scale performance testing platform developed at Meta. Currently, the diverse set of applications and ML models it tests consumes millions of machines in production, and each year it detects performance regressions that could otherwise lead to the wastage of millions of machines. A major challenge for ServiceLab is to detect small performance regressions, sometimes as tiny as 0.01%. These minor regressions matter due to our large fleet size and their potential to accumulate over time. For instance, the median regression detected by ServiceLab for our large serverless platform, running on more than half a million machines, is only 0.14%. Another challenge is running performance tests in our private cloud, which, like the public cloud, is a noisy environment that exhibits inherent performance variances even for machines of the same instance type. To address these challenges, we conduct a large-scale study with millions of performance experiments to identify machine factors, such as the kernel, CPU, and datacenter location, that introduce variance to test results. Moreover, we present statistical analysis methods to robustly identify small regressions. Finally, we share our seven years of operational experience in dealing with a diverse set of applications.

1 Introduction

In our hyperscale private cloud, tens of thousands of services run on millions of machines to serve billions of users, and engineers make thousands of code changes to these services daily. Performance or resource usage regressions caused by these changes may impact user experiences or even cause a site outage. Therefore, engineers critically rely on automated performance testing to catch regressions early.

Consider, for example, the frontend serverless platform called *FrontFaaS*. More than ten thousand engineers write code on this platform, with thousands of code changes committed daily and a new version released into production every three hours. If a code change causes even just a 0.01% regression in the platform's overall CPU usage, an alarm is raised. To our knowledge, strict thresholds of this level have not been studied before. We use this strict threshold because FrontFaaS consumes more than half a million machines and 0.01% would mean more than 50 machines. Moreover, if left undetected, many small regressions would accumulate over time. Each year, we catch regressions in FrontFaaS that amount to the capacity of more than one million machines.

This paper presents our performance testing platform called *ServiceLab*. It currently tests about one thousand diverse services and ML models, which, in aggregate, consume millions of machines in production. Although performance testing is widely used, there is no detailed report of its usage at hyperscale. Specifically, we have encountered several challenges that have not been studied before:

1. How to run tests on heterogeneous machines provided by the cloud while still ensuring comparable results?
2. How to detect regressions as small as 0.01%?
3. How to support hundreds of diverse services with one uniform testing platform?

We elaborate on each of these challenges below.

Use heterogeneous cloud machines. To detect small regressions, we must conduct numerous trials for an experiment and then apply statistical analysis. Since running these trials sequentially on one machine can take a long time (a trial takes over one hour on average), a natural solution is to run them in parallel on many machines. Ideally, these machines should be identical to reduce performance variance.

However, when a test workload is launched on a cloud, the cloud chooses machines to run the workload and even machines of the same instance type exhibit varying performance [47], due to differences in SSD wearing, memory chips from different vendors, and varying frequencies of CPU's uncore components like memory controller, etc. This phenomenon not only exists in public clouds but also in our private cloud that we use to run testing workloads. Note that

Contributions: Yang wrote the majority of the paper, followed by Mike and Chunqiang. Mike led the development of ServiceLab for multiple years, and other authors also made major contributions to its development.

our private cloud runs workloads on Linux containers instead of virtual machines (VMs) so there are no performance variances caused by VMs.

Although it is theoretically possible to reduce performance variance by maintaining our own dedicated pool of identical physical machines for testing, it is impractical for two main reasons: (1) testing workloads are spiky, and running them as on-demand workloads in the cloud is more cost-effective, and (2) maintaining a dedicated pool of tens of thousands of machines for testing requires an operations team that we cannot afford, which is exactly the problem that clouds aim to solve anyway.

Like in a public cloud, we can provision a batch of machines, keep a subset of "*nearly identical machines*" to run test workloads, and return the rest. The key question is how to select "*nearly identical machines*." Specifically, among the factors affecting a machine's performance, which are crucial for machine selection, and which can be ignored and addressed through statistical analysis?

To answer this question, we conducted a large-scale study with millions of performance experiments on various machines, using both microbenchmarks and real-world applications. We find that the performance variance on two machines is comparable to that on a single machine if the two machines share the same instance type, CPU architecture (e.g., Intel Cooper Lake), and kernel version, are located in the same datacenter region, and have CPU turbo disabled. An interesting observation is that the datacenter location matters, while other factors such as RAM vendor and RAM speed are less important. We will delve into this in §4.

Detect small regressions. For large services that consume tens of thousands of machines, we need to detect regressions as small as 0.01% while maintaining a low false positive rate. A high false positive rate not only wastes engineers' time in unnecessary debugging but also leads to engineers distrusting and ignoring the warnings even when they are correct. Our experience indicates that there is no one-size-fits-all statistical model that can accurately detect regressions for all services, due to the different outlier patterns of these services. To address this issue, we leverage multiple statistical models simultaneously and evaluate their false negatives and false positives on historical data to select the best model for each service. Although this ensemble approach may seem conceptually simple, we will discuss the intricacies of applying it at scale in highly noisy production environments.

Support diverse services. Our private cloud runs numerous services with intricate interdependencies, a complexity shared with other hyperscalers [30, 38, 48]. A single testing solution capable of covering all these services likely does not exist. Can we achieve the next best thing, i.e., having a single solution to cover the majority of code changes submitted by engineers? ServiceLab indeed accomplishes this. Currently, as a general-purpose testing platform, it covers more than half of the total code changes, surpassing the combined coverage of other specialized testing platforms.

ServiceLab takes the record-and-replay approach for testing, with three key distinctions. First, unlike past solutions that emphasize deterministic replay [8, 20, 24, 62], ServiceLab replays requests captured from a production system (PS) to a system under test (SUT) without expecting the SUT to exhibit the same behavior as the PS. In fact, due to testing changed code, it is anticipated that the SUT may make outgoing calls to downstream services that differ from those made by the PS. Therefore, ServiceLab does not replay the responses from downstream services to the SUT.

Second, ServiceLab allows the SUT to call downstream services running in production, provided there are no adverse side effects. Although users can set up a group of interdependent services in ServiceLab to create a self-contained testing environment without relying on the production environment, this approach is not consistently implemented due to practical reasons. For instance, making a per-test replica of certain massive datasets accessed by the SUT, such as the social graph for billions of users, is economically impractical.

In ServiceLab, the SUT can call downstream production services, and most of those calls do not incur side effects, as they are read-only or idempotent. If a SUT's call to a downstream service does cause side effects, ServiceLab provides a mock framework to assist the SUT in mitigating it. For example, instead of writing to a production database, the writes can be redirected to a test database.

Third, due to the complexity of hyperscale services, ServiceLab does not attempt to provide a simple but inflexible solution that requires no involvement from service owners, because such a solution would only work for a small fraction of services. Instead, ServiceLab allows and encourages the service owner's participation. For example, when testing a sharded stateful service, it is the service owner's responsibility to populate the necessary states before the test starts.

With the three key distinctions above, while ServiceLab's record-and-replay approach may necessitate occasional involvement from the service owner and does not extend to certain complex services, it effectively covers the majority of code changes submitted by engineers.

Contributions. We make the following contributions:

- We address the performance variance issue arising from running tests in the cloud. Specifically, we conducted millions of experiments to identify the factors that contribute most significantly to performance variance across machines. Such a large-scale study has not been reported before.
- We develop statistical analysis methods to robustly identify performance regression as small as 0.01%, even when tests do not use identical machines. This represents a significant refinement of existing methods, as no prior research has achieved this level of a low threshold.
- This is the first holistic report of a hyperscale testing plat-

form, including its design and our seven years of operational experience in dealing with a diverse set of applications.

2 ServiceLab from a User's Perspective

Before presenting the internals of ServiceLab, we first describe its usage from a user's perspective. ServiceLab can be used in different testing modes. The *efficiency mode* tests a code or configuration change's impact on key metrics such as latency or CPU/memory usage. The *capacity mode* tests a code or configuration change's impact on the maximum throughput that can be achieved, which affects the amount of capacity needed to run the service. The *hardware mode* compares the performance of different hardware running the same code. Below, we focus on the efficiency mode.

2.1 ServiceLab in the Development Workflow

Figure 1 depicts our development workflow, where ServiceLab is involved in the review-time test, commit test, deployment test, and config test. We elaborate on them below.

Meta uses the monorepo approach [12] to store code for all projects in one repository. When developing a new feature for an application, the developer clones the repository and makes local changes without affecting others. Once the code is ready, they submit the change, referred to as a *diff*, for peer review. Both functional and performance tests are automatically executed for the diff. The peer reviewer examines the code and test results, requesting changes as needed before approving the diff. Upon approval, the developer commits the diff, triggering post-commit tests.

On a set schedule, the continuous-deployment tool compiles a new executable for the application and creates a release candidate (RC). It conducts tests to compare the RC with the executable running in production. The RC is abandoned if a regression is identified. Otherwise, it is deployed into production in stages, and the application's health metrics, including performance metrics, are monitored continuously. If any health issue is detected, the deployment is reverted.

A common practice is to use a configuration parameter known as a *gate* to control access to the new code path. Initially, the gate is disabled so that the application continues to execute the old code path even after the new release is deployed. Then, the developer makes a remote configuration change to toggle the gate, enabling the application to execute the new code path. If any issues arise, the gate can be instantly disabled to revert back to the old code path without requiring a new code release.

2.2 Setting Up Tests with ServiceLab

To register a system-under-test (SUT) with ServiceLab, the application owner provides the following information:

- The selection criteria for code or configuration changes to trigger a test (note that it may not be necessary to run tests on every change);
- A container manifest that specifies the executable to test and how to set up Linux containers to run the executable;
- The metrics to be aggregated at the end of a test run, and the condition to fail the test;
- A traffic-recording configuration that instructs the RPC system how to sample production traffic for later replay;
- The rate at which the recorded traffic will be replayed during a test run.

ServiceLab supports both synthetic and record-and-replay traffic for testing, but primarily relies on the latter because it more accurately represents the production system. This approach records live production traffic and then replays it on a separate application instance in the testing environment, which may run modified code. It is the application owner's responsibility to ensure that a replay in the testing environment does not cause undesirable side effects on the production system. Moreover, a stateful application needs to set up its state properly so that it can handle the replayed traffic.

Once a SUT is registered at ServiceLab, it undergoes tests in four phases. The *build* phase compiles all required code into a package. The *allocation* phase acquires necessary machines from the cloud. The *running* phase initiates the target application on the allocated machines and replays the recorded workload. The *analysis* phase conducts statistical analysis on the results to draw a conclusion.

3 Applications Tested by ServiceLab

Currently, ServiceLab tests about one thousand diverse services and ML models, and their collective capacity consumption in production amounts to millions of machines. We describe several representative and large workloads below.

3.1 FrontFaaS Serverless Platform

FrontFaaS is one of the most complex software ecosystems in our private cloud. It is a serverless function-as-a-service (FaaS) platform that runs on more than half a million machines and has tens of thousands of developers making changes to its code base, with thousands of code commits every workday. ServiceLab tests FrontFaaS to detect CPU usage regression as small as 0.01%. It holistically tests different aspects of FrontFaaS: its PHP runtime called HHVM [27], the FaaS code written by tens of thousands of developers, and that code's impact on downstream services like databases.

Testing the language runtime. HHVM performs just-in-time (JIT) compilation for efficient execution. The HHVM team relies on ServiceLab to collect performance signals on compiler optimizations, monitoring metrics such as instructions per cycle, execution time, and cache misses. In addition to the core code written by the HHVM team, HHVM links with many libraries developed by other teams, any of which may cause regressions. HHVM tests compare the code running in production with the code in the *trunk* (i.e., the latest code

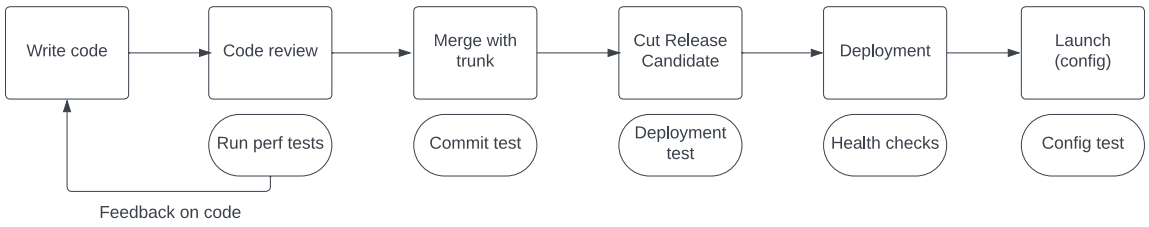

Figure 1: Code change workflow. In this figure, *tests* refer to both functional and performance tests, but this paper focuses on performance tests.

in the monorepo shared by all developers), enabling developers to catch regressions before a new release candidate is created. If a regression is detected, ServiceLab uses bisection to identify the root cause.

Selecting diffs to test. In addition to testing HHVM, ServiceLab also tests FrontFaaS' application-level FaaS code. FrontFaaS is the primary entry point for user-facing traffic for our products and runs thousands of unique application endpoints. With thousands of FaaS code changes (diffs) occurring every workday, it is not cost-effective to test every change. Moreover, since a change is unlikely to affect all thousands of application endpoints, it is unnecessary to replay the traffic for all those endpoints during a test.

A ServiceLab component called *DiffSuggester* selects which diffs to test based on a calculated *impact score* and also determines the traffic for which endpoints to replay during a test. DiffSuggester traverses the compiler's abstract syntax tree to identify functions modified by the diff. It calculates an impact score for each modified function by leveraging a profiling dataset of FrontFaaS' execution in production to estimate the global cost of the function, considering both its execution frequency and resource consumption per invocation. The diff's impact score is simply the sum of the impact scores for all impacted functions. If a diff's impact score is above a threshold, ServiceLab will run experiments for it. The threshold is statically chosen based on the number of machines available to run experiments and the distribution of diffs' impact scores. Moreover, DiffSuggester also uses the production profiling data to infer which application endpoints are impacted by the diff and selectively replay traffic for those endpoints with the right proportion.

Dealing with side effects. Because of the complexity of FrontFaaS, it is too costly to set up an entirely isolated testing environment for it. It invokes hundreds of downstream services, which recursively have their own dependencies. All these are hard to replicate in a testing environment and keep them faithful to the production environment. Moreover, given numerous concurrent tests, it is economically impractical to make a per-test copy of certain massive datasets accessed by FrontFaaS, such as the social graph for billions of users.

Therefore, ServiceLab allows a test instance of FrontFaaS to call downstream services running in production and carefully manages any adverse side effects. The non-functional side effects, such as test-induced load on downstream production systems, is not a concern because that test load is negligible compared to the production traffic from billions of users. The functional side effects, such as writing to a production database, is the main concern and is managed carefully.

By default, FrontFaaS' writes to databases, caches, and data warehouses are automatically dropped via a shim layer in the client libraries, while reads to these production systems are allowed. Unlike data stores where differences between read and write can be easily identified, for generic RPC calls, ServiceLab and the RPC system cannot easily infer whether an RPC method has undesirable side effects or not. Therefore, the RPC system drops calls to downstream production systems by default to ensure safety, while users can provide a list of specific RPC calls that are allowed to proceed. However, this method may prevent certain code paths from being executed and result in ServiceLab missing the opportunity to detect regressions on those code paths. If the owners of certain FrontFaaS endpoints really want to cover those code paths, it is their responsibility to modify the code's behavior so that it can run in ServiceLab to exercise those code paths without causing adverse side effects to production systems. We will delve into this in §5.3.

Testing performance impact on downstream services. A FrontFaaS diff may not cause regressions in the resource usage of FrontFaas itself but may regress in the load it imposes on downstream services. Specifically, the social graph database (*TAO* [11]) is one of the most important downstream services for FrontFaaS, and ServiceLab also detects increased reads to TAO. During a test, ServiceLab monitors the number of read requests that FrontFaaS issues to TAO when processing a replayed end-user request. Statistics are gathered at the granularity of each type of end-user request because the number of reads to TAO may vary widely depending on the type of the end-user request. Similar to reporting regressions on FrontFaaS' own metrics, ServiceLab also reports regressions in reads to TAO.

3.2 Sharded and Stateful Services

LASER is a low-latency key-value store that is frequently accessed by FrontFaaS on the critical path of serving user requests. LASER primarily serves as an indexing service for data in the data warehouse. Its index can be updated either by real-time stream processing that extracts data from data streams or by running daily MapReduce computation to build the index and then performing a bulk load into the key-value store. LASER is sharded and managed by a shared, central control plane similar to ShardManager [36] and Slicer [4], which dynamically assigns shards to different LASER servers.

Testing LASER faces several challenges. First, to bring up a LASER instance in the isolated test environment, we have to disconnect it from the central shard control plane and specifically instruct it to load certain shards, instead of relying on the control plane's dynamic shard assignment. Second, we allow LASER to perform read-only accesses to the data warehouse to set up its stateful index for testing. Finally, in production, requests routed to a specific LASER shard have an RPC header with the shard ID that matches with the shard; otherwise, the requests would be rejected. LASER uses record-and-replay for testing, which broadly samples requests for different shards. When requests are replayed, ServiceLab dynamically adds an RPC request header that matches with the shard ID of the LASER server under test.

LASER uses three major metrics for regression detection. These metrics and their regression thresholds are CPU usage (2%), anonymous memory usage (5%), and SSD storage usage (5%).

3.3 ML Prediction

MLPredictor is a shared ML deployment platform used by ML engineers to effortlessly deploy and manage thousands of ML models without the need for an understanding of the underlying infrastructure. This "serverless" approach conceals the operational complexities of large-scale distributed systems, which are often unfamiliar to ML engineers.

MLPredictor uses record-and-replay, along with ServiceLab's *capacity mode*, to test performance under varying load levels. ServiceLab incrementally increases the load level of the replayed traffic until MLPredictor breaches its service level objective (SLO), helping identify both the maximum throughput and potential capacity regressions. Initially, we recorded traffic for different models using uniform sampling, leading to an overwhelming number of samples from high-traffic models. Later, we switched to interval-based reservoir sampling [5,57], capping the number of samples for a popular model at a constant per time interval.

MLPredictor uses the maximum requests rate for regression detection, with a threshold of 5%.

3.4 Data Aggregation

DataAggregator is a CPU-intensive backend service that handles all news feed rankings. It is invoked by FrontFaaS upon a user request, and its role is to collect all relevant information about posts and analyze all the features (e.g., how many people have liked this post previously) to predict the posts' values to the user. New releases of DataAggregator are deployed to production multiple times throughout the day, and it primarily uses ServiceLab for release-time testing.

Instead of using record-and-replay, it uses a forker service to duplicate live production traffic and send it to the testing environment in ServiceLab. The forker sends the production system's responses back to users but drops the test instance's responses so that they will not affect users. DataAggregator prefers testing with shadow production traffic instead of recorded traffic because the setup is straightforward for them, and the existence of the forker even predates ServiceLab.

DataAggregator uses 68 key metrics for regression detection. Examples of the key metrics and their regression thresholds include container-level CPU usage (1.25%), process-level CPU usage (0.6%), and p99 memory usage (3%). Some metrics are related to the application logic, e.g., log error or warning counts (5%), no stories returned (2%), and latency to process all stories in the ranking service (30%).

3.5 XFinder

XFinder is a large service performing ads aggregation and ranking. Upon receiving a user request, it fans out requests to many leaf services, aggregates, and ranks the results before returning them to the user. XFinder uses record-and-replay, but to obtain accurate results, it requires near real-time traffic recorded from production within the past hour. Each week, it conducts over 3,000 and 1,000 experiments on code and configuration changes, respectively. To understand the impact of a change more precisely, instead of A/B tests, it runs 3-sided experiments: (1) the version currently running in production; (2) the latest version before this change; and (3) the new version with the change to be tested.

XFinder uses 65 key metrics for regression detection. These key metrics all use a regression threshold of 0.5%. The key metrics include total CPU time, log error or warning counts, count of ads returned, number of calls to downstream services, and failure rates of these calls.

3.6 Ranker

Ranker executes a graph of rules for ranking. A diverse set of application clients calls Ranker with different rules to provide ranking for their specific purposes, and these rules impact Ranker's performance. Ranker relies on record-and-replay to capture these rules. Requests from each application client are sampled on the client side and stored in the data warehouse. Each major client corresponds to a different shard of Ranker deployment, and these different shards run the same Ranker executable but serve different clients. Previously, Ranker created a mix of requests when replaying them for testing in ServiceLab. However, maintaining the correct ratio of requests in the mix became a burdensome process, and an incorrect

ratio would lead to missed regressions. As a result, Ranker now runs separate experiments to replay traffic from different major clients.

Ranker uses 30 key metrics for regression detection. The key metrics and thresholds include container-level CPU usage (9%), container-level memory usage (7%), CPU MIPS busy (5%), and application metrics such as different types of candidates fetched (20%).

3.7 Applications not Using ServiceLab

ServiceLab tests now cover more than half of the total code changes at Meta. The remaining applications choose other testing methodologies for various reasons as described below. A common theme among them is that setting up a service for testing in ServiceLab requires effort, and sometimes a simpler alternative exists.

First, Meta's continuous deployment tool, Conveyor [22], and its in-production detection tool, FBDetect, can catch performance regressions either during the staged deployment process or during steady-state execution in production. Despite the higher risk of catching issues in production, these tools work sufficiently well for some services, leading those services to skip pre-production testing in ServiceLab.

Second, some services have complex interdependencies, and services like Meta's cluster manager ecosystem [42, 54] even depend on the physical data center environment. These complex services have their own sophisticated ways of setting up their testing environments, which are often overly complicated to migrate to ServiceLab.

Third, some stateful services require a massive amount of data for effective testing. It is too slow to populate such data in newly allocated containers during each ServiceLab test run. Therefore, these services maintain their own dedicated and persistent test environments with prepopulated data, without relying on ServiceLab.

Fourth, some services do not consume significant capacity and do not have stringent performance requirements. As a result, thorough performance testing is not a priority for them. Their developers often prefer simpler ad-hoc testing methods, as opposed to the burden of setting up and maintaining their service setup in ServiceLab.

Finally, in a large organization with tens of thousands of developers, our experience indicates that achieving universal adoption of a technology is challenging unless it becomes a company priority, as demonstrated by the Push4Push program driving the universal adoption of the continuous deployment tool at Meta [22]. So far, ServiceLab has relied entirely on organic, bottom-up adoption without top-down push.

4 Taming Performance Variance

A key challenge in designing any testing platform is managing variance in testing data to separate signals from noises. To set the stage for the discussion, we first define some terminology. Assessing a code change's performance impact uses an *A/B test* to compare two *test runs*, one with the change and one without. A *trial* is a singular A/B test, and an *experiment* comprises multiple such trials. An *A/A* test compares two runs of the same code.

Performance differences may stem from (a) *accidental variance* caused by code's random factors such as the timing of lock contention; (b) *environment variance*, stemming from testing environment differences like CPU generation and kernel version; and (c) *true regression* in the code change. Our goal is to minimize the impact of accidental and environment variance to identify true regression.

To detect true regressions as small as 0.01%, we must aggressively reduce both accidental and environmental variance, as they could conceal small regressions. To reduce accidental variance, we collect a large amount of test data and then apply statistical analysis. To reduce environmental variance, we always acquire entire machines from our private cloud to run tests, avoiding the "noisy neighbor" problem. However, sequentially executing all test runs on one machine, while minimizing environmental variance, leads to prolonged test times and a slowdown in the iteration speed of software development.

One fundamental decision we have made is to run tests concurrently on different machines to expedite testing. Initially, the ServiceLab team operated its own dedicated machine pool and meticulously configured the machines to be nearly identical to reduce environment variance across machines. However, as the pool size expanded, maintaining it became uneconomical, leading us to switch to using our private cloud's shared machine pool. Moreover, the cloud allows ServiceLab to use temporarily reclaimed resources called "Elastic Servers", akin to Spot Instances in AWS, for testing. Since Elastic Servers can be revoked, our cloud employs predictive models to infer the availability of Elastic Servers and run tests correspondingly. When Elastic Servers are revoked unexpectedly, ServiceLab simply re-runs the interrupted tests.

When acquiring machines from the cloud, ServiceLab can specify a certain coarse-grained configuration such as CPU cores and memory, but cannot control other details, such as memory chip or kernel version. Note that the cloud automatically updates kernels at its own schedule to ensure security compliance. ServiceLab can provision a batch of machines, retain a subset of "*nearly identical machines*" to run test workloads, and return the rest. We do not require machines to be identical in every aspect as finding a sufficient number of such machines is difficult. Next, we discuss how to select "*nearly identical machines*" by using factors that impact a machine's performance most.

4.1 Machine Factors Impacting Performance

We analyze millions of test records to identify key factors impacting a machine's performance. Our analysis involves two large datasets. The Release to Production (RTP) dataset comprises 21.5 million records, each executing a CPU or memory

benchmark. The ServiceLab dataset contains 186K records, each testing a real production application. Each record in both datasets specifies the test result alongside the used hardware and software configuration. Leveraging both datasets is crucial as they complement each other. The RTP dataset provides diverse hardware results, though its benchmarks are less complex. Conversely, the ServiceLab dataset contains real application results, but the machines used are less diverse.

We use the ANOVA method [52] to identify factors that best explain the variance in the data. ANOVA is similar to linear regression but operates on categorical data. Its output, the coefficient of determination (R^2), represents the proportion of the variance in the dependent variable (performance metrics in our case) that is predictable from the independent variables (e.g., CPU generation or kernel version). Our goal is to find a minimal subset of key machine factors (independent variables) that can explain as much variance as using many factors. This allows us to use these key factors for machine selection. Otherwise, a large number of factors would make it hard to find matching machines due to overly aggressive filtering. To achieve our goal, we first use many factors to establish an approximate upper bound for R^2, and then explore different subsets of factors to approach the upper limit.

To set the stage for discussion, we will first describe how physical machines are classified with three levels of granularity. At the most coarse level, machines are classified into tens of *ServerTypes*. Examples of ServerTypes include single-CPU general-purpose machine, two-CPU general-purpose machine, GPU training machine, GPU inference machine, etc. The median granularity, known as *ServerSubType*, takes into account more hardware information, such as RAM size and CPU architecture (e.g., Intel Skylake, Cooper Lake, etc.). The finest granularity, referred to as *ServerModel*, includes the model names of all major components, such as CPU, RAM, NIC, disk, etc. Typically, users specify ServerType when requesting machines from our private cloud. While specifying ServerSubType is allowed, it is discouraged because it limits flexibility for the cloud to choose machines. Users are not allowed to specify specific ServerModel. Concretely, our private cloud uses O(10) *ServerTypes*, O(100) *ServerSubTypes*, and more than 10,000 *ServerModels*. They are equipped with O(100) CPU models, O(100) RAM models, O(1000) disk models, and O(100) NIC models.

To approximate the upper bound of R^2, we use ServerModel as one factor since it includes almost all hardware information and add non-hardware factors like the kernel release version. We first report our results on the RTP dataset. These factors can achieve an R^2 of 0.89 for the CPU benchmark and an R^2 of 0.97 for the memory benchmark. In the remainder of this section, we will focus on the CPU benchmark as the memory benchmark exhibits much less variance.

We explore various factor subsets to determine if a small combination can achieve an R^2 close to the upper limit. Using three factors—ServerType, CPU architecture, and kernel release—we attain an R^2 of 0.87 on the CPU benchmark, closely approaching the upper bound. In practice, we observe that the cloud can generally provide matching machines based on these three factors. Note that this subset is not the only viable option. As hardware factors are correlated, some factors can be replaced by others. Additionally, we find that certain factors, such as RAM speed and RAM vendor, have minimal impact, even in memory benchmarks.

Analyzing the ServiceLab dataset reveals two additional important factors: CPU turbo and the datacenter region where the test was executed. Their impact varies across applications, and adding these factors can increase R^2 by up to 0.23. In comparison, in the RTP dataset, adding these factors only increases R^2 by 0.01 for the CPU benchmark. The influence of CPU turbo, previously reported in research [40], manifests only in the ServiceLab dataset, not in the RTP dataset. This difference arises due to constant CPU activity in the RTP benchmarks. The datacenter region is significant for real applications tested in the ServiceLab dataset because many of them have external dependencies. For example, if the test instance of an application reads from a production database in the region, the test result would be affected by the database's performance, which tends to vary across regions. In contrast, the RTP benchmarks have no external dependencies.

While the key factors account for 87% of the variance, the remaining 13% is attributed to other smaller factors. For example, in machines with CPUs of the same model, the frequency of their uncore components, such as cache and memory controller, can vary, resulting in approximately a 2% performance difference across tests. However, these factors cannot be used for selecting machines from the cloud as they are not exposed by the cloud.

Our analysis further reveals that certain CPU models and kernel versions contribute significantly more variance than others. Like prior works [40], we use Coefficient of Variance (CoV), defined as the ratio of the standard deviation to the mean, to compute the variance of a set of values. Specifically, regarding CPU models, Intel Xeon E5-2680 v4 @ 2.40GHz has the highest CoV at 42%, while AMD EPYC 7D13 36-Core Processor has the lowest CoV at 5.6%, with an overall P50 at 19%. Regarding the kernel, version 5.6.13-0 has the highest CoV at 52%, and 5.2.0-240 has the lowest CoV at 9.5%, with the overall P50 at 36%. While investigating the root cause of CoV is beyond the scope of this paper, ServiceLab avoids using CPUs or kernel versions with high variance.

In summary, the strategy we use is to select similar machines with matching kernel versions, ServerTypes, CPU architecture, and datacenter regions, while disabling CPU turbo. To assess whether performance variance within the similar machines selected by our criteria is comparable to that for a single machine, we compare their CoVs. The comparison is conducted using the RTP dataset with turbo disabled. The CoVs for same-machine tests are 5.9% at P50 (50 percentile) and 28% at P99, while for similar-machine tests, they are

5.7% at P50 and 38% at P99. The P50 values are nearly identical, with a higher difference at P99. Overall, the difference is deemed acceptable, considering the advantage of running tests in parallel.

Applicability to public cloud. To implement our machine selection method in a public cloud, we recommend using bare metal instances, which are offered by all major cloud providers, rather than the more commonly used virtual machine instances. Although it is reported that lightweight hypervisors like AWS Nitro System can match the performance of bare metal machines [34], they may still introduce greater performance variability than bare metal machines. We have not validated our method in virtualized environments.

4.2 Statistical Methods

Despite using matching machines, experiments still exhibit variance. We conduct experiments multiple times and employ statistical methods to determine the level of regression with a confidence interval. In general, we observe that there is no one-size-fits-all model due to the diverse requirements and varying performance-data distribution of different services. Therefore, ServiceLab incorporates multiple models with a mechanism to learn the best model for each service based on historical data.

Recall that a *trial* is a singular A/B test, and an *experiment* comprises multiple trials. An *A/A* test compares two runs of the same code. A test may generate multiple data points. For example, a test may measure CPU utilization for an hour and generate a CPU-utilization data point per minute.

A model used by ServiceLab is a combination of a statistical test method and a data preprocessing method. ServiceLab uses the following statistical test methods:

- **Student's t-test** [17]. If an experiment only contains a single trial, we use the student's t-test to determine whether there is a significant difference between the means of the A side and the B side.
- **Permutation test** [6]. If an experiment includes multiple trials, for each trial, we first compute the difference in means between the A side and the B side. This step results in a vector of m values called $\vec{M}$, where m is the number of trials. Then we posit the null hypothesis H_0: $\mu_\Delta = 0$, where μ_Δ is the mean of $\vec{M}$. We apply a permutation test for this hypothesis as follows. We generate a large number of permuted samples from $\vec{M}$ and calculate the mean for each. Then we derive the p-value from the proportion of permuted sample means that are as extreme as or more extreme than the observed mean of $\vec{M}$.
- **Confidence interval test**. The above tests infer the distribution of the data from the experimental data. Since we can only run a limited number of trials within an experiment and some tests may incur outliers, such inference may not be accurate. The confidence interval test builds the data distribution from historical data. Specifically, it leverages A/A tests from the past two weeks to build the distribution of $mean(A') - mean(A)$, and further computes the confidence interval given the p-value, i.e., the probability of the observed difference of means being smaller than the confidence interval is larger than $1-p$. Then, for an experiment, it can test whether the B side follows the same distribution as the A side by determining whether $mean(B) - mean(A)$ is smaller than the confidence interval.

ServiceLab uses the following data preprocessing methods:

- **Square root transformations.** An important preprocessing step involves square root transformations. This is motivated by recognizing significant heterogeneity in the cost of requests, with certain requests disproportionately impacting mean metric values. Such disparities are exacerbated across multiple trials, leading to skewed aggregations. The square root transformation mitigates this, ensuring a more uniform contribution from each request to the trial's mean metric value. This adjustment has been empirically validated to enhance detection accuracy, especially for high-demand services.
- **Outlier detection.** We use conventional outlier mitigation methods, such as winsorization [18], which are particularly effective in moderating the elevated variance observed when services operate under strenuous conditions. Specifically, we either delete data points that are above a certain percentile (called outlier-elim) or cap those data points at the percentile value (called outlier-cap).

Out of all possible combinations of statistical test methods and data preprocessing methods, currently ServiceLab uses seven combinations: t-test-none, t-test-sqrt, t-test-outlier-elim, t-test-outlier-cap, permutation-test-none, permutation-test-sqrt, and confidence-interval-none, as well as some service-specific models.

ServiceLab uses an adaptive method to determine the best model for each <*service, metric*> combination. It conducts periodic A/A experiments and artificial A/B experiments (i.e., A/A experiments with injected regression on one side) to generate a "ground truth." Then, ServiceLab tests each model on the results of these experiments to obtain the model's false positive rate (from the A/A experiments), the false negative rate (from the artificial A/B experiments), and the detectability [10] (from the A/A experiments). ServiceLab then selects the model with the highest score, which is a linear combination of the false positive rate, false negative rate, and detectability, under the constraint that its false positive rate is below a threshold. ServiceLab runs this model selection algorithm periodically to adapt to changes in existing services and accommodate new services and metrics.

In our production, 51% of the services have adopted the confidence-interval-none model, 21% have adopted the t-test-sqrt model, 21% have adopted the adaptive method, 5% have adopted the t-test-none model, and 1% have adopted the

permutation-test-none model. Not all services use the adaptive method, either because they find a fixed model always works well or because they have not tried the recently introduced adaptive method.

The breakdown of the models chosen by the adaptive method is as follows: confidence-interval-none (49%), t-test-none (19%), t-test-outlier-cap (10%), t-test-outlier-elim (9%), permutation-test-none model (5%), t-test-sqrt (4%), and permutation-test-sqrt (3%). Over a 90-day period, for services using the adaptive method, more than 50% of them have changed models at least once, and more than 10% of them have changed models at least four times. This indicates that the best model for a service can change as the service's code and characteristics evolve.

Next, we discuss our journey to arrive at the current set of models. Initially, ServiceLab only supported single-trial experiments and thus used the student t-test with outliers handled by winsorization. When working with services requiring multiple trials, we found that the t-test did not work well. Outliers in both trials and requests within a trial affected the experiment results, showing up as either false positives or missed regressions due to excluding outliers. Consequently, for multiple-trial experiments, we added the permutation-test-none and permutation-test-sqrt models. The confidence-interval test was added to handle noisy metrics or ones that are not continuous like CPU or memory. We found that for these metrics, looking at the historical data to find the regression threshold would work better than dealing with a t-distribution (as the t-test and other variants do). Finally, motivated by the observation that different metrics follow different distribution patterns and such patterns may change over time, we added the adaptive method to help users find the best model.

Finally, we describe two optimizations that improve the accuracy of the statistical methods.

Test warm up time. Identifying and excluding initial warm-up periods in service operations is crucial for isolating steady-state performance metrics. We employ an algorithm using exponential moving averages to determine the point at which a time series reaches approximate stationarity. Observations before this point are discarded. As the duration of the warm-up phase depends on the test environment, this determination is made on an individual trial basis, ensuring that only matured performance data undergoes further analysis.

Periodic A/A experiments. ServiceLab conducts periodic A/A tests, and aggregates the results into a user dashboard, enabling users to monitor the false positive rate and detectability. This dashboard aids users in modifying workload settings to enhance the statistical signal of their experiment. For instance, users can adjust parameters such as increasing the number of trials, extending experiment duration, removing noisy metrics, or changing the aggregation method.

5 ServiceLab Design

This section presents the design of ServiceLab, utilizing the architecture diagram shown in Figure 2 in our discussion.

5.1 Experiment Lifecycle

During an experiment's lifecycle, it transitions through several phases: *queued→build→allocation→running→analysis*. An experiment begins when a user or an automation tool submits a request via the Windtunnel API, which enqueues the request into a *DurableQ* (durable queue) and creates an entry in the Windtunnel DB to represent the experiment, setting its phase as *queued*. The phase transition of an experiment is managed by a *processor*, and multiple processors can work independently to manage different experiments. When a processor determines it can take on additional work, it polls the DurableQ to claim a queued experiment and locks the corresponding Windtunnel DB entry to prevent other processors from performing duplicate work.

After some input validation and preprocessing, the processor transitions the experiment to the *build* phase, where the experiment's executables are created. The processor does not compile the executables directly but instead sends a request to a separate build service, which acts as a caching layer to prevent duplicate builds.

Once all executables are built, the experiment enters the *allocation* phase. Each team is configured with a certain testing-machine quota that they are allowed to use. The processor tracks the already used portion of the quota and determines when to allocate machines for experiments, enforcing priority and fairness. An experiment may need to run multiple jobs, such as one for the A side of the A/B test and another for the B side. As all jobs of an experiment must be allocated from the same datacenter region to minimize variance (§4.1), the processor decides from which region to allocate the jobs based on the remaining quotas in different regions. Additionally, the processor filters machines based on ServerType, CPU architecture, and kernel version to minimize variance across the selected machines (§4.1).

In addition to allocating the system-under-test (SUT) jobs, the processor also allocates a traffic-replay job and a test-harness job. The test-harness job drives the experiment and monitors the test's status. Depending on the experiment's purpose, different test-harness jobs can be used. For example, to measure the maximum throughput that the SUT can sustain, the *capacity* test harness can gradually increase the test throughput until the SUT violates its SLOs, such as response time, error rate, or CPU utilization exceeding a threshold.

Once all the necessary jobs are allocated, the experiment enters the *running* phase. If the experiment is testing a configuration change, the corresponding configuration canary [13,15,53] is set up correctly on the test machines. Subsequently, the traffic-replay job loads the previously recorded requests that will be replayed during the experiment. Finally, the processor instructs the test harness to start the test. Through-

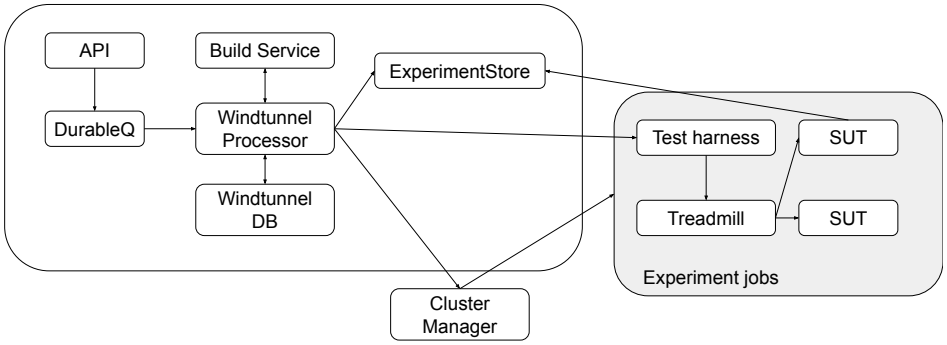

Figure 2: ServiceLab architecture. *Windtunnel* is the orchestration engine and *Treadmill* replays traffic and runs tests.

out the experiment, the test harness monitors the health checks of all jobs and fails the experiment if any job fails its health check. Meanwhile, the SUT exports performance metrics to monitoring databases during the experiment.

After the test finishes, the processor deallocates all jobs and transitions the experiment to the *analysis* phase. Aggregation and statistical analysis of performance metrics are performed by a service called *Experiment Store* (ES). The results are written to the Windtunnel DB, which users can view from a UI. These results can also trigger certain actions, such as blocking a service from being released into production.

5.2 Traffic Record-and-Replay

At Meta, all services use the Thrift [49] RPC protocol, which is leveraged by ServiceLab to record production traffic transparently. The user specifies the Linux containers where RPC traffic should be recorded. Upon receiving a request on these containers, the Thrift server's recording module flips a coin to decide whether to sample the request. To minimize the impact on RPC latency, the recorded data is written asynchronously. The user configures the request sampling rate, and by default, requests are sampled uniformly. For a service with a highly variable request rate, reservoir sampling [5, 57] ensures that sampled requests are evenly spread over time, with at most K samples during each T time interval. In practice, the median sampling rate is 0.03%, and above that, 22% of services set sampling rate to 1%.

By default, ServiceLab assumes that RPC requests are independent—meaning the execution of one request does not depend on the execution of the previous request. However, this assumption may not hold for some services. In these cases, the service can record all requests, and then during replay, all recorded requests are replayed in order. For some services, request dependencies are encoded at the RPC layer and can thus be recorded accurately and transparently.

For sharded services, since sharding is done at the application layer and is invisible to the RPC layer, ServiceLab relies on the service owner to specify the set of Linux containers to capture requests for all shards.

ServiceLab leverages the open-source load testing platform Treadmill [61] to replay requests. Treadmill employs an open-control loop to send requests at a fixed rate. We have implemented several modifications to Treadmill, extending it to load recorded requests from a datastore and replay Thrift RPC requests. In support of A/B experiments, we further enhanced Treadmill to ensure consistent pacing and request rates for both sides. A single Treadmill instance loads an identical set of requests to be dispatched to both SUTs, synchronizing the sending of requests to ensure simultaneous receipt on both sides. For the capacity mode, a control loop in the test harness monitors the SUT and instructs Treadmill to dynamically adjust the request rate.

Additionally, services may have a warm-up period during which performance measurements should not be taken until the service reaches a steady-state behavior. ServiceLab can be configured to enable a *warm-up phase*, allowing a lower request rate to be set during this phase to gradually increase the load on the service. The service exports a counter to indicate whether it has reached warm-up. Treadmill waits for both sides to be warmed up before synchronizing and sending requests once again for steady-state performance measurement. For example, HHVM employs a JIT compiler, and steady-state performance measurements should start after JIT compilation is sufficiently warmed up.

Finally, services with high variability in request processing time due to diverse request types are harder to handle. FrontFaaS is one such example. We can reduce variability by testing only with request types relevant to a specific code change, as opposed to all request types.

5.3 Handling Service Dependencies

Meta products are built out of tens of thousands of services with intricate interdependencies, akin to those documented in prior research [30, 38, 48]. For example, FrontFaaS invokes hundreds of downstream services. Consequently, testing a service in isolation is challenging due to these interdependencies. ServiceLab tackles this issue through various approaches.

First, users can set up a group of interdependent services together in ServiceLab, creating a self-contained testing environment. While theoretically possible, this approach is not consistently implemented in practice due to various reasons. For instance, replicating the massive datasets accessed by

services, like the social graph for billions of users, is often economically impractical.

Second, ServiceLab allows a system under test (SUT) to invoke certain services in the production environment, provided there are no adverse side effects. In ServiceLab, most calls to downstream production services do not incur side effects because they are read-only or idempotent. Moreover, the testing load imposed on these downstream services is often negligible compared to their ample capacity to serve billions of users. However, as these downstream services often exhibit performance variance across datacenter regions, meaningful comparisons can only be drawn from tests conducted in the same region (§4.1).

Third, for a SUT that potentially can cause side effects on downstream production services, ServiceLab requires the service owner to modify the SUT's behavior to prevent those side effects. For example, the SUT may use a mock interface of a database so that it writes data to a test database instead of the production database. Moreover, to prevent a SUT from accidentally accessing a production service, the RPC layer can be instructed to block all traffic to production services except those on an allowed list. Mocking or blocking traffic can result in certain code paths not being executed, potentially causing false negatives in testing results. However, § 6.1 shows that the false negative rate of ServiceLab is acceptable.

Fourth, the business and performance metrics logged by the SUT are kept separately from those generated by its counterpart in production. This ensures that the analytics for these metrics do not interfere with each other.

In summary, ServiceLab provides tools to assist service owners in managing service dependencies during test environment setup but does not offer complete isolation out of the box. As a result, some complex services (e.g., MySQL) are not tested in ServiceLab. They either use a specialized test environment or conduct canary tests directly in production by deploying new code to some instances of the production service and comparing those instances with the rest. Despite its limitations, ServiceLab is successful as a general-purpose testing platform, covering more than half of the total code changes by all services and surpassing the combined coverage of all other specialized testing platforms.

6 Production Experience

During its steady state, ServiceLab constantly leverages tens of thousands of machines to test hundreds of services and hundreds of ML models. We use production data to answer the following questions:

1. What are the statistics for different use cases (e.g., regression thresholds, number of trials, etc.)?
2. What are the false positive and false negative rates of ServiceLab?
3. How much regression did actually ServiceLab prevent?

6.1 Testing FrontFaaS

As FrontFaaS is our largest programming platform and has more code changes than other services, we report its statistics separately. ServiceLab has been running for FrontFaaS for over 5 years in production. It has a regression threshold as low as 0.01%, and by default, it runs 25 trials in each experiment. On average, developers made over 100,000 FrontFaaS changes per month. ServiceLab ran at least one experiment on 23% of those changes during that period. Leaving out 77% showcases the importance of ServiceLab's DiffSuggester in reducing the machine capacity needed for testing. For the code changes tested by ServiceLab, ServiceLab signaled performance regressions on 0.3% (5,560) of those changes.

ServiceLab assigns regression tickets to developers, and we calculate ServiceLab's accuracy based on the developers' actions in these tickets. We classify a signaled regression as a true positive if the developer fixed the issue or marked the issue as "*expected*," perhaps due to a new product feature requiring more resources. We classify it as a false positive if the developer identified it as such. If the developer did not provide a clear answer, we classify the regression as unknown. Among all signaled regressions, 57% (3,173) are true positives, 15% (823) are false positives, and 28% (1,564) are unknown. Assuming the unknowns have the same false positive rate as others, the overall false positive rate is about 21%.

Although the false positive rate of 21% may seem high initially, it actually signifies a significant success of ServiceLab because FrontFaaS uses a very low regression threshold, 0.01%. Out of all FrontFaaS diffs submitted by developers, only 0.014% experience a false positive flagged by ServiceLab, calculated as $23\% \times 0.3\% \times 21\% = 0.014\%$. Assuming a developer writes one diff per day, they will experience a false positive about once every 20 years! While promoting the adoption of ServiceLab, we learned that the per-developer experience significantly affects whether developers ignore the regression tickets assigned by ServiceLab. If a developer frequently receives false-positive tickets, they are likely to ignore them after repeated futile investigations. Conversely, if they receive a false-positive ticket only once every 20 years, they will likely always take ServiceLab regression tickets seriously and investigate them. The good developer experience even at a very low regression threshold of 0.01% demonstrates the robustness of ServiceLab's statistical methods.

Figure 3 shows the distribution of the level of regression of those true positive cases. The median value (p50) is 0.14%, p90 is 1.7%, and p99 is 38.7%. Summing them together, they account for 12284% of regression over five years. Since very large regressions are often caused by experimental purposes, if we only sum those causing less than 1% regression, they account for 545% of regression, which translates to over 2 million machines (i.e., 545% $\times$ the number of machines used by FrontFaaS). This shows that ignoring small regressions is not acceptable, as they will accumulate to a large number over time. That is why FrontFaaS uses a strict threshold.

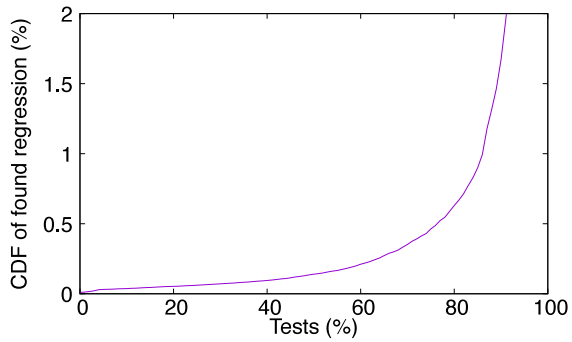

Figure 3: Cumulative distribution of regressions detected by ServiceLab for FrontFaaS.

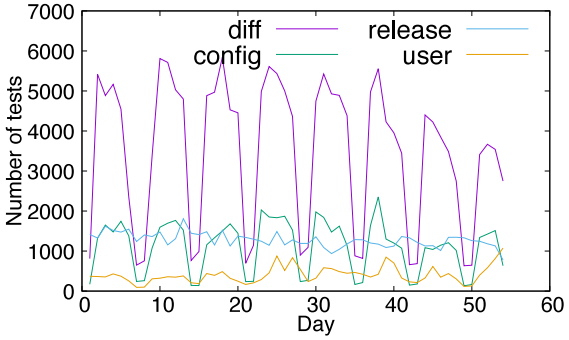

Figure 4: Number of experiments completed each day.

To approximate the false negative rate, we rely on reports from in-production monitoring of performance changes. FrontFaaS has a production monitoring system that examines the per-function CPU usage of functions and raises a signal if it detects a performance regression. It then attempts to triage the performance regression to a code change. Out of all changes, 0.02% (2038) of changes were found to cause regressions but have been missed by ServiceLab, leading to a false negative rate of 32%, calculated as $\frac{2038}{2038+5560\times(1-21\%)}$. However, this number should be viewed with caveats because 1) there are potential performance regressions that cannot be root-caused to their original changes, which are not included in this number, and 2) there is no guarantee that the production monitoring system is fully accurate.

In summary, despite FrontFaaS' low threshold of 0.01%, ServiceLab achieves a reasonable false positive rate and false negative rate, and helps us prevent a significant amount of regressions, which could accumulate over years.

6.2 Testing Other Services

While FrontFaaS is reported in its own category, in this section, we report the aggregate statistics for all non-FrontFaaS services in one category. Figure 4 shows the number of ServiceLab experiments completed each day for non-FrontFaaS services. The majority of completed experiments are run automatically as part of code changes, configuration configs, or service releases. ServiceLab supports a total of 483 distinct

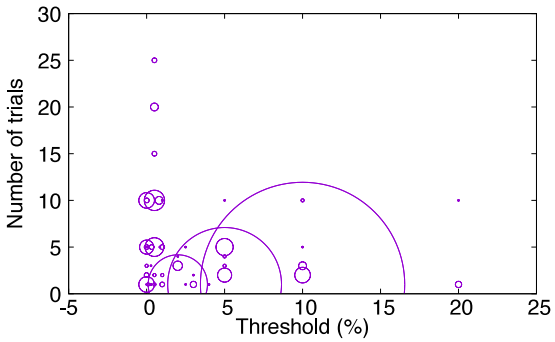

Figure 5: Thresholds and number of trials. The size of the circle represents the count of use cases with the same setting.

use cases, and their breakdown is shown below. Note that ServiceLab also tests hundreds of distinct ML models, which are counted as a single use case.

- 44% (N=211) of the use cases have experiments that automatically run on code diffs.
- 15% (N=74) run automatically on code commits.
- 21% (N=100) run automatically on configuration changes.
- 22% (N=107) run as part of their release process.

The distribution of the number of trials in experiments is as follows: p50=1, p90=10, p99=10, and p100=25. The distribution of the execution time of trials is as follows: p50= 2,820 seconds, p90= 4,200 seconds, p99=p100=259,200 seconds. Among the 483 use cases, 413 have defined a relative threshold on some metric; 5 have defined an absolute threshold on some metric; the remaining ones do not define any threshold. We focus on the 413 cases with a relative threshold in the following discussion.

Each use case may contain multiple metrics with different thresholds. Since the number of trials and trial duration are usually determined by the strictest threshold, we define the threshold of a use case as the smallest threshold among all its metrics. 23% of the use cases have a threshold smaller than 1%, while p50=5%, p90=10%, and p99=20%. This, once again, emphasizes the importance of using small thresholds.

Figure 5 plots the threshold and the number of trials used by different use cases. A circle in this figure represents the count of use cases using a specific setting. This figure shows that a large number of use cases use a relaxed threshold of 5% or 10% with only one trial, but a small number of use cases use a very small threshold with up to 25 trials. This small subset includes many of the largest services.

Services often run preliminary experiments with a large number of trials to determine how many trials are needed to achieve a certain confidence interval in regression detection. Specifically, they run multiple trials of A/A tests, compute the difference between each pair of A/A test (i.e., $\frac{A'-A}{A}$), and then determine the confidence interval (i.e., 95% within two standard deviations assuming normal distribution). Figure 6

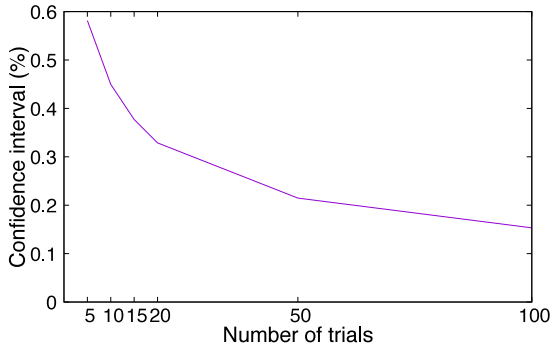

Figure 6: The number of trials required for detecting small regressions.

shows, for one service, how the confidence interval decreases with more trials. Users can then decide the number of trials according to their required confidence interval.

We examine how often ServiceLab signals a regression on code changes during the 54 day period shown in Figure 4. During this time, 15,058 code changes were tested and ServiceLab signaled on 2,742 (18.5%) of those code changes with at least one metric crossing its configured threshold. For non-FrontFaaS services, ServiceLab reports on a diverse set of metrics. Across these code changes, 2,714 different metrics were considered as significant. 80% of the signaled metrics had a threshold of less than 2%. Unlike the uniform FrontFaaS platform used by over ten thousand developers, for these 413 diverse use cases, there are no uniform tools and hence no clear marking about whether a reported regression is a true or false positive. In subsequent sections, we will present some examples with these use cases to understand their impact.

6.2.1 Example of True Positives

ServiceLab helps non-performance experts understand the performance implications of their code. For example, consider one case where ServiceLab successfully detected and prevented a CPU performance regression in XFinder (§3.5) before the change landed in production.

In the change, the developer inadvertently copied a large data structure when introducing a new function. The ServiceLab experiment that ran for this change detected a significant CPU regression of around 20%. ServiceLab flagged this change to both the developer and performance engineers working on XFinder. The developer was working on a product feature across multiple services, and was neither familiar with the XFinder codebase nor C++. After ServiceLab flagged the regression, the developer applied a fix by adding `const` when passing the parameter to the function, eliminating the memory copy of the data structure.

In another incident involving the Ranker service (§3.6), a change increased the service's memory usage by 50%. The change involved enabling a new ranking library that increased memory usage due to loading additional ranking configurations. The increase in memory was expected due to the additional functionality; however, the amount of increased memory was not. ServiceLab detected the memory regression before a release deployment. In this case, the developer who included the additional ranking library knew that there would be an added resource cost. However, ServiceLab helped the developer and service owners understand the resource cost of the regression before deployment. The developer reverted the change and found optimizations to minimize the use of the ranking library by excluding unused ranking configurations.

6.2.2 Example of False Positives

In another incident, a ranking service using ServiceLab occasionally experienced high rates of false positives due to a production issue with a downstream dependency. A production misconfiguration led to imbalanced load among the machines in the downstream service. During experiments, some of the SUTs would send requests to these overloaded instances of the downstream service. The queuing resulting from those overloaded downstream instances affected the performance measurement in ServiceLab, resulting in false positives. To remediate this issue, the production routing configuration that led to the imbalanced load was fixed. This remediated the load imbalance issue in production and also eliminated the false positives in ServiceLab.

6.2.3 Examples of False Negatives

False negatives are incidents where ServiceLab does not report a regression but a regression actually occurs. These cases are often reported by service owners. In one incident with XFinder (§3.5), a developer was implementing a new feature to read from an online classifier instead of an offline classifier. The change introduced a new function call making use of the new classifier to better classify the type of ads to return. The change resulted in an increase of 0.62% in the total capacity used by XFinder. ServiceLab failed to report a regression for this change since this regression only applied to a subset of request types, and those types were not represented in the set of requests replayed in the experiment. Those request types were newly added after the request trace was captured.

6.2.4 Summary

In our experience, the top reason for false positives is that another event, such as another test or deployment either in the SUT or in the downstream services, is happening concurrently with a ServiceLab test, which will disrupt the result of the ServiceLab test. The top reason for false negatives is that a newly introduced feature is not tested since the requests for replaying were recorded when this feature does not exist.

6.3 False Positive in A/A Experiments

As described in § 4.2, periodic A/A experiments provide an empirical measurement of whether a metric would be considered significant with the same experiment inputs. Periodic A/A experiments run every two hours and test for statistically significant differences without considering any signal-

ing thresholds. Over a two-month period, we examined 6,783 metrics from A/A experiments where signaling was enabled. Among these 6,783 metrics, the p50, p90, and p99 metric had a false positive rate of 0.6%, 40%, and 64%, respectively. This signifies the inherent variance in the services and the test environment. It also emphasizes the importance of our method of using results from A/A experiments to help select the best statistical model for each service (§4.2).

6.4 Key Takeaways

We have learned several key lessons from our experience of operating ServiceLab over seven years. Initially, the ServiceLab team maintained a dedicated pool of identical physical machines for testing to reduce performance variance. However, as ServiceLab adoption increased, we had to switch to using heterogeneous cloud machines. This change was driven by the high maintenance burden of a dedicated machine pool and the lower cost of running some tests using the cloud's elastic capacity.

Testing a wide range of services is a key design goal for ServiceLab. To achieve this, unlike traditional systems that aim for completely isolated and reproducible environments, ServiceLab allows Systems Under Test to call external dependent services that handle live production traffic. This approach significantly broadens the scope of services that ServiceLab can test, accommodating those with complex interdependencies that are too intricate or costly to replicate fully in a test environment. Moreover, ServiceLab is extensible and allows for developer customizations, recognizing that a one-size-fits-all approach would fall short in supporting diverse services. For example, while traffic record-and-replay simplifies test setup, some services face strict time constraints for replay, and others choose to use synthetic traffic.

7 Related Work

Performance Variance. Performance variance is a well-known issue for performance experiments and reproducibility, especially when the performance of two versions to compare is close. There are multiple lines of work in this direction: 1) some works mitigate the problems by re-designing systems [9, 16, 21, 25, 45, 51, 60], tuning configuration parameters [37, 39, 59, 64], or changing hardware [40, 58, 63]. 2) Some works try to detect machines that are significantly slower than others [19, 23, 28, 29, 43], so as to exclude such outlier machines from performance experiments. We also run routine performance tests to filter those outlier machines. 3) Some works propose statistical methods for performance comparison [26, 31, 40].

The closest work is the study by Maricq et al. on performance variance in CloudLab [40]. ServiceLab differs from the CloudLab study in several ways. First, the CloudLab study assumes repeated experiments are run on the same or identical machines, whereas ServiceLab identifies heterogeneous machines with comparable performance to run experiments in parallel. Second, the CloudLab study focuses on the number of experiments needed to achieve a certain confidence interval, whereas ServiceLab addresses the problem more holistically, using an ensemble of statistical models, A/A tests, and artificial A/B tests. Finally, the CloudLab study only runs microbenchmarks in a single-machine environment, whereas ServiceLab must be robust enough to work in real-world scenarios with full services and complex interdependencies.

Performance Testing. Synthetic benchmark [14, 41, 46, 50, 55] and record-and-replay [1–3] are two primary methods for performance evaluation. ServiceLab supports both but primarily uses record-and-replay due to its high fidelity in testing real applications.

Treadmill [61] and TailBench [32] overcome several common pitfalls of performance testing frameworks with synthetic traffic, allowing them to precisely measure at microsecond-scale. Lancet [33] incorporates online statistical tests to ensure the obtained measurements are statistically sound. Primorac et. al. leverage kernel-bypass networking and advanced NIC features to further improve the precision of microsecond-scale tail latency measurements [44].

Performance data from Google's gmail [7] shows that workloads change constantly, both QPS and response size. Hence we need to do real production traffic record and replay. Recent studies including Kraken [56] and WSMeter [35] directly utilize production traffic to carry out the performance tests, to address the limitation of synthetic benchmarking in how accurately they can reproduce the complex production environment. Similarly, deterministic record and replay are commonly leveraged to reduce the non-determinism to simplify multiprocessor software development and testing, which can be done at multiple levels (e.g., virtual machine-level [20], OS-level [8], and library-level [24]).

8 Conclusion

We have presented ServiceLab, which tests a large, diverse set of applications to catch small performance regressions. It selects similar but non-identical machines for testing and learns the best statistical model for each service. During seven years of production, ServiceLab has helped us prevent a significant amount of regression, which could accumulate over time if not detected promptly.

Acknowledgments

This paper presents over seven years of work by past and current members of several teams at Meta, including ServiceLab, HHVM, Ads, Release Engineering, and Instagram. In particular, we would like to thank those who have contributed to ServiceLab and this paper: Matt Meyers, Elena Mironova, Yun Jin, Harish Dattatraya Dixit, and Gautham Vunnam. We also thank all reviewers, especially our shepherd, Atul Adya, for their insightful comments.

References

[1] Azure Public Dataset. https://github.com/Azure/AzurePublicDataset.

[2] Google Cluster Workload Traces 2019. https://research.google/resources/datasets/google-cluster-workload-traces-2019/, 2019.

[3] Google Workload Traces 2022. https://research.google/resources/datasets/google-workload-traces-2022/, 2022.

[4] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: Auto-sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 739–753, 2016.

[5] Charu C Aggarwal. On Biased Reservoir Sampling in the presence of Stream Evolution. In *Proceedings of the 32nd international conference on Very large data bases*, pages 607–618, 2006.

[6] Marti J Anderson and John Robinson. Permutation tests for linear models. *Australian & New Zealand Journal of Statistics*, 43(1):75–88, 2001.

[7] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance Analysis of Cloud Applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, 2018.

[8] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D Gribble. Deterministic Process Groups in dOS. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[9] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail Latency Aware Caching – Dynamic Reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, Carlsbad, CA, 2018. USENIX Association.

[10] Howard S Bloom. Minimum Detectable Effects: A Simple Way to Report the Statistical Power of Experimental Designs. *Evaluation review*, 19(5):547–556, 1995.

[11] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, page 49–60, USA, 2013. USENIX Association.

[12] Nicolas Brousse. The Issue of Monorepo and Polyrepo In Large Enterprises. In *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*, pages 1–4, 2019.

[13] Amazon CloudWatch - Creating a canary. https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch_Synthetics_Canaries_Create.html.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[15] David Daly. Creating a Virtuous Cycle in Performance Testing at MongoDB. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 33–41, 2021.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[17] Wilfrid J Dixon and Frank J Massey Jr. Introduction to Statistical Analysis. 1957.

[18] Wilfrid J Dixon and Kareb K Yuen. Trimming and winsorization: A review. *Statistische Hefte*, 15(2-3):157–170, 1974.

[19] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[20] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[21] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[22] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, and Chunqiang Tang. Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta. In Roxana Geambasu and Ed Nightingale, editors, *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 325–342. USENIX Association, 2023.

[23] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 1–14, Oakland, CA, February 2018. USENIX Association.

[24] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M Frans Kaashoek, and Zheng Zhang. R2: An Application-Level Kernel for Record and Replay. In *OSDI*, volume 8, pages 193–208, 2008.

[25] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 168–183, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, USA, 1st edition, 2013.

[27] HHVM. https://hhvm.com.

[28] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and Enhancing In Situ System Observability for Failure Detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018. USENIX Association.

[29] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 150–155, New York, NY, USA, May 2017. ACM.

[30] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *Proceedings of the 2023 USENIX Annual Technical Conference*. USENIX, 2023.

[31] Raj Jain. *The Art Of Computer Systems Performance Analysis: Techniques For Experimental Measurement, Simulation, And Modeling*. john wiley & sons, 2008.

[32] Harshad Kasture and Daniel Sanchez. TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[33] Marios Kogias, Stephen Mallon, and Edouard Bugnion. Lancet: A self-correcting Latency Measuring Tool. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 881–896, Renton, WA, July 2019. USENIX Association.

[34] Matt Koop. Bare metal performance with the AWS Nitro System. https://aws.amazon.com/blogs/hpc/bare-metal-performance-with-the-aws-nitro-system/.

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

[36] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 553–569, 2021.

[37] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[38] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[39] Ashraf Mahgoub, Paul Wood, Alexander Medoff, Subrata Mitra, Folker Meyer, Somali Chaterji, and Saurabh Bagchi. SOPHIA: Online Reconfiguration of Clustered NoSQL Databases for Time-Varying Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 223–240, Renton, WA, July 2019. USENIX Association.

[40] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming Performance Variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 409–425, Carlsbad, CA, October 2018. USENIX Association.

[41] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. MLPerf Training Benchmark, 2020.

[42] Andrew Newell, Dimitrios Skarlatos, Jingyuan Fan, Pavan Kumar, Maxim Khutornenko, Mayank Pundir, Yirui Zhang, Mingjun Zhang, Yuanlai Liu, Linh Le, Brendon Daugherty, Apurva Samudra, Prashasti Baid, James Kneeland, Igor Kabiljo, Dmitry Shchukin, Andre Rodrigues, Scott Michelson, Ben Christensen, Kaushik Veeraraghavan, and Chunqiang Tang. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, 2021.

[43] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi. IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 47–62, Renton, WA, July 2019. USENIX Association.

[44] Mia Primorac, Edouard Bugnion, and Katerina Argyraki. How to Measure the Killer Microsecond. *SIGCOMM Comput. Commun. Rev.*, 47(5):61–66, oct 2017.

[45] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, Carlsbad, CA, 2018. USENIX Association.

[46] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. MLPerf Inference Benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 446–459. IEEE Press, 2020.

[47] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proceedings of the VLDB Endowment*, 3(1-2):460–471, 2010.

[48] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. A Cloud-Scale Characterization of Remote Procedure Calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 498–514, 2023.

[49] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. *Facebook white paper*, 5(8):127, 2007.

[50] Standard Performance Evaluation Corporation. https://www.spec.org/.

[51] Akshitha Sriraman and Thomas F. Wenisch. uTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, 2018. USENIX Association.

[52] Lars St and Svante Wold. Analysis of variance (ANOVA). *Chemometrics and intelligent laboratory systems*, 6(4):259–272, 1989.

[53] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343, 2015.

[54] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat,

Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 787–803. USENIX Association, 2020.

[55] Transaction Processing Performance Council. The TPC home page. http://www.tpc.org.

[56] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 635–651, 2016.

[57] Jeffrey S. Vitter. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, mar 1985.

[58] Guosai Wang, Lifei Zhang, and Wei Xu. What Can We Learn from Four Years of Data Center Hardware Failures? In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–36. IEEE, 2017.

[59] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. Understanding and Auto-Adjusting Performance-Sensitive Configurations. *SIGPLAN Not.*, 53(2):154–168, March 2018.

[60] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 15–28, Santa Clara, CA, February 2017. USENIX Association.

[61] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 456–468. IEEE Press, 2016.

[62] Wei Zheng, Ricardo Bianchini, G John Janakiraman, Jose Renato Santos, and Yoshio Turner. JustRunIt: Experiment-Based Management of Virtualized Data Centers. In *Proc. USENIX Annual technical conference*, volume 96, 2009.

[63] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. wPerf: Generic Off-CPU Analysis to Identify Bottleneck Waiting Events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 527–543, Carlsbad, CA, 2018. USENIX Association.

[64] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 338–350, New York, NY, USA, 2017. Association for Computing Machinery.