Detecting Tiny Performance Regressions at Hyperscale

YANG WANG, Meta Platforms Inc, Columbus, United States and The Ohio State University, Columbus, United States
MIKE CHOW, Meta Platforms Inc, New York, United States
DONG YOUNG YOON, Meta Platforms Inc, San Mateo, United States
ROHAN BOPARDIKAR, Meta Platforms Inc, Los Altos Hills, United States
SHERRY CHEN, Meta Platforms Inc, Sunnyvale, United States
ZOLTAN FARKAS, Meta Platforms Inc, New York, United States
CANER GOCMEN, Meta Platforms Inc, Foster City, United States
KAPIL GOENKA, Meta Platforms Inc, Dublin, United States
AYICHEW HAILU and DANIEL HODGES, Meta Platforms Inc, New York, United States
ELVIS HUANG, Meta Platforms Inc, Hoboken, United States
IVOR HUANG, Meta Platforms Inc, New York, United States
ZHIHUI HUANG, Meta Platforms Inc, Mountain View, United States
JUAN JONES, Meta Platforms Inc, Chicago, United States
OSMAN KOCAS and JOCELYN KONG, Meta Platforms Inc, New York, United States
ABHINAY KUKKADAPU, Meta Platforms Inc, San Diego, CA, United States
YANJUN LIN, Meta Platforms Inc, San Carlos, United States
MATT MCNALLY, Meta Platforms Inc, Natick, United States
DAVID MEISNER, Meta Platforms Inc, San Francisco, United States
RODRIGO PAIM, Meta Platforms Inc, Plainsboro, United States
DHRUV PATEL, Meta Platforms Inc, New York, United States
JIALIANG QU, Meta Platforms Inc, Jersey City, United States
SANTOSH SONAWANE, Meta Platforms Inc, Livingston, United States
WILLIAM WANG and MACK WARD, Meta Platforms Inc, New York, United States
JONATHAN WIEPERT, Meta Platforms Inc, Greenlawn, United States
MIAO YU, Meta Platforms Inc, New York, United States, Meta Platforms Inc, Mountain View, United States, and The Ohio State University, Columbus, United States
BIN ZHANG, Meta Platforms Inc, Ridgewood, United States
YUNQI ZHANG, Meta Platforms Inc, New York, United States
CHUNQIANG TANG, Meta Platforms Inc, San Jose, United States

Just Accepted

Authors' Contact Information: Yang Wang, Meta Platforms Inc, Columbus, OH, United States and The Ohio State University, Columbus, OH, United States; e-mail: yangwang@meta.com; Mike Chow, Meta Platforms Inc, New York, NY, United States; e-mail: mcchow@meta.com;

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 1557-7333/2025/12-ART
https://doi.org/10.1145/3785504

This paper presents Meta's performance testing and monitoring systems, which advance the state of the art in performance regression detection by catching regressions as small as 0.005%. These tiny regressions matter due to our large fleet size and their potential to accumulate over time. Detecting such tiny regressions, however, is challenging, due to various kinds of noises caused by heterogeneous machines, server failures, maintenance operations, load spikes, etc. To detect tiny regressions despite such noises, Meta has developed two complementary systems, ServiceLab and FBDetect.

ServiceLab is a *pre-production* testing platform, which conducts A/B experiments in an isolated environment by reserving servers from our private cloud. While this approach avoids most of the noise in production, heterogeneous machines are still a major challenge, since even machines of the same instance type from our private cloud may have subtle differences in various ways. To address this challenge, 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.

Despite the success of ServiceLab, we observe that pre-production testing has its inherent limitation: Due to the limited resource that can be used for testing purpose, pre-production testing cannot fully reproduce the scale and complexity of the production system. As a result, some buggy code or config changes slip through. To capture such buggy changes, FBDetect monitors performance *in production* as the last line of defense. Unlike ServiceLab, FBDetect cannot control the machines or run repeated experiments. Instead, to battle against noise, FBDetect introduces advanced techniques to capture stack traces fleet-wide, measure fine-grained subroutine-level performance differences, filter out deceptive false-positive regressions, deduplicate correlated regressions, and analyze root causes.

Both systems have been in production for over seven years. They detect regressions in thousands of services and ML models running on millions of servers. Each year they detect performance regressions that could otherwise lead to the wastage of millions of machines in the following years[1].

CCS Concepts: • **General and reference → Performance**; **Measurement**; **Experimentation**.

Additional Key Words and Phrases: Regressions, A/B testing, time series analysis, root cause analysis

[1]Throughout the paper, we normalize savings as the number of standard CPU machines.

This paper integrates two works, one titled "ServiceLab: Preventing Tiny Performance Regressions at Hyperscale through Pre-Production Testing", originally published in OSDI 2024, and the other titled "FBDetect: Catching Tiny Performance Regression at Hyperscale through In-Production Monitoring", originally published in SOSP 2024, both of which were invited for extended publication in ACM TOCS upon recommendation by the Chairs of OSDI 2024 and SOSP 2024, respectively.

Dong Young Yoon, Meta Platforms Inc, San Mateo, CA, United States; e-mail: dongyoung@meta.com; Rohan Bopardikar, Meta Platforms Inc, Los Altos Hills, CA, United States; e-mail: rohanfb@meta.com; Sherry Chen, Meta Platforms Inc, Sunnyvale, CA, United States; e-mail: zhc047@meta.com; Zoltan Farkas, Meta Platforms Inc, New York, NY, United States; e-mail: zfarkas@meta.com; Caner Gocmen, Meta Platforms Inc, Foster City, CA, United States; e-mail: caner@meta.com; Kapil Goenka, Meta Platforms Inc, Dublin, CA, United States; e-mail: kgoenka@meta.com; Ayichew Hailu; e-mail: ayichew@meta.com; Daniel Hodges, Meta Platforms Inc, New York, NY, United States; e-mail: hodgesd@meta.com; Elvis Huang, Meta Platforms Inc, Hoboken, NJ, United States; e-mail: elvishuang@meta.com; Ivor Huang, Meta Platforms Inc, New York, NY, United States; e-mail: ivorh@meta.com; Zhihui Huang, Meta Platforms Inc, Mountain View, CA, United States; e-mail: kaley.huang@meta.com; Juan Jones, Meta Platforms Inc, Chicago, IL, United States; e-mail: juanjones@meta.com; Osman Kocas; e-mail: osmankocas@meta.com; Jocelyn Kong, Meta Platforms Inc, New York, NY, United States; e-mail: joceyk@meta.com; Abhinay Kukkadapu, Meta Platforms Inc, San Diego, CA, United States; e-mail: abhinayk@meta.com; Yanjun Lin, Meta Platforms Inc, San Carlos, CA, United States; e-mail: yanjun.lin@outlook.com; Matt McNally, Meta Platforms Inc, Natick, MA, United States; e-mail: mattmcnally@meta.com; David Meisner, Meta Platforms Inc, San Francisco, CA, United States; e-mail: meisner@meta.com; Rodrigo Paim, Meta Platforms Inc, Plainsboro, NJ, United States; e-mail: rrpaim@meta.com; Dhruv Patel, Meta Platforms Inc, New York, NY, United States; e-mail: dhruvmpatel@meta.com; Jialiang Qu, Meta Platforms Inc, Jersey City, NJ, United States; e-mail: jialiangqu@meta.com; Santosh Sonawane, Meta Platforms Inc, Livingston, NJ, United States; e-mail: turbosantosh@meta.com; William Wang; e-mail: williamlw@meta.com; Mack Ward, Meta Platforms Inc, New York, NY, United States; e-mail: mack@meta.com; Jonathan Wiepert, Meta Platforms Inc, Greenlawn, NY, United States; e-mail: Jonathan.Wiepert@gmail.com; Miao Yu, Meta Platforms Inc, New York, NY, United States and Meta Platforms Inc, Mountain View, CA, United States and The Ohio State University, Columbus, OH, United States; e-mail: yu.3053@buckeyemail.osu.edu; Bin Zhang, Meta Platforms Inc, Ridgewood, NJ, United States; e-mail: bin23@meta.com; Yunqi Zhang, Meta Platforms Inc, New York, NY, United States; e-mail: yunqifb@meta.com; Chunqiang Tang, Meta Platforms Inc, San Jose, CA, United States; e-mail: tang@meta.com.

Just Accepted

1 Introduction

Meta operates a private cloud with millions of servers. At this scale, even a minor performance regression can result in the waste of many servers. These small regressions are common. For example, among approximately 800 regressions per year in our serverless platform, about one-tenth result in only a 0.005% to 0.01% increase in CPU usage, yet these tiny regressions collectively would have wasted around 4,000 servers if left undetected. Moreover, the median CPU regression is also small, at just 0.048%. These data highlight the critical need for accurate detection of small regressions.

Ideally, developers should test their code or config changes before deploying them to production, so that a regression will not manifest in production. For this purpose, Meta has developed a pre-production 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. Despite its success, we observe that pre-production testing has its inherent limitation: Since the number of machines that can be used for testing is limited, pre-production testing cannot fully reproduce the scale and complexity of the production system. As a result, some buggy code or config changes slip through. To capture such buggy changes, Meta has developed *FBDetect*, an in-production performance monitoring system as the last line of defense.

This paper presents the designs of ServiceLab and FBDetect, two complementary systems to capture tiny performance regressions in Meta's hyperscale private cloud. In particular, while they share the same goal, they face very different challenges due to their environment (i.e., pre-production vs in-production) and thus have adopted very different solutions.

1.1 ServiceLab

At a high level, ServiceLab follows the traditional A/B testing paradigm: A developer reserves a number of machines, deploys the target system on these machines, and then executes a workload to measure its performance. The developer performs these steps on both the old version and the new version of the target system to determine whether their performance differ, which often requires repeating the measurement multiple times for a statistically significant conclusion. Although this paradigm 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, ServiceLab 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 [124], 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 our private cloud runs workloads on Linux containers instead of virtual machines (VMs) so there are no performance variances caused by VMs. The variance caused by containers does not have a significant impact in our experience, especially since ServiceLab has techniques to exclude the warmup period.

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,

Just Accepted

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.

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 [59, 89, 127]. 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 [16, 41, 49, 162], 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. ServiceLab can be configured to either drop the responses from downstream services to the SUT, which may prevent certain code paths from being tested, or replay the responses if the developer is certain that a diverged response is not problematic.

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.

Just Accepted

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 simple 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. In our empirical experience, a majority of the services, which are small and simple, can use ServiceLab without much service owner's participation; a few large and complex services require more participation, but keep in mind that they consume a majority of our fleet capacity [47].

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

1.2 FBDetect

If a buggy code or config change slips through ServiceLab testing, the performance regression will manifest in production. Meta constantly monitors various health metrics (e.g., CPU usage) of each system and records each metric as time-series data. FBDetect performs in-production regression detection by identifying anomalies in such service time-series data. On one hand, directly using production system and traffic alleviates FBDetect from tasks like selecting machines, selecting the right workload, and setting up the dependency services, which are the exact challenges faced by ServiceLab. On the other hand, due to the lack of control of such factors and inability to run repeated experiments, FBDetect faced the key challenge of how to battle against all kinds of noises in production, which is particularly challenging due to our goal of detecting tiny regressions. In addition, FBDetect needs to identify the root cause (i.e., the exact code or config change that caused the regression) among potentially thousands of concurrent changes.

We illustrate several challenges it faces through examples. While Figure 1(a) appears to show no regression, and Figures 1(b) and 1(c) seem to show obvious regressions, the reality is the opposite. Figure 1(a) contains a minor regression of 0.005%, indicated by a barely visible change in the straight line representing the population mean. Despite this subtlety, FBDetect must accurately detect it by managing the high noise. Figure 1(b) shows a rise in a specific subroutine's CPU usage, but FBDetect must filter out this false-positive regression, as it is caused by code refactoring that moves code across subroutines without increasing their total CPU usage. Figure 1(c) shows a drop in throughput, but FBDetect must filter out this false-positive regression caused by transient issues, such as server failures, maintenance operations, load spikes, software rolling updates, canary tests, and traffic shifts, which can last from seconds to hours. Existing methods struggle to handle these transient issues. For instance, typical change-point detection algorithms [10] would result in a 99.7% false positive rate in our environment.

To address these and other challenges, we propose a comprehensive set of techniques, as summarized below.

Subroutine-level regression detection. We significantly reduce the variance (i.e., noise) in performance data by measuring CPU usage at the subroutine level rather than at the overall service level. Detecting a tiny 0.005% regression in a service's overall CPU usage is a daunting task, as illustrated in Figure 1(a). However, if this 0.005%

Just Accepted

regression originates from a single subroutine that consumes 0.1% of the total CPU, the relative change at the subroutine level is $\frac{0.005\%}{0.1\%}$=5%, which is much more substantial. Consequently, small regressions are easier to detect at the subroutine level. This effect occurs because CPU metrics exhibit lower variance at the subroutine level than at the overall service level (§5.1).

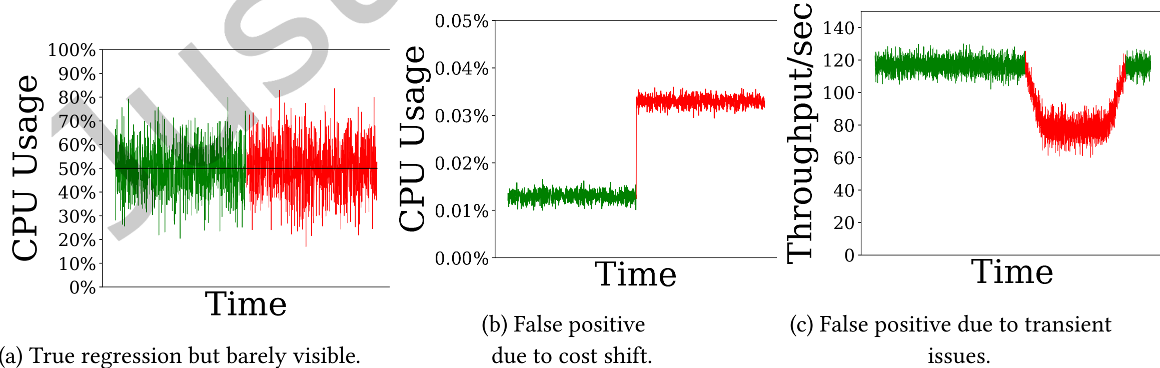
Filtering subroutine-level false positives. Unfortunately, isolated subroutine level metrics can be misleading, as simply moving code from subroutine A to subroutine B may create the illusion of regression in subroutine B, like the one in Figure 1(b). Our evaluation shows that 34% of subroutine-level regressions are false positives caused by these cost shifts—an issue not addressed by existing algorithms. FBDetect employs code analysis to filter out these false positives. It examines higher-level cost domains such as upstream callers or the encapsulating class of the subroutine under investigation. If the cost change in a higher-level cost domain is negligible, FBDetect considers the regression merely a cost shift and will not report it.

Subroutine-level measurement. Instrumenting every subroutine in every service to track subroutine-level CPU usage is not only cumbersome to deploy but also incurs high runtime overhead. To address this issue, we elevate the stack-trace sampling approach to hyperscale, enabling efficient measurement of CPU usage for every subroutine across all services. FBDetect leverages eBPF or a language runtime's ability to periodically collect stack-trace samples fleet-wide. From these samples, it derives each subroutine's relative CPU usage. For example, if 100 stack-trace samples are collected for a service, and a subroutine *foo* appears in 8 of these samples, the normalized CPU usage of *foo* is calculated as 8%.

However, sampling the stack trace of a program written in interpreted languages like Python results in retrieving the interpreter's stack trace, instead of the Python program's stack trace. To address this issue, we introduce *PyPerf*, which uses a kernel eBPF profiler to sample and map the Python interpreter's stack trace to the Python program's stack trace. To our knowledge, PyPerf is the first profiler capable of deriving an end-to-end stack trace across a Python program and the native C/C++ libraries it invokes.

Filtering transient issues. FBDetect accurately classifies 99.7% of regressions, such as the one shown in Figure 1(c), as false positives caused by transient issues, using a combination of sophisticated techniques. These include change point detection [10] to identify anomalies, trend analysis [32] to remove seasonality, Symbolic Aggregate approXimation (SAX) [86] to distinguish anomalies caused by different factors, and methods that examine beyond a single change point to assess whether an anomaly recovers autonomously.

Deduplicating regressions. A single code change can lead to anomalies in numerous monitoring metrics, resulting in an overwhelming number of incident reports. FBDetect leverages clustering algorithms to merge



(a) True regression but barely visible.

(b) False positive due to cost shift.

(c) False positive due to transient issues.

Fig. 1. Challenges for FBDetect: It must catch the barely visible tiny regression in Figure (a). It also must filter out the deceptive false-positive regressions in Figures (b) and (c).

Just Accepted

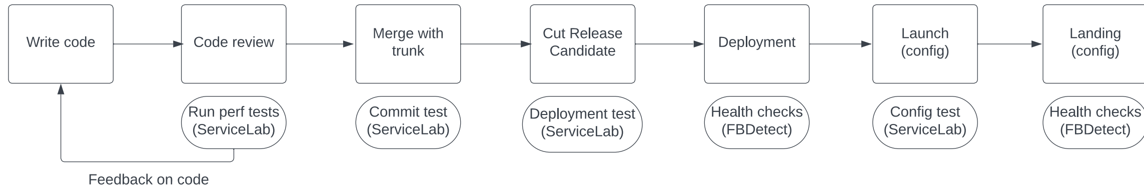


Fig. 2. Development workflow. In this figure, *tests* refer to both functional and performance tests, but this paper focuses on performance tests.

regressions likely caused by the same change. While clustering algorithms are well-known, our unique contributions include: (1) proposing a hybrid clustering algorithm that combines the efficiency of Self-Organizing Map [74] with the accuracy of pairwise clustering, and (2) introducing effective domain-specific features for clustering, such as regression root-cause candidates, subroutine names, and performance-metric names.

Root cause analysis. FBDetect combines multiple techniques to identify the code or config change responsible for a regression: 1) code and stack-trace analysis, where for a regression in subroutine A, code changes that modify downstream subroutines transitively invoked by A are flagged as potential suspects; 2) text analysis, which calculates a similarity score between the regression context (e.g., stack trace) and the code change context (e.g., change description); and 3) time series correlation, which identifies metrics that strongly correlate with the regression's timing.

Contributions. Our primary contribution is a comprehensive set of techniques that enables FBDetect to detect regressions as small as 0.005%—a level of precision previously unreported. Moreover, a key strength of FBDetect over prior work is its battle-tested robustness, proven over seven years of production use and each year catching regressions that would have wasted millions of servers if left undetected.

1.3 Combining ServiceLab and FBDetect to Monitor ML Training

ML training, as the fastest-growing datacenter workload, has created several challenges for regression detection. For ServiceLab, it is usually too costly to run multiple trials of experiments due to the high cost and scarcity of GPUs. For FBDetect, root cause analysis for training jobs is more challenging, since many regressions are caused by changes to the model configuration or the underlying compiler (e.g., PyTorch). These regressions usually do not show up as increased CPU or GPU utilization in user-defined subroutines and thus make FBDetect's stack-trace analysis less effective.

To address these challenges, we have made two efforts. First, we have studied PyTorch related regressions and proposed metrics to improve FBDetect's effectiveness on training workloads. Second, we have combined ServiceLab and FBDetect to build a continuous testing pipeline. It uses ServiceLab to run benchmarks continuously and feeds their results to FBDetect as a time series data. When FBDetect finds a regression, it suggests a list of suspected changes, and then ServiceLab performs a binary search among these suspected changes to identify the true root cause with multi-trial experiments. This approach reduces the cost of testing, since it performs multi-trial experiments only for a subset of suspected changes, and releases FBDetect from the challenging task of root cause analysis. Since their production in the middle of 2024, these efforts have helped Meta save tens of thousands machines.

Just Accepted

2 Overview

Figure 2 depicts our development workflow. Meta uses the monorepo approach [27] 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 Conveyor [47] 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.

In this workflow, ServiceLab, or pre-production testing in general, is applied in the review-time test, commit test, deployment test, and config test, since these tests were executed before deploying the change. FBDetect is applied to in-production health checks. If a regression is found, typically the corresponding diff will either be blocked for deployment (if detected by ServiceLab) or reverted (if detected by FBDetect). The developer will investigate and fix the regression offline, which may take days, and may try to deploy the fixed version later. Of course, if the regression is expected, often for a new feature, the developer can choose to ignore the warnings from ServiceLab or FBDetect.

Pre-production testing and in-production monitoring face distinct challenges, which lead to very different design choices. Concretely, pre-production testing can control the experiment setup (e.g., workload, machines, dependency, ets) to minimize noise during experiments, but it is often challenging to create a testing setup that matches the production setup, especially since the production setup is much larger and continuously changing. On the other hand, in-production monitoring is using the production setup automatically but needs more sophisticated methods to filter noise and identify the root causes of regressions.

Table 1 compares the design choices of ServiceLab, FBDetect, and other standard testing methods. Traditional pre-production testing maintains a dedicated cluster, a synthetic benchmark, and a copy of the dependent services to minimize noise but needs to continuously update them to match the production setup, which is costly. ServiceLab reduces such maintenance cost by running experiments on our private cloud, using record and replay to generate the workload, and using dependent services in production, and introduces techniques to reduce the noise incurred by these methods. In the production environment, a developer often deploys her change in phases, with the help of tools like Canary [9] and Conveyor [137], and then uses FBDetect for long-term monitoring. Both use the production setup. FBDetect has incorporated sophisticated techniques to detect tiny regressions and to identify the root cause (i.e., the diff) of the found regression.

3 Example Workload

This section presents FrontFaaS as an example about how applications execute performance testing with the help of ServiceLab and FBDetect. Section 4.2 and Section 5.2 present more examples for each system respectively.

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

Just Accepted

	Pre-production		In-production	
	Traditional	ServiceLab	Phased Deployment	FBDetect
Machines	Dedicated cluster	Private cloud	Production (subset)	Production
Workload	Synthetic benchmark	Record and replay	Production	Production
Dependent services	Set up own copy	Use production ones	Production	Production
Statistical method	A/B testing	A/B testing	Threshold	Time series analysis
Data points	Repeated experiments	Repeated experiments	-	Call stack sampling; scale
Root cause analysis	Unnecessary	Unnecessary	Unnecessary	Necessary

Table 1. Design choices of different performance testing methods.

Just Accepted

making changes to its code base, with thousands of code commits every workday. FrontFaaS hosts some of Meta's most important products. Due to its scale, complexity, and importance, FrontFaaS has adopted different tools at different stages of development. To be concrete, FrontFaaS uses ServiceLab for pre-production testing, adopts staged deployment to incrementally deploy changes to production servers, and finally relies on FBDetect for in-production monitoring. This section presents the details with ServiceLab and FBDetect. The details about staged deployment can be found in the Conveyor paper [47].

3.1 Testing FrontFaaS with ServiceLab

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 [55, 102], 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 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.

Name	Number of servers used	Number of servers saved yearly	Language	Leverage Stack Trace	Detection Threshold ($\Delta_{\text{threshold}}$)	Re-run Interval	Historical Window	Analysis Window	Extended Window
FrontFaaS (large)	O(100,000)	O(100,000)	PHP	Yes	3%	30 minutes	10 days	3 hours	N/A
FrontFaaS (small)					0.005%	2 hours	10 days	4 hours	6 hours

Table 2. Configurations of FBDetect for FrontFaaS. Periodically, at every "re-run interval," FBDetect analyzes data within the most recent historical window, analysis window, and extended window to detect regressions.

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 §4.4.3.

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* [25]) 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 Monitoring FrontFaaS with FBDetect

To detect regressions at subroutine level, we use the Xenon [110] profiler in the PHP runtime to sample the stack traces of FrontFaaS. Besides subroutine-level regressions, FBDetect detects *endpoint-level regressions* for FrontFaaS. An endpoint is a user-facing URL. As an endpoint request may involve asynchronous and concurrent processing across multiple threads, we use end-to-end tracing [64] to aggregate the costs of all subroutines involved. Regressions in this aggregated cost are called *endpoint-level regressions.* Moreover, FBDetect detects metadata-annotated regressions for FrontFaaS. A subroutine can annotate its stack

Just Accepted

frame by calling *SetFrameMetadata()* to provide additional context. This is useful for detecting regressions that occur only under certain conditions, e.g., processing requests on behalf of a specific category of users.

Table 2 shows two FBDetect configurations that are used for FrontFaaS simultaneously. One detects large regressions (3%) more quickly, while the other detects small regressions (0.005%) but needs to wait longer to collect more data.

4 Pre-production Testing with ServiceLab

This section presents the details of ServiceLab.

4.1 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.

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; see the DiffSuggester in Section 3.1);
- 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.

4.2 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. While we have already presented FrontFaaS in Section 3.1, we describe a few additional representative and large workloads below.

4.2.1 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

Just Accepted

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 ShardManager [81], a shared, central control plane, 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 ServiceLab starts an experiment, it selects the requests that match the shards loaded in from the experiment's static shard assignment. Additionally, ServiceLab modified the shard IDs of those requests from the recorded dynamically allocated shard ID to the static shard assignment in the experiment.

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%).

4.2.2 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 [7, 148], 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%.

4.2.3 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%).

4.2.4 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,

Just Accepted

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.

4.2.5 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%).

4.2.6 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 [47], 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 [100, 138] 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 [47]. So far, ServiceLab has relied entirely on organic, bottom-up adoption without top-down push.

4.3 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

Just Accepted

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 (see Section 2). 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 [50]. 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.

4.3.1 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 [133] 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

Just Accepted

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 [93], 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 work [93], 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.

Just Accepted

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 [75], they may still introduce greater performance variability than bare metal machines. We have not validated our method in virtualized environments.

4.3.2 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** [38]. 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** [11]. 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 [39], which are particularly effective in moderating the elevated variance observed when services operate under strenuous

Just Accepted

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 [20] (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.

Typically the warm-up phase takes about 30 to 60 seconds. If the service has a longer warm-up time, the service can expose an interface to ServiceLab that says it is currently warming up and data should be ignored until the service is "warmed up". ServiceLab polls the interface until the service has reported itself to be warmed up. For example, HHVM, which has a JIT compiler, monitors the amount of code that has been compiled and the rate of new code compilation. Once both have reached the warmed up threshold, it notified ServiceLab that warm up is complete.

Just Accepted

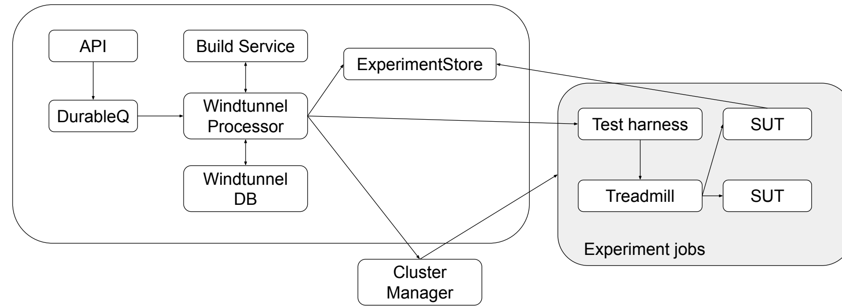


Fig. 3. ServiceLab architecture. *Windtunnel* is the orchestration engine and *Treadmill* replays traffic and runs tests.

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.

Artifical A/B experiments. ServiceLab also conducts artificial A/B experiments, which inject regressions on the B side, to understand how accurately ServiceLab can identify regressions. ServiceLab injects regressions in two ways. First, ServiceLab can ask compilers to inject delays in the application's code before an experiment starts. For example, we worked with the HHVM engineers to have the JIT compiler insert a noop instruction every N instructions (e.g., a noop instruction every 100 instructions would equate to a 1% CPU instruction regression). Second, after an experiment finishes, ServiceLab can also manipulate the collected time series data to simulate a regression before running the statistical analysis.

The ServiceLab team relies on the periodic A/A and artificial A/B experiments to continuously evaluate the accuracy of different models. If some metric incurs a high false positive or false negative rate, maybe due to a deviation from the expected data distribution, the team will further investigate the data and tune the models.

4.4 ServiceLab Design

This section presents the design of ServiceLab, utilizing the architecture diagram shown in Figure 3 in our discussion.

4.4.1 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

Just Accepted

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 (§4.3.1), 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 (§4.3.1).

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 [9, 36, 137] 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. Throughout 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.

4.4.2 Traffic Record-and-Replay. At Meta, all services use the Thrift [129] 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 [7, 148] 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. To be concrete, Thrift allows a client to create a "stream", and the routing layer ensures that RPC requests of the same stream are sent over the same connection to the same host. As requests of the same stream probably have dependencies, the service can record all requests of a subset of streams.

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. For example, one shard (or set of containers) might serve one type of application, while another shard (or set of containers) might serve another (see Section 4.2.5 for an example). The service owner has to pick whether to record one shard or another by selecting the container(s) they would like to record, or they can use to record from a mix of containers to record a diverse set.

ServiceLab leverages the open-source load testing platform Treadmill [161] 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.

Just Accepted

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, as discussed in Section 4.3.2, services may have a warm-up period. ServiceLab can be configured to allowing a lower request rate to be set during this period to gradually increase the load on the service.

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.

4.4.3 Handling Service Dependencies. Meta products are built out of tens of thousands of services with intricate interdependencies, akin to those documented in prior research [59, 89, 127]. 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 (§4.3.1).

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, §4.5.1 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.

4.5 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?

Just Accepted

4.5.1 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 (3,085 were fixed and 88 were expected), 15% (823) are false positives, and 28% (1,564) are unknown. Assuming the unknowns have the same false positive rate as others, ServiceLab achieves a precision of $\frac{3173}{823+3173} = 79.4\%$ and a false positive rate of approximately $23\% \times 0.3\% \times \frac{15\%}{57\%+15\%} = 0.014\%$, since most cases are true negatives.

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 rarely receive a false-positive ticket, 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 4 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% × 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.

To approximate the false negative rate, we rely on reports from FBDetect. FBDetect examines the per-subroutine CPU usage 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 $\frac{2038}{2038+5560\times\frac{57\%}{57\%+15\%}}$. 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 FBDetect 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.

4.5.2 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 5 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 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.

Just Accepted

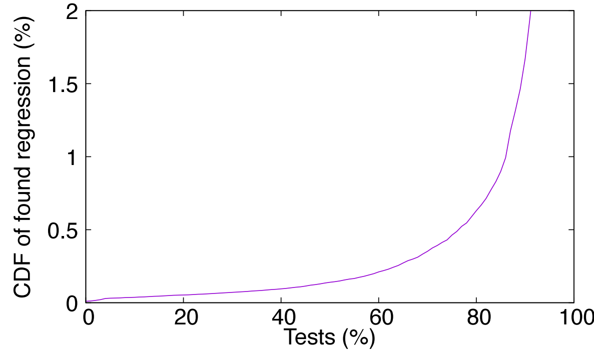


Fig. 4. Cumulative distribution of regressions detected by ServiceLab for FrontFaaS.

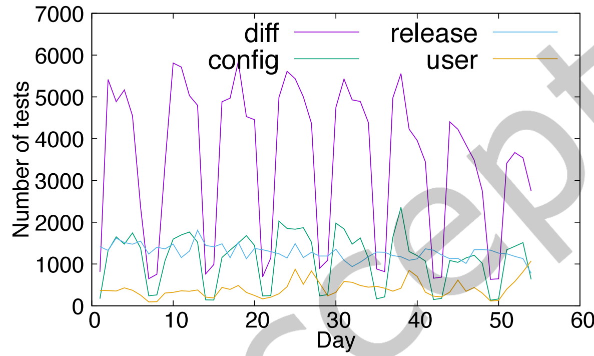


Fig. 5. Number of experiments completed each day.

Just Accepted

- 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 6 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 7 shows, for one service,

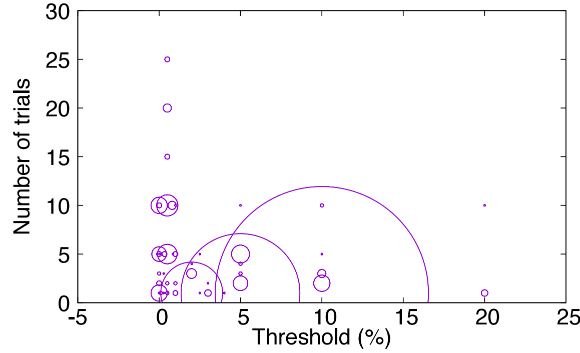


Fig. 6. Thresholds and number of trials. The size of the circle represents the count of use cases with the same setting.

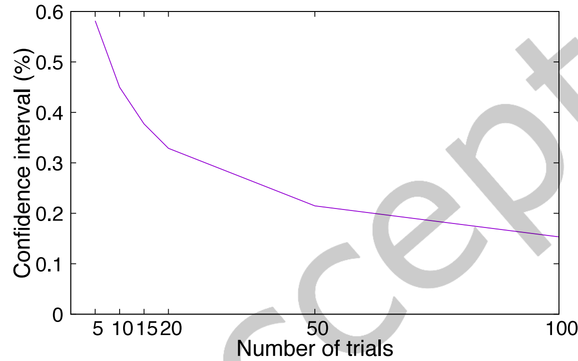


Fig. 7. The number of trials required for detecting small regressions.

Just Accepted

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 5. 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.

4.5.3 Examples. In this section, we present a few examples about true positives, false positives, and false negatives of ServiceLab. They include examples from FrontFaaS and other services.

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 (§4.2.4) 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 (§4.2.5), 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.

In Meta, ServiceLab is also used to test different OS kernel versions. For this purpose, ServiceLab is configured in a non-standard manner, using different versions of OS kernels but the same version of the application code during its A/B testing. When Linux 6.9 was released, ServiceLab experiments found that it introduced a significant regression to our storage service. Our investigation showed that this is probably due to the new deadline-based CPU scheduler introduced in Linux Kernel 6.6+ to replace the Completely Fair Scheduler that's been in mainline since 2.6. When we purposefully induce contention under 6.9, we see 10x higher kernel scheduler delays than on 6.4. We did further testing with kernel 6.11 and 6.13. 6.11 doesn't help, but 6.13 brings the cpu-busy metric back to 6.4 levels and eliminates the contention issues we were having under 6.9.

Example of False Positives. In one incident spanning over a week, HHVM experienced a significant number of false positives. The issue stemmed from a production experiment where a configuration change was enabled on 1% of all machines running HHVM, including machines running ServiceLab experiments. The configuration change enabled a computationally expensive feature in HHVM's JIT compiler, changing the performance characteristics. To remediate this issue, the production experiment was modified to exempt applying the configuration to machines running ServiceLab experiments.

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.

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 (§4.2.4), 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.

Not enough trials is another source of false negatives. Our investigation shows that in some cases, ServiceLab identifies a regression in the experiment results, but the confidence level is not high enough for ServiceLab to trigger the warning. With more trials, ServiceLab probably would have generated the warning. That's why we introduced the artificial A/B tests to help users tune such settings.

Just Accepted

Effects of Using Downstream Services in Production. As discussed earlier, ServiceLab allows the SUT to invoke downstream services in the production environment. While this approach simplifies the setting of the experiments, it can introduce both false positives and false negatives.

In early versions of ServiceLab, we found timing of tests can be a source of false positive or false negative, since tests share the same downstream service. For example, in an A/B test, early versions of ServiceLab start both together but do not synchronize their requests. As a result, one side may be slightly ahead of the other consistently. When this happens, the read requests of the faster side hit the downstream service first, and may cause the corresponding data to be loaded into cache. Then when the read requests of the slower side arrives at the downstream service, they can be served from the cache directly, causing an unintended bias in the result. Depending on whether A side or B side is faster, this problem can show up as either false positives or false negatives. This problem has motivated us to synchronize the timing of requests to both sides in later versions of ServiceLab.

In another example, we found running experiments in a data center region different from where the original requests were recorded can cause problems as well. For example, for FrontFaaS, if we record requests of an European user and replay the requests in US, it could happen that those requests can mostly hit cache in downstream services in Europe but will need cold start in downstream services in US. Such change of behavior could hide or exacerbate certain problems, causing either false positives or false negatives.

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. While we allow the SUT to invoke downstream services in the production environment directly for simplicity, it can introduce both false positives and false negatives and thus should be managed carefully.

4.5.4 False Positive in A/A Experiments. As described in § 4.3.2, 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 signaling 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 (§4.3.2).

4.5.5 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.

Just Accepted

5 In-production Monitoring with FBDetect

This section presents FBDetect in details.

5.1 Feasibility of Detecting Tiny Regressions

At first glance, detecting the tiny regression in Figure 1(a) seems implausible, so we first address its feasibility.

We develop a simple analytic model to aid in the discussion. Suppose we collect n performance samples after a code change to assess its performance impact. Let σ^2 denote the sample variance, and $\Delta_{\text{threshold}}$ denote the detection threshold—the smallest performance difference that can be reliably detected, such as a 0.005% difference in CPU usage. In Appendix A.2, we derive the following expression, where the symbol $\propto$ denotes proportionality.

$$\Delta_{\text{threshold}} \propto \sqrt{\sigma^2/n} \tag{1}$$

We lower the detection threshold $\Delta_{\text{threshold}}$ by simultaneously reducing σ^2 and increasing n. While a hyperscale environment can more easily increase n by collecting samples across many servers, it also exhibits high variance (σ^2) due to factors like mixed server generations and diverse request types. Thus, relying solely on fleet size to increase n, without FBDetect's optimizations to reduce σ^2, would take weeks to years to collect enough samples for a low $\Delta_{\text{threshold}}$, even at Meta's hyperscale. FBDetect's optimizations reduce σ^2 by 100-10000 times, obtaining sufficient samples within hours or a few days (§5.2). Additionally, reducing variance is crucial for minimizing fleet-wide resource waste (Appendix A.4).

Subroutine-level measurements. To lower $\Delta_{\text{threshold}}$, we reduce the variance σ^2 by measuring CPU usage at the subroutine level rather than at the Linux process level.

For simplicity, assume a process comprises k subroutines with independent and identically distributed (IID) CPU usage.[2]

Let random variables X_{process} and $X_{\text{subroutine}}$ denote the CPU usage of the process and each subroutine, respectively, with $X_{\text{process}} = \sum_{i=1}^{k} X_{\text{subroutine}_i}$. Then,

$$\mathit{Variance}(X_{\text{subroutine}}) = \mathit{Variance}(X_{\text{process}})/k. \tag{2}$$

According to Expression 1, the smaller variance at the subroutine level allows detection of regressions $\frac{1}{\sqrt{k}}$ times smaller.

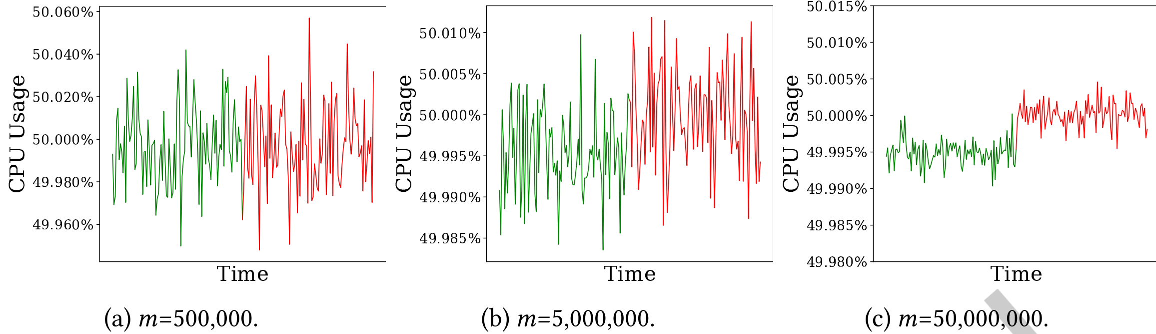
While this analysis shows that using a subroutine A's absolute-CPU-usage metric X_A can detect small regressions, Appendix A.3 shows that using the relative-CPU-usage metric gCPU_A achieves the same effect, where $\text{gCPU}_A = \frac{X_A}{X_{\text{process}}}$. We prefer gCPU_A over X_A because gCPU_A can be more easily calculated from stack-trace samples. For example, if 100 stack-trace samples are collected, and subroutine A appears in 8 of these samples, gCPU_A=8%.

For a hyperscale service, its number of subroutines (k in Expression 2) can be very large. Excluding negligible subroutines, we call those with a gCPU of 0.001% or higher as "non-trivial." The non-trivial subroutines in our serverless platform have a median gCPU of 0.0083%. Accordingly, we estimate the k in Expression 2 as k=1/0.0083%=12,048. The large value of k significantly reduces the variance in Expression 2, enabling detection of small regressions.

In addition to CPU metrics, FBDetect can support other subroutine-level metrics, such as latency, throughput, and error rate per RPC endpoint. However, memory and other application-level metrics require manual instrumentation if subroutine-level detection is desired.

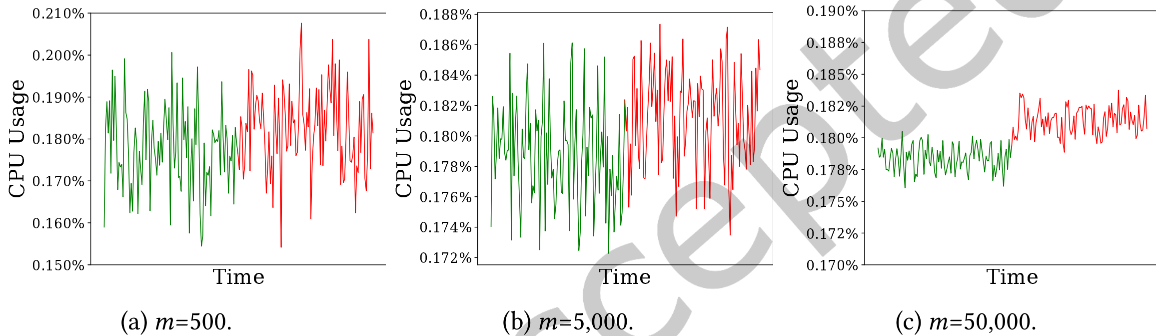
[2] Although subroutines generally do not follow the IID assumption and are hard to analyze mathematically, the essence of this simplified analysis holds—the process-level variance is decomposed across numerous subroutines, resulting in much smaller variance at the subroutine level.

Just Accepted



(a) m=500,000. (b) m=5,000,000. (c) m=50,000,000.

Fig. 8. The average of m time series from m servers measuring Linux-process-level CPU usage. As m increases, noise reduces. This effect can be explained using the Law of Large Numbers, as described in Appendix A.1.



(a) m=500. (b) m=5,000. (c) m=50,000.

Fig. 9. The average of m time series from m servers measuring subroutine-level CPU usage. This figure uses samples from 1000 times fewer servers than Figure 8.

Just Accepted

Validation via simulation. Production evaluation of our approach for detecting small regressions will be presented in §5.5. Here, we validate its feasibility through simulations. In these simulations, we conservatively set k=1000.

Figure 1(a) simulates CPU usage data collected from a single server, sampling from a normal distribution with mean μ=0.5 (i.e., 50% CPU usage) and variance σ^2=0.01, while capping sample values within [0, 1]. In the second half of the time series, the mean increases to 50.005%, representing a 0.005% regression, though this change is barely visible.

Figure 8 simulates sampling from many servers. We sample from m servers, generate m time series like the one in Figure 1(a), average them, and plot the average for various values of m. To simulate servers of different generations, the m servers exhibit different performance. Samples from half of the m servers have μ=40% and σ^2=0.01, with the mean changing to μ=40.003% mid-series to simulate a 0.003% regression. The other half have μ=60% and σ^2=0.02, with the mean changing to μ=60.007% mid-series. The regression amounts, 0.003% and 0.007%, differ because a code change may perform differently across server generations. Figure 8(c) shows that the tiny regression can be detected with sufficient samples, though sampling from 50,000,000 servers is impractical.

Figure 9 simulates subroutine-level measurements. The Linux-process-level CPU usage in Figure 8 is distributed across k=1000 subroutines.[3] The lower variance at the subroutine level (see Expression 2) enables Figure 9 to

[3]The sample mean in Figure 9 is higher than μ=50%/1000=0.05% because filtering out negative samples from the normal distribution with mean μ=0.05% raises the sample mean above 0.05%.

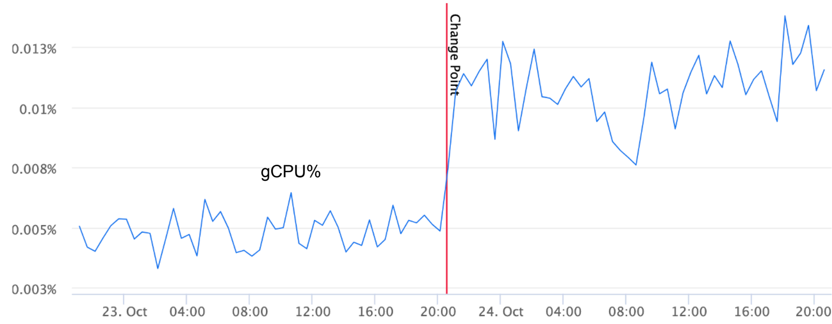


Fig. 10. A real-world example of detecting tiny regressions with subroutine-level measurement.

Name	Number of servers used	Number of servers saved yearly	Language	Leverage Stack Trace	Detection Threshold ($\Delta_{\text{threshold}}$)	Re-run Interval	Historical Window	Analysis Window	Extended Window
FrontFaaS (large)	O(100,000)	O(100,000)	PHP	Yes	3%	30 minutes	10 days	3 hours	N/A
FrontFaaS (small)					0.005%	2 hours	10 days	4 hours	6 hours
PythonFaaS (large)	O(100,000)	O(100,000)	Python	Yes	0.5%	1 hour	10 days	6 hours	N/A
PythonFaaS (small)					0.03%	4 hours	10 days	6 hours	6 hours
TAO (FrontFaaS)	O(100,000)	O(10,000)	C++	Yes	0.05%	2 hours	10 days	4 hours	1 day
TAO (non-FrontFaaS)					0.05%	1 hour	10 days	1 day	6 hour
AdServing (short)	O(1,000,000)	O(10,000)	C++	Yes	0.2%	6 hours	10 days	1 day	12 hours
AdServing (long)					0.1%	1 day	16 days	9 days	N/A
Invoicer (short)	O(10)	Negligible	C++	Yes	0.5%	12 hours	14 days	1 day	1 day
CT-supply (short)	Diverse	O(10,000)	Diverse	No	5% (relative)	12 hours	7 days	1 day	1 day
CT-supply (long)					5% (relative)	12 hours	10 days	7 days	1 day
CT-demand					5% (relative)	12 hours	7 days	1 day	N/A

Table 3. Configurations of FBDetect for different workloads. Periodically, at every "re-run interval," FBDetect analyzes data within the most recent historical window, analysis window, and extended window to detect regressions. FBDetect can be configured to use either an absolute threshold (first nine rows) or a relative threshold (last three rows). For example, an increase of gCPU from 1% to 1.1% is a 0.1pp absolute change and a 10% relative change.

Just Accepted

detect the tiny regression by sampling from 1000 times fewer servers than Figure 8, making it practical for production use.

Although the simulation focuses on large services, subroutine-level measurements also enable accurate regression detection in small services. For instance, FBDetect can detect regressions as small as 0.5% for our *Invoicer* service running on only 16 servers, as described in the next section,

Read-world example. Figure 10 shows a real-world example about how subroutine-level measurement helps us detect tiny regressions. As shown in the figure, a change caused about 0.008% regression in CPU utilization, which would be very hard to detect in the overall system. However, when measured at the subroutine level, since the subroutine originally consumed only 0.005% of the CPU, this 0.008% regression became quite obvious and easy to detect.

5.2 Workloads

In this section, we summarize the diverse workloads supported by FBDetect. While prior methods have demonstrated effectiveness on certain workloads, for large-scale adoption, the biggest challenge is ensuring a method's

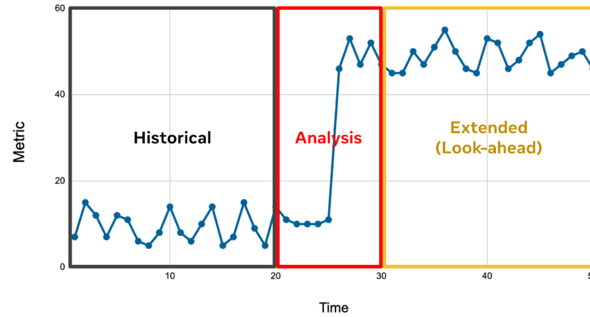


Fig. 11. Time windows used in regression detection.

robustness across diverse workloads. Robustness is a key strength of FBDetect. Currently, FBDetect monitors around 800,000 time series to detect regressions in hundreds of services. These time series are from a wide range of metrics, including CPU, memory, throughput, latency, error rate, coredump count, and many application-level metrics.

Among services supported by FBDetect, about 500 use stack-trace sampling. Their size, i.e., the number of servers they consume, varies from five to more than half a million servers. The P5 (5th percentile), P10, P50, and P90 of the service sizes are 23, 64, 1,251, and 40,527, respectively. This shows that FBDetect works for services big and small.

As the workload descriptions involve the setup of detection windows, we explain the concept below. FBDetect periodically scans a service's time-series data to detect regressions. It divides the time series into three parts, as shown in Figure 11: (1) the *historic window*, which serves as the baseline for comparison; (2) the *analysis window*, where regressions are reported by comparing its data against that of the historic window; and (3) the *extended window*, which evaluates whether an observed regression persists or disappears. Below, we describe several workload examples in addition to FrontFaaS as discussed in Section 3.2.

PythonFaaS is Meta's serverless platform for Python code. For PythonFaaS, FBDetect detects regressions in subroutines and endpoints, as well as per-data-type I/O regressions to the downstream database (see TAO below).

TAO [26] is a graph database. For its traffic from FrontFaaS and PythonFaaS, FBDetect detects regressions in subroutines, endpoints, and per-data-type I/Os. For other traffic, FBDetect detects regressions in query-processing throughput.

AdServing is a group of ultra-large services that work together to serve ads to different products.

Invoicer is a small service running on just 16 servers to generate billing invoices. To ensure sufficient stack-trace samples, eBPF collects about one sample per server per second for Invoicer, compared to one sample per server per minute for FrontFaaS. Additionally, it uses long historical, analysis, and extended windows of 14 days, 1 day, and 1 day, respectively. These settings enable FBDetect to collect sufficient samples for detecting gCPU regressions as small as 0.5% in this small service.

Capacity Triage (CT). CT is a tool that leverages FBDetect to detect throughput regressions for a diverse set of services. CT relies on Kraken [147] to benchmark a service's per-server maximum throughput. If this maximum throughput unexpectedly drops, it is a regression on the supply side ("CT-supply" in Table 3). Additionally, if the total peak requests to a service's all servers unexpectedly increase, it is a regression on the demand side ("CT-demand" in Table 3).

Summary. The examples in Table 3 demonstrate FBDetect's ability to detect regressions across diverse workloads. Their historical, analysis, and extended windows range from hours to days. These long windows allow FBDetect to gather sufficient samples to enable a small detection threshold. This highlights the importance of using

Just Accepted

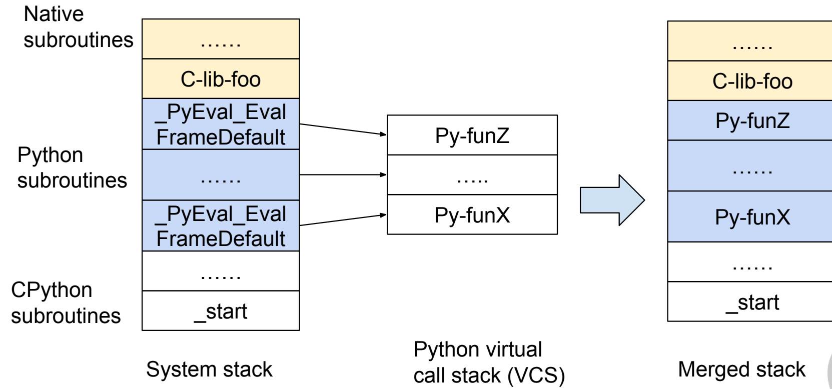


Fig. 12. How PyPerf reconstructs the stack trace.

Just Accepted

fine-grained subroutine-level measurements to reduce variance. Without significantly reducing variance, it would take 100-10000 times longer to obtain enough samples, causing unacceptable delays.

5.3 Performance Profiling Using Stack Traces

Before detailing FBDetect's regression detection algorithms, we describe how FBDetect collects performance data as its inputs, focusing on fine-grained subroutine-level CPU data.

We periodically collect stack traces across the entire fleet to infer relative time spent in each subroutine. For example, if 100 stack-trace samples are collected for a service, and subroutine *foo* appears in 8 samples, its normalized CPU usage (gCPU) is 8%. Note that the gCPU of a subroutine includes not only the cost of the subroutine itself but also the child subroutines recursively invoked by the subroutine.

However, sampling the stack trace of a program written in interpreted languages results in retrieving the interpreter's stack trace, instead of the program's stack trace. To address this, we use different solutions for different interpreted languages. Java and PHP virtual machines offer built-in support for generating stack traces [60, 109, 110]. For Python, we developed an eBPF-based profiler called *PyPerf*, which handles various Python versions and provides end-to-end stack traces across both Python code and the C/C++ native libraries it invokes.

PyPerf utilizes an eBPF probe in the Linux kernel to collect stack traces from CPython, as shown in Figure 12. The stack trace comprises: 1) a sequence of calls internal to CPython, 2) a sequence of *_PyEval_EvalFrameDefault* calls, and 3) a sequence of calls to native C/C++ libraries invoked by the Python program. Our key insight is that each *_PyEval_EvalFrameDefault* call in CPython's C code maps precisely to a corresponding call in the Python code. This enables us to reconstruct the end-to-end stack trace as follows.

CPython maintains a virtual call stack (VCS) for a Python program, akin to the call stack in C programs. The VCS is a linked list of frames, each containing information about the source-code address of the corresponding Python subroutine. The head of the VCS is stored at a fixed location within CPython. PyPerf's eBPF probe walks through the VCS, starting from its head, to reconstruct the call stack of the Python program by mapping *_PyEval_EvalFrameDefault* calls to subroutines recorded in the VCS.

PyPerf produces a precise end-to-end stack trace by merging 1) the native call stack from CPython, 2) the Python code's call stack as described above, and 3) the native call stack of C/C++ libraries invoked by the Python code. In contrast, the state-of-the-art Python profiler, Scalene [18], can only approximate the time spent in C/C++ libraries since its Python-level profiling cannot reach into C/C++ code.

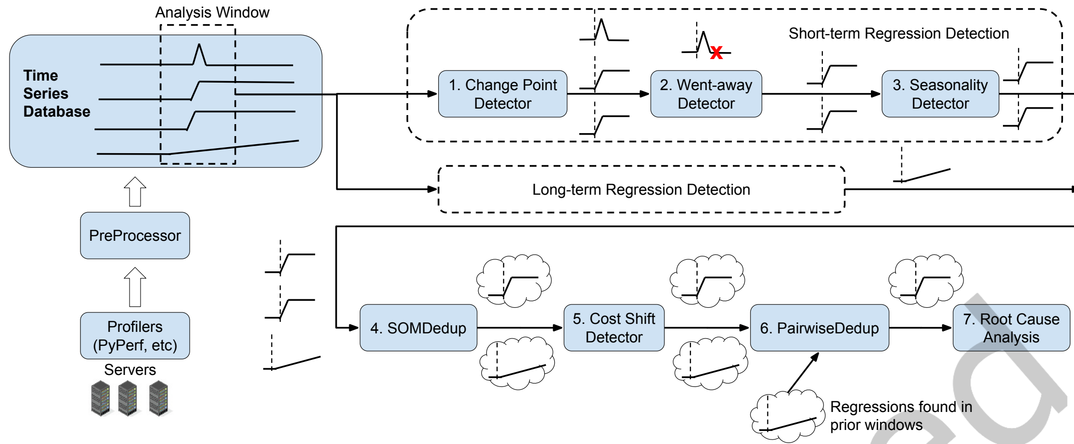


Fig. 13. Workflow of FBDetect.

Although Python 3.12 introduced stack-trace collection support in October 2023 [45], FBDetect has required this capability since 2017. Python 3.12 adds to the call stack a frame with the corresponding Python function name for each *_PyEval_EvalFrameDefault* call, enabling tools like Linux's `perf` to map *_PyEval_EvalFrameDefault* to Python functions. However, this approach has several limitations. First, the additional frame introduces significant overhead [112, 140], which cannot be mitigated by sampling. Second, it may interfere with Just-In-Time (JIT) optimizations, leading to further performance degradation [140].

5.4 Regression Detection Algorithms

In this section, we present FBDetect's detection algorithms, starting with an overview and then providing detailed explanations of FBDetect's individual techniques.

5.4.1 Overview. We follow Figure 13 to provide an overview of FBDetect. In the bottom left of the figure, profilers such as PyPerf periodically capture stack-trace samples on all servers (§5.3), which are then converted to subroutine-level gCPU time series. FBDetect periodically examines these time series within a recent time window to detect regressions.

We define a regression as a shift in the mean of a time series. Without loss of generality, we assume that an increase in a metric's value means a regression. There are two types of regressions: a sudden change resembling a step function and a gradual incremental change over a longer period. Accordingly, we have designed two separate algorithms for short-term and long-term regression detection. The short-term algorithm is more sensitive to sudden changes but is carefully designed to filter out noisy, transient changes, while the long-term algorithm is insensitive to sudden changes and focuses on the long-term trend.

The "*short-term regression detection*" path in Figure 13 executes the following steps in sequence:

(1) The change point detector applies change point detection [10] to identify anomalies, which are regression candidates. (§5.4.2)
(2) The went-away detector filters out regressions that disappear spontaneously, like the one in Figure 1(c). (§5.4.3)
(3) The seasonality detector applies trend analysis [32] to filter out regressions caused by seasonality. (§5.4.4)
(4) The SOMDedup clustering algorithm performs a fast, shallow analysis to efficiently deduplicate regressions likely caused by the same change. (§5.4.7)

Just Accepted

(5) The cost shift detector filters out regressions resulting from code refactoring that merely shifts cost between subroutines, like the one in Figure 1(b). (§5.4.6)
(6) The PairWiseDedup clustering algorithm performs a slower, more thorough comparison to further deduplicate remaining regressions. (§5.4.8)
(7) Finally, root cause analysis is applied to each remaining regression to pinpoint the specific code or configuration change responsible for the regression. (§5.4.9)

The specific ordering of steps 2–6 is designed to execute faster algorithms in the early steps to filter out as many regressions as possible, thereby reducing computation in the later, more resource-intensive steps.

For ease of operation, FBDetect runs on a common serverless platform at Meta, scanning different time series in parallel. Overall, it utilizes capacity equivalent to hundreds of servers, analyzing approximately 800,000 time series to detect regressions across hundreds of services.

In the subsequent sections, we detail FBDetect's individual techniques outlined in Figure 13.

5.4.2 Change Point Detector. This detector applies the Cumulative Sum (CUSUM) [14] and Expectation Maximization (EM) [97] algorithms iteratively to identify change points. This process continues until it converges at the change point with the maximum likelihood of having different means before and after the change point, or until it uses up the computation time. Once a change point is identified, FBDetect conducts a statistical hypothesis test to validate it:

- Null hypothesis H0: there is no change point in the time series and there is only one mean μ;
- Alternative hypothesis H1: there is one change point t in the time series, the mean before t is μ_0 and after t is μ_1.

FBDetect conducts the likelihood-ratio chi-squared test [146] with the significance level of 0.01, and reports a regression only if the null hypothesis is rejected.

Overall, we find change-point detection algorithms necessary but insufficient for detecting small regressions in noisy environments. Specifically, for transient issues like the one in Figure 1(c), they either require a long time window to filter them out as noise, which delays regression detection, or a large threshold, which fails to catch small regressions. This prompted us to introduce the went-away detector.

5.4.3 Went-away Detector. Filtering out transient regressions is challenging, and our algorithm has undergone multiple iterations. In the first iteration, FBDetect conducted an additional CUSUM analysis using the data after the change point. The purpose was to find an inverse regression and check whether its magnitude sufficiently compensates for the original regression. However, this method was too sensitive to transient issues after true regressions. For instance, if the time series temporarily dips and then quickly recovers after a true regression, this method would incorrectly filter out this true regression.

In the second iteration, to improve robustness, we used the average trend instead of a single change point. We added a short-term trend analysis using the Mann-Kendall test [70, 92] to check whether the end result of a given regression shows a decreasing trend, which might indicate the regression went away. However, a decreasing trend itself is insufficient, since the value needs to recover to the normal level for the regression to be considered went-away. Therefore, if the Mann-Kendall test shows a decreasing trend, FBDetect further compares the end values of the regression with values in a historical window. Unfortunately, choosing the right historical window turned out to be difficult. For example, in Figure 14, if the algorithm happens to choose the window with a spike as the baseline, it will mistakenly conclude that the regression at the end of the time series is a false positive. In practice, we found that this happens quite frequently.

In the third iteration, which is our current version, we added an additional logic to further improve robustness. If the values within the time window after a change point are "very different" from those within the window after

Just Accepted

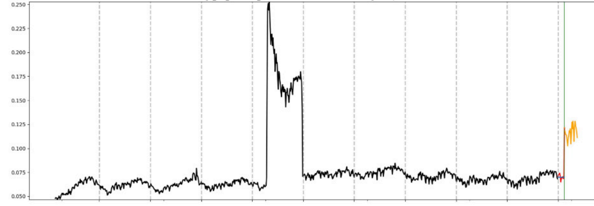


Fig. 14. Catching the regression at the end.

another change point, we consider them to be caused by different reasons. With this method, we can identify that, in Figure 14, the regression at the end and the spike in the middle are caused by different reasons.

For real-number values, determining whether they are "very different" can be challenging, as all numbers differ to some extent. To address this, FBDetect discretizes the time series into a string representation using Symbolic Aggregate approXimation (SAX) [86]. SAX divides the value range into buckets, replacing values in each bucket with a corresponding letter. For example, a time series like [1.1, 2.0, 3.1, 4.2, 3.5, 2.3, 1.1] can be represented as the string 'abcdcba', using four buckets where 'a' represents [1, 2) and 'b' represents [2, 3), and so on. SAX is configurable: among N buckets, a bucket is considered valid only if it contains at least $X\%$ of the data points. We tested various combinations of N and X and settled on N=20 and X=3%, which proved robust to outliers without missing obvious regressions.

After outlining the key ideas of the went-away detector, we now present it formally. FBDetect marks a regression as true if the following predicate evaluates to true:

NewPattern **OR** [SignificantRegression **AND** LastingTrend **AND** (**NOT** RegressionGoneAway)].

The terms are defined as follows.

- NewPattern: This term checks if the post-regression pattern significantly differs from historical patterns. If so, FBDetect reports the regression as true. In the SAX string representation, a letter is valid if its number of occurrences exceeds a predefined threshold. If most letters in the post-regression SAX string are invalid, FBDetect treats the post-regression time series as a new pattern and reports a regression, unless the average value is lower than the lowest valid bucket in historical data, indicating no significant cost increase despite the new pattern.
- SignificantRegression: This term checks if the regression magnitude is significant. FBDetect considers the regression significant if the largest letter in the post-regression analysis window is greater than or equal to the largest valid letter pre-regression. Additionally, FBDetect verifies that the 90th percentile of values after the change point exceeds both the 95th percentile of the historical window and the 90th percentile of the previous day, confirming the regression's significance.
- LastingTrend: This term assesses whether a regression trend persists after the change point. FBDetect performs the Mann-Kendall test on both the post-regression and the entire analysis window to detect monotonic upward trends. If a trend is found, FBDetect uses Theil-Sen's Slope Estimator [141] to measure its magnitude and intercept. The window with the lower slope is used to avoid over- or under-estimation. FBDetect then compares the slope to a regression threshold, calculated as the Median Absolute Deviation [82] with a normality constant of 1.4826, and applies a regression coefficient (default 1.5) for sensitivity. The final regression threshold is set to $\textit{coefficient} \times \textit{median} \times 1.4826$.
- RegressionGoneAway: This term checks whether the regression has gone away in the last few data points, serving as the final sanity check.

The combination of these terms form a robust predicate that helps FBDetect effectively filter out transient issues.

Just Accepted

5.4.4 Seasonality Detector. This detector removes seasonality from the time series and then checks if the regression still exists. To determine if seasonality is present in the time series, FBDetect applies an autocorrelation function and checks if the correlation is significant. If so, FBDetect runs the Seasonal and Trend decomposition using Loess (STL) algorithm [32] to decompose the time series into three parts: SEASONALITY, TREND, and RESIDUAL. FBDetect then removes SEASONALITY, computes the median values of TREND + RESIDUAL before and after the CUSUM change point, and computes the difference between these two median values. Finally, FBDetect normalizes the difference by the standard deviation of the RESIDUAL and calculates the pseudo 'z-score' [134]. If the 'z-score' is smaller than a given threshold, FBDetect filters out the regression as a false positive caused by seasonality. FBDetect computes the z-score in both the analysis window and the extended window, and requires both to be smaller than the threshold. Overall, we find that this method can remove most false positives caused by seasonality while introducing very few false negatives.

Discussion of alternatives. As an alternative to the STL algorithm, we also experimented with using the moving-average algorithm [23] to handle seasonality. We found that STL is superior because it is sensitive to slight changes in seasonality while being robust against sudden changes.

5.4.5 Long-term Regression Detection. The long-term regression detection algorithm consists of three steps: seasonality decomposition, regression detection, and change-point detection. FBDetect first uses the STL algorithm to decompose the original time series into SEASONALITY, TREND, and RESIDUAL. FBDetect then conducts regression detection on the TREND time series alone to determine if a regression is present. If so, FBDetect runs change-point detection to locate the change point. The following presents regression detection and change-point detection in detail.

In the regression-detection step, FBDetect calculates the means at the start of the analysis window and the historical window, and uses the bigger one as the BASELINE. Similarly, FBDetect computes the means at the end of the analysis window and the extended window, and uses the smaller one as the CURRENT value. If the difference between CURRENT and BASELINE is above a given threshold, FBDetect reports it as a long-term regression.

In the change-point detection step, FBDetect checks if the regression represents a gradual change by running a linear regression model to fit the normalized TREND and calculating the root mean square deviation (RMSE). If the RMSE is smaller than a given threshold, FBDetect sets the change point at the beginning of the TREND. Otherwise, FBDetect uses the normal loss and dynamic programming search [145] to find the change point. It aims to identify the partition point that minimizes the variance on both sides, with the partition point being the change point.

The long-term detection algorithm is similar to the short-term one but has several key differences. First, the long-term algorithm runs seasonality detection as the first step, while the short-term one runs seasonality detection as the last step. This is because seasonality detection smooths the time series, which is beneficial for detecting gradual regression but harmful for detecting sudden changes. Moreover, the long-term detection algorithm does not use the went-away detector, as it already focuses on long-term patterns.

5.4.6 Cost-shift Detector. Subroutine-level metrics help detect small regressions by reducing variance but may cause false positives due to cost shifts from code refactoring, such as moving code from one subroutine to another. The cost-shift detector utilizes the concept of *cost domains* to help filter out such false positives. A cost domain is a group of subroutines within which a cost shift is likely to occur. FBDetect provides several default detectors for common cost domains. For instance, one detector analyzes stack traces to find upstream callers of a subroutine and treats them as a cost domain. Another treats all subroutines within the same class as a cost domain. Additionally, a detector uses user-defined metadata to group subroutines with the same metadata prefix, while another considers endpoints with matching name prefixes. A further detector groups all subroutines modified by a code commit. Finally, FBDetect allows developers to create custom detectors for specific cost domains.

Just Accepted

Given a regression and its associated cost domain, the cost-shift detector performs the following checks to determine whether a regression is caused by a cost shift:

- If the domain does not exist before the regression, e.g., a new class, the regression is not a cost shift within the domain.
- If the domain's cost is significantly larger than the regression's cost change, we exclude the domain from the cost-shift detector. For instance, when examining a domain with a 20% CPU cost to investigate a 0.005% CPU regression, the domain's seasonal pattern alone could obscure the regression's effect. This also means FBDetect should avoid using a very large cost domain (e.g., a cost domain to include all subroutines) when possible.
- If the domain's cost change is negligible compared to the regression's cost change, we consider the regression a cost shift within the domain. For example, if a class's method X's cost increases significantly while the total cost of all the class's methods hardly changes, it is likely that the cost just shifts from another method Y in the class to method X.

The cost-shift detector runs between the deduplication steps, SOMDedup and PairwiseDedup, which will be explained next. This execution order prioritizes faster algorithms like SOMDedup to filter out regressions early, minimizing the computational load in the later, slower steps.

5.4.7 SOMDedup. FBDetect deduplicates regressions caused by the same code or configuration change. For instance, a regressed subroutine may trigger regressions in all its upstream callers. Additionally, a single change might impact various metrics, such as CPU usage and throughput.

While classic clustering algorithms can deduplicate n regressions by comparing each pair with a complexity $O(n^2)$, Self-Organizing Maps (SOM) [74] offer a more scalable solution, with a complexity of $O(n)$. To reduce running time, FBDetect employs a two-step approach for regression deduplication. First, *SOMDedup* uses SOM to deduplicate metrics of the same type (e.g., different subroutines' gCPUs) within the same analysis window, often reducing regressions by two orders of magnitude. For example, it addresses cases where multiple subroutines call the same regressed subroutine. The remaining regressions are then processed by *PairwiseDedup*, which applies a pairwise-comparison clustering algorithm to further merge regressions across different metrics (e.g., gCPU and throughput) and time windows.

SOMDedup is optimized for speed. It uses SOM to map high-dimensional features into a lower-dimensional space and merge nearby items into a cluster. Each regression is represented as an item, and the features used for clustering include typical time-series metrics like Fourier frequencies, variance, and change points, along with several distinguishing features we have introduced.

One distinguishing feature is candidate root causes. Finding the exact root cause of a regression is the ultimate goal of FBDetect; therefore, the exact root cause is unknown at this step. However, with the information already available, FBDetect can narrow down the potential root causes and use them as a feature. Specifically, FBDetect finds a list of potential root causes for a regression by searching for changes that modify the regressed subroutine and are introduced right before the regression starts. FBDetect encodes this list as a bitmap feature, where each bit represents whether a change may be the root cause of the regression.

Another distinguishing feature is the metric ID, a concatenation of the subroutine name and metric name. Regressions with similar metric IDs are likely to share the same root cause. To avoid the scalability issues of pairwise comparisons, we convert metric IDs into integers using TF-IDF [130] with 2- and 3-gram lengths.

After grouping related regressions using SOM, within each group, FBDetect presents the regression with the highest ImportanceScore to developers as the representative.

$$\begin{aligned}\text{ImportanceScore} = \; & w_1 \times \text{RelativeCostChange} + \\ & w_2 \times \text{AbsoluteCostChange} + \\ & w_3 \times (1 - \text{PopularityScore}) + \\ & w_4 \times \text{PotentialRootCauseFound}\end{aligned}$$

Just Accepted

Here w_i are tunable weights with default values: w_1=0.2, w_2=0.6, w_3=0.1, w_4=0.1. RelativeCostChange and AbsoluteCostChange represent the magnitude of change in the regression; we aim to select a representative regression with significant changes. PopularityScore indicates the probability of the regressed subroutine appearing in a random stack trace sample; we aim to avoid widely invoked subroutines. PotentialRootCauseFound is a boolean indicating whether any potential root causes are found; we prefer regressions with known root causes.

Discussion of alternatives. To identify a scalable clustering algorithm, we considered several alternatives, including K-Nearest Neighbors (KNN) [34] and hierarchical clustering [62]. Ultimately, we chose SOM due to its robustness in setting hyperparameters. Each algorithm has hyperparameters that significantly impact its effectiveness, such as the number of clusters in KNN, the cut-level in hierarchical clustering, and the grid size in SOM.

For KNN and hierarchical clustering, automatically setting these hyperparameters to be robust across diverse workloads proved challenging. Determining the number of clusters (K) in KNN beforehand is impractical due to the varying number of regressions, and iterating over different K values is computationally expensive. Similarly, the cut-level in hierarchical clustering depends on the data distribution. We attempted to automate cut-level selection by testing different values and evaluating their Silhouette scores [120], which measure clustering quality. However, we found that these scores often do not converge to an optimal value.

In contrast, SOM's hyperparameter can be set in a robust way. Using a grid size of $L \times L$, where $L = \lceil \sqrt[4]{n} \rceil$ and n is the number of regressions, consistently yields good results across diverse workloads. Therefore, we chose SOM.

5.4.8 PairwiseDedup. The second pass of regression deduplication, PairwiseDedup, is optimized for quality by maximizing deduplication. While SOMDedup focuses on deduplicating regressions within the same analysis window and with the same type of metrics, PairwiseDedup aims to deduplicate regressions across different analysis windows and with different types of metrics such as gCPU and throughput.

PairwiseDedup takes as input a list of representative regressions newly identified by SOMDedup and filtered by cost-shift analysis, along with a list of past representative regressions already grouped by prior rounds of PairwiseDedup execution. It compares each new regression with existing groups, merging it into the most similar group if the similarity is above a threshold or creating a new group otherwise. While PairwiseDedup offers higher accuracy than SOMDedup, its scalability is limited by pairwise comparisons.

PairwiseDedup computes similarity scores for a set of features between a new source regression and a target group. It then applies user-defined rules based on these scores to determine whether the source should be merged into the target. Users can define the metrics to consider for merge (e.g., gCPU and throughput), the similarity threshold for each feature, and how to combine multiple features to make the final decision. For example, the merge may require all or a subset of feature scores to exceed certain thresholds. If the source can be merged into multiple targets, we choose the one with the highest aggregate feature scores.

Below are some examples of the most useful features for which we compute similarity scores:

- *Pearson time series correlation coefficient* [15]. We compute the coefficient between the source regression and each regression in the target group, and use the maximal value.
- *Text cosine similarity* [126]. We compute the similarity between the metric ID of the source regression and the metric ID of each regression in the target group, and use the maximal value.
- *Stack-trace overlap*. Since multiple subroutines may appear in one stack-trace sample, the sample can be used to calculate gCPU for all these subroutines. The stack-trace-overlap feature measures the percentage of shared samples used for calculating two subroutines' gCPU. If the target group consists of multiple time series, we use the union of their stack traces to compare with those of the source.

Just Accepted

Stack-trace samples	gCPU before regression	gCPU after regression
A->B->C	0.01	0.02
B->E->F	0.02	0.03
D->B->C	0.02	0.02
B->E->D	0.04	0.06
G->B->D	Does not exist	0.01
Total	0.09	0.14

Table 4. Example of gCPU changes involving subroutine *B*.

```
def fun1()
    ......
    if conf.get("new_feature_A") == True:
        code of new feature A
    ......
```

Fig. 15. Example code with the gate mechanism.

5.4.9 Root Cause Analysis. We define the root cause of a regression as the specific code or configuration change causing it. For example, tens of thousands of developers write code on FrontFaaS, leading to hundreds or even thousands of changes per release. As a result, identifying the root cause can be challenging.

To identify the root cause of a regression, FBDetect generates a set of candidates by examining code or configuration changes deployed immediately before the regression occurred. It then ranks these candidates based on a set of weighted factors that measure the candidates' relevance to the regression. Finally, developers are presented with these ranked candidates to guide their investigation.

Below are the commonly used factors that measure the relevance between a candidate change and a regression. FBDetect computes a weighted sum of these factors as the relevance value.

Subroutine gCPU. For services using stack-trace sampling, this factor measures the fraction of the reported regression in gCPU that can be attributed to the subroutines affected by a code change. We illustrate this using an example. Suppose a regression in subroutine *B*'s gCPU is detected. Table 4 lists the stack-trace samples that contain *B*, where *A* to *G* are subroutines and *A->B* means *A* invokes *B*. The gCPU of *B* is 0.09 before the regression and 0.14 after the regression, as shown in the last row of Table 4. Therefore, *B*'s regression magnitude is $\mathcal{R}$=0.14-0.09=0.05.

Suppose a code change modifies subroutines *A* and *E*. The stack-trace samples involving *A* or *E* are *A->B->C*, *B->E->F*, and *B->E->D*. The gCPU before the regression for these three samples is 0.01+0.02+0.04=0.07. The gCPU after the regression for these three samples is 0.02+0.03+0.06=0.11. Therefore, among B's samples, those involving *A* and *E* cause a regression magnitude of $\mathcal{L}$=0.11-0.07=0.04. The fraction of the regression in *B* that can be attributed to this code change (i.e., *A* and *E*) is $\mathcal{L}/\mathcal{R}$=0.04/0.05=80%. A higher value indicates the change is more likely to be the root cause.

This approach requires FBDetect to be able to know which subroutines a code change has modified. For standard code changes, FBDetect uses Meta's diff comparison tools. However, the *gate* mechanism presented in Section 2 has introduced additional challenges. As a review, with the gate mechanism, the developer first submits a code change like the one in Figure 15.

Just Accepted

Execution sequence of FBDetect techniques	FrontFaaS		PythonFaaS	AdServing	
	Short-term regression	Long-term regression	Short-term regression	Short-term regression	Long-term regression
# Change points detected (§5.4.2 and §5.4.5)	3.96M	1.09K	324.85K	239.67K	1.9K
After went-away detection (§5.4.3)	10.85K (1/365)	——	3.98K (1/82)	5.09K (1/47)	——
After seasonality detection (§5.4.4)	8.46K (1/468)	——	2.93K (1/111)	4.13K (1/58)	——
After threshold filtering (Table 3)	5.15K (1/769)	1.06K (1/1.03)	1.1K (1/295)	1.45K (1/165)	1.85K (1/1.03)
After SameRegressionMerger	4.6K (1/861)	114 (1/9.6)	650 (1/500)	750 (1/320)	275 (1/7)
After SOMDedup (§5.4.7)	1.12K (1/3536)	39 (1/28)	183 (1/1775)	309 (1/776)	104 (1/18)
After cost-shift analysis (§5.4.6)	650 (1/6092)	15 (1/73)	81 (1/4010)	309 (1/776)	104 (1/18)
After PairwiseDedup (§5.4.8)	210 (1/18857)	12 (1/91)	62 (1/5240)	203 (1/1180)	95 (1/20)
Total	582 (1/7779)				

Table 5. Effectiveness of individual techniques in filtering out spurious change points. All numbers in parenthesis are relative ratios to those in the first row. PythonFaaS skips long-term regression detection, while AdServing skips cost-shift analysis. SameRegressionMerger deduplicates the same regression that shows up in multiple overlapping analysis windows.

When the change is applied, the configuration "new_feature_A" will be set to false so that the new feature is not activated. Later, the developer will submit a config change, setting "new_feature_A" to be true, activating the new feature. The gate mechanism presents additional challenges to FBDetect, because the regression is triggered by the config change, but FBDetect cannot apply diff comparison to a config change as the config change does not modify any source code directly.

To mitigate the gap, FBDetect tries to build a mapping between the name of a gate config and the subroutines it affects, by searching for subroutines reading the corresponding config parameter. However, compared to the code above, there exist more complex patterns in which the config parameter is read in a wrapper function, which requires inter-procedural code analysis. Accurate inter-procedural code analysis is known to be hard, so we take a pragmatic approach: We analyzed the wrapper patterns and found most are simple and include only one layer of wrapping. Then we only search for those common patterns during code analysis. For the remaining ones, the developers can manually create the mapping.

Text similarity. Text similarity can help identify the root cause. For example, suppose FBDetect detects a regression in subroutine *foo* and cannot find code changes that directly modify it. However, there may be another code change with a description like "loosening constraints for *foo*." We can use this information to rank that change higher than others. Concretely, FBDetect computes text similarity between a regression and a code change by tokenizing both into feature vectors. The regression vector is based on timing, metric name, metric type, stack traces (if available), and other factors. The code change vector is based on the descriptive title, summary, file name, change content, and more. FBDetect measures relevance as the cosine similarity between these feature vectors.

Time series correlation. A service may record a time series that does not directly reflect its performance but indicates a certain setup of the service, such as which algorithm is being used for processing requests. Since such metrics do not represent performance, FBDetect cannot directly run regression detection on them. However, if the time series of such a metric strongly correlates with a regression, it may indicate a root cause. We compute the Pearson correlation between the time series for such a metric and the time series for a found regression. The higher the correlation coefficient, the more likely the change caused the regression.

5.5 Evaluation

Our evaluation answers the following questions:

Just Accepted

- Among many techniques included in FBDetect, what is the breakdown of each technique's contribution to filtering out spurious performance anomalies?
- How many false positives and false negatives are reported by FBDetect?
- What is the accuracy of FBDetect' root cause analysis?
- Does FBDetect indeed catch regressions as small as 0.005%?
- How does FBDetect compare with prior art?
- Does PyPerf's stack-trace sampling add high overhead?

5.5.1 Contribution of FBDetect's Individual Techniques. Due to high noise levels in production environments, service performance metrics frequently exhibit anomalies, many of which are false positives. This section demonstrates how each of FBDetect's techniques reduces the number of performance anomalies requiring developers' attention.

Table 5 shows, over one month for several workloads, the number of remaining performance anomalies after being filtered by FBDetect's different techniques in sequence. We use FrontFaaS as an example to demonstrate how to read the table. Running the short-term regression detection algorithm for FrontFaaS detects 3.96 million change points. The value of "*1/3536*" at the intersection of the row "*after SOMDedup*" and the column "*FrontFaaS short-term regression*" means that, after executing all the steps between "*went-away detection*" and "*SOMDedup*," the number of remaining performance anomalies is filtered down to $\frac{1}{3536}$ of the original 3.96 million change points detected.

As shown in Table 5, FBDetect reduces the number of performance anomalies that developers need to investigate by three to four orders of magnitude, greatly boosting productivity. Among the techniques, the went-away detector is the most effective, filtering out 99.7% of detected change points. The seasonality detector further removes 22% of the regressions output by the went-away detector. Additionally, SOMDedup filters out 72%, cost-shift analysis filters out 34%, and PairwiseDedup filters out 49% of their respective input regressions. Overall, short-term change points are more prevalent and noisier than long-term ones, necessitating more aggressive filtering.

5.5.2 False Positive and False Negative. Evaluating FBDetect's false positives (i.e., reporting regressions when they do not actually exist) and false negatives (i.e., real regressions missed by FBDetect) faces several challenges. First, it is difficult to rely on tens of thousands of developers consistently to perform manual classification. Moreover, sometimes the ground truth is unknown. For example, if a true 0.005% regression is missed by FBDetect, it is less likely to be caught by developers as well, so it will remain unknown. Despite these challenges, we use different data to corroborate the evaluation, focusing on FrontFaaS, because it involves tens of thousands of developers and its manually tagged data is more complete than other services.

False negative. Given FBDetect's aggressive filtering of performance anomalies by three to four orders of magnitude (Table 5), false negatives could be a concern, as many true regressions could be mistakenly filtered out. To evaluate the false negatives of FBDetect, we use FrontFaaS's performance-related incidents in production as the ground truth and evaluate how many of them should have been caught by FBDetect. This ground truth, though not theoretically complete, serves as a high bar, as FrontFaaS is closely monitored by a dedicated team, and all of its incidents are rigorously recorded and reviewed.

For the entire year of 2023, only four performance-related incidents were recorded for FrontFaaS, despite its highly dynamic and incident-prone environment (§3)—thousands of code commits daily and automated code deployment every three hours. This low incident count is not because FrontFaaS rarely has performance regressions; each year, FBDetect catches regressions in FrontFaaS that, if left unchecked, would waste more than half a million servers.

Among the four performance incidents, two were detected by FBDetect but were not acted upon by developers in time. The third incident occurred because the developer did not configure the regressed metric to be exported

Just Accepted

to FBDetect, and the last one was due to a capacity-management issue unrelated to code or configuration changes. Overall, for all of FrontFaaS's performance incidents in 2023, FBDetect did not miss any that it was supposed to catch. However, this does not imply that FBDetect never misses regressions larger than 0.005% for FrontFaaS; it only means that the missed regressions are not significant enough to be noticed by developers.

We further searched all site incidents in the past three years (not limited to FrontFaaS) and found a single one caused by FBDetect's false negative. In this incident, FBDetect's cost-shift detector mistakenly filtered out a regression and caused a large service to experience a 0.015% error rate. Since then, we have improved the detector's algorithm and tuned its threshold.

False positive. Surprisingly, false positives are not a major concern for FBDetect. Over a one-month evaluation period, FBDetect reported 217 regressions for FrontFaaS. At this rate, and with tens of thousands of developers writing code for FrontFaaS (and thousands of code commits daily), a FrontFaaS developer, on average, will be assigned a ticket by FBDetect to investigate a regression only once every four years. This ideal situation is thanks to FBDetect's capability of aggressively filtering performance anomalies by three to four orders of magnitude before reporting them to developers.

Among the 217 regressions reported by FBDetect, developers explicitly confirmed either true regressions or false positives for only 70 of them, marked another 123 as RESOLVED, and did not act on the remaining 24.

Among the 70 confirmed cases, 49 are true regressions and 21 are false positives. The 21 false positives include 1 duplicate regression that was not merged, 15 cost shifts that were not filtered out, 1 temporary spike missed by the went-away detector, and 4 miscellaneous cases. Given that 15 of the 21 false positives are due to cost shifts, this will be a focus of future research.

For the 123 regressions that are marked as RESOLVED by developers, unfortunately, there is no ground truth regarding whether they are true regressions or false positives. Data suggests that many of them are likely true regressions. For example, for 24 of them, FBDetect's regression report provided root causes, and developers confirmed the accuracy of those root causes, even though those regressions are still marked as RESOLVED instead of true regressions. The authors of the paper, not the FrontFaaS developers, manually investigated some RESOLVED cases and found that many of them match well with the same magnitudes and similar timings of regressions recorded by Meta's canary-test tool, which is a strong indicator that they are true regressions caused by code changes. However, since we lack explicit confirmation from developers, we still consider the ground truth unknown.

Some of the RESOLVED cases simply went away without any data indicating how they were fixed. This suggests that FBDetect's went-away detector might be able to filter them out if longer extended windows were used (Figure 11). However, this would delay the timeliness of regression detection. This trade-off is challenging and requires future research.

LLM-assisted auto classification. As discussed above, reducing false positives is one of the primary goals of FBDetect, and for this purpose, it is important to classify regressions reported by FBDetect, so that we can understand the false positive rate and the main sources of false positives. Due to the difficulty of getting developers' feedback and motivated by the success of large language models (LLMs), the FBDetect team has developed *AutoLabeler*, which uses LLM to read developers' discussions and comments to classify a reported regression. To be concrete, AutoLabeler first uses LlaMa-3.1-8b model with the following prompt:

Just Accepted

	Ground truth	AutoLabeler TP	AutoLabeler FP	Precision	Recall
Cost Shift	432	270	8	97%	63%
Fixed	363	213	49	81%	59%
Will Not Fix	352	194	47	80%	55%

Table 6. The result of using AutoLabeler to classify regression tasks. For each row, which represents a category, "Ground truth" is the total number of tasks classified by human developers as this category; "AutoLabeler TP" is the number of true positives marked by AutoLabeler and "AutoLabeler FP" is the number of false positives marked by AutoLabeler. Precision = $\frac{TP}{TP+FP}$; Recall = $\frac{TP}{TP+FN} = \frac{TP}{GroundTruth}$.

You are asked to determine the status of a regression task among the following possible statuses: Cost Shift, Expected, Duplicate, Will Not Fix, Fixed, False Positive, Cannot Determine.

Choose one of the above statuses based on the following comments that are in chronological order. Each comment is wrapped by double quotes. Choose the status that is most likely:

<Task comments wrapped with double quotes>

Just answer by the status only.

Just Accepted

In the second step, AutoLabeler further refines the output of the LLM. In particular, if the LLM cannot determine the category of a regression, AutoLabeler uses additional information to determine whether the regression is fixed (FIXED), went away by itself (RESOLVED), or needs further investigation (REVIEW). Specifically, 1) If a regression task was later associated with a revert of the change or a revised change, then it is classified as FIXED. However, note that a revised change is tagged by a developer manually, and such tagging may be incomplete, so this rule still cannot cover all FIXED cases. 2) If LLM outputs "Cannot Determine" for a regression task, and the regression finally went away, it is classified as RESOLVED; if the regression did not go away, it is classified as REVIEW, which means it needs further investigation. Note that REVIEW is the default tag when a regression task is created. 3) AutoLabeler merges "Expected" and "Will Not Fix" into one category WILL NOT FIX, since for the purpose of FBDetect, they essentially indicate the same thing, but we separate them in LLM prompt for better accuracy.

We use AutoLabeler to classify 1278 regression tasks with the REVIEW tag. AutoLabeler reports 62 COST SHIFT cases, 6 DUPLICATE cases, 16 FALSE POSITIVE cases, 190 FIXED cases, 109 WILL NOT FIX cases, 516 RESOLVED cases, and 379 REVIEW cases. Compared to human labeling, AutoLabeler reduced the number of REVIEW cases by 70%, from 1278 to 379.

To validate the accuracy of AutoLabeler, we run AutoLabeler on a group of regression tasks with clear developers' tagging. We focus on COST SHIFT, FIXED, and WILL NOT FIX, since these three categories dominate the result. Though RESOLVED and REVIEW categories also have large numbers, they lack clear ground truth and thus we do not include them in the comparison. For example, a RESOLVED case could be because a developer submitted a fix but did not tag the task as FIXED. In this case, the case will be closed after a certain period and since the regression is gone, it will be automatically tagged as RESOLVED. However, its true ground truth should be FIXED.

Table 6 shows the result. As shown in this table, comparing the "AutoLabeler TP" and "AutoLabeler FP" columns, AutoLabeler achieves a good precision. Comparing the "AutoLabeler TP" and "Ground truth" columns, AutoLabeler can cover a majority of the cases but still misses a good number of the cases. While imperfect, AutoLabeler is helpful to reduce developers' tagging effort and to illustrate the general trend of false positives.

5.5.3 Root Cause Analysis. In this section, we evaluate the accuracy of FBDetect's root cause analysis for FrontFaaS. Given a regression, FBDetect suggests root-cause candidates only if its confidence in the recommendation

is sufficiently high; otherwise, it will not suggest any root cause. Out of the 217 regressions reported by FBDetect over a one-month period, FBDetect suggested root causes for 75 of them. Of these, 71 were confirmed correct by developers, meaning the real root cause was among the top-three change candidates suggested by FBDetect.

Among the 142 regressions for which FBDetect did not suggest root causes, developers manually root-caused an additional six. While the success rate of FBDetect's root cause analysis seems mediocre (75/217=35%), even developers' manual efforts could only marginally improve it to (75+6)/217=37%. This indicates that root cause analysis in complex production environments is challenging. Despite the limitations, FBDetect still significantly improves developer productivity, as the majority of successful root cause analysis is automated by FBDetect, with developers contributing only an additional 2pp. Moreover, in most cases, FBDetect's behavior of not pinpointing a single root cause is actually appropriate, as explained below.

To understand why FBDetect does not pinpoint root causes for certain regressions, we manually investigated 61 regressions marked as true regressions by developers but not root-caused by FBDetect. Note that, to find a sufficient number of such cases, these 61 cases are beyond the time period covered by Table 5. Below is a breakdown of these 61 cases based on why they were not root-caused by FBDetect:

- 28 cases do not have a clear root cause. The most common reason is that the regression is caused by a new feature release, which is expected to consume more resources. Moreover, the new feature involves many code changes, and no single change dominates the regression. Therefore, FBDetect's behavior of not pinpointing a single root cause is appropriate.
- 5 cases are caused by failures in production. As they are not caused by code or configuration changes, it is appropriate for FBDetect not to report root causes for them.
- 11 cases are caused by changes not exported to FBDetect. Therefore, it is appropriate for FBDetect not to report root causes for them. Future enhancements require FBDetect to be integrated with a broader set of change sources.
- 5 cases have their root causes reported by FBDetect, but they are not ranked among the top three candidates.
- 12 cases have their root causes considered by FBDetect, but they are filtered out because their relevance scores fall below certain thresholds.

After analyzing these cases, we conclude that the success rate of FBDetect's root cause analysis is significantly higher than it initially appears (35%). Only the last two categories represent true failures of FBDetect's algorithm. Assuming the two periods we investigated (the period reporting 217 regressions and the period reporting 61 regressions not root caused by FBDetect) have the same failure rate, and excluding the 11 cases caused by changes not exported to FBDetect, we estimate the true failure rate of FBDetect's root cause analysis to be only (1-35%)*(5+12)/(61-11)=22%.

5.5.4 Catching Small Regressions. FBDetect is effective at catching small regressions, as shown in Table 7, where "*All*" means all regressions reported by FBDetect, and "*TR*" and "*FP*" mean true regressions and false positives explicitly confirmed by developers. The smallest true regression is indeed 0.005% as expected, while the largest is 3.9%. The difference in distributions between true regressions and all regressions is minor, whereas their difference with respect to false positives is more pronounced.

One might expect that for the tiny regressions between 0.005% and 0.01%, FBDetect's false positive rate would be higher as they are more likely to be caused by noise. However, the data shows that the false positive rate is not higher for tiny regressions, because the P10 for all regressions, true regressions, and false positives are nearly identical. Interestingly, the reported largest regressions tend to be false positives. As discussed in §5.5.2, false positives are mostly cost shifts, indicating an area for future research.

5.5.5 Comparison with Yahoo's EGADS. We compare FBDetect with Yahoo's EGADS [79], which offers multiple anomaly detection algorithms. The test data consists of a random set of 107 time series where FBDetect reported

Just Accepted

	Smallest	P10	P50	P90	P99	Largest
All	0.005%	0.010%	0.043%	0.232%	0.948%	15.094%
TR	0.005%	0.011%	0.048%	0.241%	0.809%	3.862%
FP	0.006%	0.012%	0.062%	0.442%	4.003%	15.094%

Table 7. Magnitude of detected regressions (TR=true regressions; FP=false positives). The 0.01% cell at the intersection of the "*All*" row and the "*P10*" column means that the 10th percentile of all regressions detected by FBDetect is 0.010%.

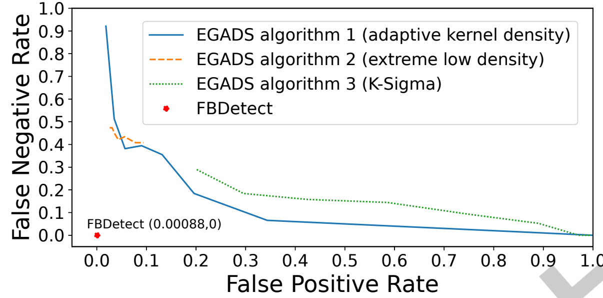


Fig. 16. EGADS's algorithms cannot simultaneously reduce both false negatives and false positives.

regressions and around 35K time series where FBDetect reported no regressions. Manual analysis of the 107 positive cases reveals 76 true positives and 31 false positives. We report the false positive rate (i.e., the fraction of the "35K+31" true negatives classified as positives) and the false negative rate (i.e., the fraction of the 76 true positives classified as negatives).

FBDetect's false positive rate is 31/(35K+31)=0.00088. Since FBDetect has almost no false negatives based on the results in §5.5.2, we assume its false negative rate is zero. FBDetect's false-positive and false-negative rates are shown in Figure 16.

The EGADS algorithms have sensitivity parameters that can be tuned to reduce either false positives or false negatives, but not both. We tune these parameters and show the tradeoff in Figure 16. For a fair comparison, EGADS uses the same historical time window as FBDetect but combines FBDetect's analysis and extended windows as EGADS's analysis window. EGADS struggles with transient issues like the one in Figure 1(c) because using a large threshold to filter them would miss small regressions, while using a small threshold would incorrectly flag many of them as regressions.

Given 76 true positive cases in the test data, if an algorithm reports 760 or more positive cases for developers to investigate manually, over 90% of these investigations would be futile, eroding developer's trust in the algoritm. Thus, we require a false positive rate below $760/(35K+31) \approx 0.02$. Among the EGADS algorithms in Figure 16, only "EGADS algorithm 1" can meet this false positive rate, but at the cost of a 0.84 false negative rate, meaning it would miss 84% of true regressions. This shows that EGADS algorithms are ineffective in simultaneously achieving both low false positives and low false negatives. In contrast, with the help of the went-away detector, FBDetect catches nearly all small regressions without introducing many false positives.

5.5.6 PyPerf Profiling Overhead. To measure PyPerf's overhead in collecting stack traces for Python programs, we created a CPU-intensive micro-benchmark that repeatedly serializes a large data structure, compresses it, and writes it to a file. We compare its throughput with and without PyPerf.

PyPerf's sampling rate for PythonFaaS is one sample per server every 30 minutes. At this rate, we do not observe any noticeable overhead on the micro-benchmark. To understand the worst-case scenario, we configured PyPerf to collect one sample per server per second. This is the highest rate used in production and is only applied

Just Accepted

to the smallest services that run on just a few servers to collect sufficient samples. At this sampling rate, PyPerf reduces the throughput of the micro-benchmark by about 0.8%, which is rather moderate. Moreover, note that the overhead for collecting stack traces for Python programs is higher than for C/C++ programs.

Our measurement of tail latency shows the same trend. A high sampling rate can significantly increase the p99.9 latency of the micro-benchmark. For example, a sampling rate of about 2,000 samples per second can increase the p99.9 latency by 14%. However, a sampling rate lower than one sample per second does not show a noticeable impact. This is reasonable, because lower sampling rate means fewer requests getting sampled and/or fewer samples per request.

5.5.7 Overlapping between ServiceLab and FBDetect. For a regression found by FBDetect, a natural question is whether it has been reported by ServiceLab. For many cases, the answer is no, either due to the service not adopting ServiceLab yet or due to the false negatives of ServiceLab, which has been analyzed in Section 4.5.1. However, we also find that, for a number of cases, the answer is yes, which brings up the question why the change is not prevented by ServiceLab.

We analyzed 486 overlapping cases between ServiceLab and FBDetect. This means that both systems reported a regression on the same code change. For 482 of them, we found that the users ignored the reports from ServiceLab and continued to deployment, and the regressions were later detected by FBDetect. Our further analysis shows that 262 out of the 482 cases were finally marked as true regressions, which means that the users wrongly ignored the warnings from ServiceLab. The remaining 220 cases, however, were false positives, due to reasons like cost shift, which mean the users were right to ignore the warnings. Such data highlights the importance of minimizing false positives. Otherwise, the users may not fully trust the tool and ignore the warnings, even when some of them indicate true problems. Finally, in 4 cases, after ServiceLab reported a regression, the user marked it as "partially fixed" and then FBDetect reported the regression again, which is reasonable as the regression is not fully fixed and thus is expected.

6 Monitor ML Training with ServiceLab and FBDetect

The success of ML applications has resulted in ML training becoming the fastest-growing datacenter workload. The unique characteristics of training jobs have introduced challenges to both ServiceLab and FBDetect.

To create a training program, a programmer usually just needs to create a high-level description of the ML model, including the number of neurons per layer, the forward function, etc. Then the ML framework (e.g., PyTorch) will compile the high-level description into an executable that can run on GPUs or other accelerators [42]. The compilation may create tensors to store the data necessary for training, select kernels for computation, and determine how to parallelize the computation. PyTorch further applies just-in-time (JIT) compilation to recompile the program based on the information collected at run time.

Obviously, the ML framework plays an important role in the performance of ML training, and it has introduced several challenges to ServiceLab and FBDetect. For ServiceLab, due to the high cost of training and the scarcity of GPUs, it is usually not possible to run multiple trials of A/B tests. This can be particularly troublesome when testing a change to the ML framework, as the regression may only show up for a certain type of model, and testing every model with the change is often too expensive. For FBDetect, root cause analysis becomes harder. Recall that FBDetect depends on stack-trace sampling for root cause analysis: If a subroutine shows increased CPU utilization, then FBDetect searches for code or config changes that may affect the subroutine. A change to the ML framework, however, may cause it to compile an ML model in a different way, such as using a different kernel or parallelization strategy. The change may also cause the compilation itself to take longer time, maybe for some specific models. Such regressions usually do not show up as a clear CPU or GPU increase in one or a few subroutines, but may cause a change in the high-level workflow.

Just Accepted

To address these challenges, we have made two efforts. First, we studied a number of regressions related to model compilation, and proposed several metrics to complement FBDetect's existing metrics. Second, we have built a continuous testing pipeline, by collaborating ServiceLab and FBDetect, to identify regressions and their root causes.

6.1 Identifying Appropriate Metrics for Model Compilation

A buggy change to model compilation may cause two types of problems: First, the compiled model may experience suboptimal performance during training. This is addressed in Section 6.2. Second, compilation itself may crash or take longer time, which will also affect the overall training performance. In Meta, some models are re-trained frequently (in the order of minutes) [94] and thus have strict SLOs for compilation time as well. This section focuses on this type.

We investigated 16 regressions in PyTorch compilation that happened within a period of two months. We find most of them were reported by the users rather than by the monitoring tools like FBDetect. This indicates that we probably lack appropriate metrics to monitor this type of regressions. Our investigation shows three main categories of symptoms, which can be captured by adding appropriate metrics:

- Rank divergence. In distributed training involving N GPUs, those GPUs are labeled from rank 0 to rank N-1. We find a number of regressions are caused by one rank taking significantly more time to compile than other ranks. Our further investigation shows two root causes. First, PyTorch uses caching to improve compilation efficiency. For certain ranks within a training job, we can save compilation time if a previously compiled kernel is in the cache and matches with the rank to be compiled. Therefore, any bugs in the caching related code may cause a regression. Second, PyTorch applies the JIT idea to recompile ranks at run time, and inappropriate triggering of recompilation may also cause one rank to have longer compilation time than others.
- Compiler crash. A bug in PyTorch can cause the compiler to crash, and of course the training job will get stuck as a result. Providing such information to FBDetect is helpful as a stuck job can be caused by various reasons, and thus it is helpful to know the root cause is in the compiler.
- Long compilation time. A bug can also cause a significant slowdown to the compilation. For example, in one regression, some compiler passes raise exceptions from C++ to Python, triggering massive slowdown due to exception handling.

Based on this study, we added the following metrics: 1) To capture long compilation time, we compute the total compilation time for each model, including the time of all attempts and all ranks. 2) To capture rank divergence, for each model, we compute the sum of compilation time per rank and report the maximal one. 3) To further understand the root causes of rank divergence, we report the number of recompiles, the number of cache cold starts, and the number of cache warm starts. 4) We report the number of compiler failures. In addition, we also monitor GPU utilization and job stuck rate, which is an indicator of the underlying problems.

6.2 Continuous Benchmarking to Monitor Training Performance

Identifying and root causing performance regressions in ML training is hard due to numerous changes originating from various orthogonal dimensions. These dimensions encompass compiler level changes, modifications to model architecture, training configurations, model configurations, and communication protocols (e.g., NCCL). As explained above, many of these changes cannot be root caused by FBDetect's root cause analysis method. Additionaly, running multiple trials of A/B tests with ServiceLab is costly.

To address this problem, we have introduced a new testing method by collaborating ServiceLab and FBDetect. In this method, we leverage ServiceLab to run benchmarks continuously, possibly on different versions of models, PyTorch, and configurations. We then combine results from multiple runs into a time-series data and feed it into

Just Accepted

FBDetect. When FBDetect identifies a regression, it selects a number of suspected changes and provides them to ServiceLab. ServiceLab then determines which one causes the regression by performing a binary search among these suspected changes with rigorous multi-trial tests.

The benefit of this method is two-fold: First, assuming most changes do not cause a regression, tests on multiple such no-regression changes naturally provide multiple data points for statistical analysis, which allows us to avoid to run multi-trial tests for every change. Instead, we only run multi-trial tests on a subset of those suspected changes, which significantly reduces the cost of testing. Second, FBDetect is released from the task of root cause analysis, which is particularly challenging for ML training. Instead, FBDetect only needs to suggest a number of suspected changes, and then ServiceLab takes over to determine which one is the real cause.

Benchmarks. For PyTorch, we use the TorchBench [52] benchmark suite, which is designed to test different aspects of PyTorch. Other teams have started to build their own benchmarks, usually by simplifying their production models into smaller ones and using smaller training sets.

Identifying suspected changes. For regressions with sudden increases or drops, FBDetect selects the change closest to the point where the regression occurred. For gradual regressions, FBDetect uses the following heuristics to choose the starting point and the end point of suspected changes: In the window before the regression, it chooses the first change whose metric value falls below 75 percentile of all metric values as the starting point. In the window after the regression, it chooses the last change whose metric value stays above 25 percentile of all metric values as the end point. It reports all changes between these points as suspected to ServiceLab. Typically, FBDetect can report 5,000 to 20,000 changes as suspected for one gradual change.

6.3 Experience

We started these two efforts since the middle of 2024, and since then, they have helped us capture regressions that could have cost Meta tens of thousands of machines. To give a few examples:

In one incident, a buggy change in PyTorch caused CUDA Memory to not be recycled after the model is deleted. The reason is because the CUDA Generator has longer lifetime than the model, which means we cannot rely on model deletion to release the CUDA Generator. The author amended the change to add a reference count on the CUDA Generator to recycle its consumed GPU memory.

In another incident, a change in PyTorch led to a certain model regressing 5% in latency and 9.9% in GPU memory. This change introduces a huge "foreach_copy" operator, which is an unreasonably long kernel in the CUDAGraph run.

Both examples are related to memory consumption. FBDetect alone can detect such regressions but cannot identify their root causes, since they do not show up as increased subroutine CPU utilization. In the past, to diagnose such issues, we have relied on additional memory profiling tools. Now, with continuous benchmarking, FBDetect can identify the problem and then ServiceLab can apply a binary search to identify the root cause.

In another example, FBDetect reported 11,769 regressions as suspected, and then ServiceLab took 4.5 hours to run 25 tests in total (some tests were run in parallel) to identify the exact root cause. Again, without this new approach, FBDetect would have to identify the root cause among those 11,769 regressions alone, which could be challenging for certain kinds of bugs.

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 [17, 37, 44, 51, 113, 132, 158], tuning configuration parameters [83, 90, 150, 164], or changing hardware [93, 149, 163]. 2) Some works try

Just Accepted

to detect machines that are significantly slower than others [40, 48, 57, 58, 105], 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 [53, 61, 93].

The closest work to ServiceLab is the study by Maricq et al. on performance variance in CloudLab [93]. 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. Performance regression can be detected before production [68, 93, 161] or during production [9, 47, 80, 147].

Synthetic benchmark [33, 96, 116, 131, 144] and record-and-replay [1–3] are two primary methods for pre-production performance evaluation. ServiceLab supports both but primarily uses record-and-replay due to its high fidelity in testing real applications.

Treadmill [161] and TailBench [68] overcome several common pitfalls of performance testing frameworks with synthetic traffic, allowing them to precisely measure at microsecond-scale. Lancet [73] 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 [111].

Performance data from Google's gmail [13] 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 [147] and WSMeter [80] 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 [41], OS-level [16], and library-level [49]).

Time Series Analysis. Anomaly detection in time series identifies data points or short periods that deviate significantly from expected behavior. It emphasizes isolated deviations over persistent shifts. A recent survey [125] categorizes anomaly detection methods into six families. Forecasting-based approaches model normal behavior by predicting future values from a sliding window of past observations and then flag any significant deviation between forecasted and observed values as anomalies [5, 22, 91, 99]. Reconstruction methods learn a compressed representation of normal subsequences, and use the difference between the original and reconstructed signals as the anomaly score [106, 123]. Encoding methods bypass explicit reconstruction by mapping subsequences directly into a low-dimensional latent space, and identify anomalies by measuring deviations of encoded representations from a model of normality [21, 71, 128, 151]. Distance-based methods quantify abnormality by measuring a point's or subsequence's dissimilarity to other points in the time series [24, 115, 157]. Distribution methods estimate the underlying probability distribution of the data or its subsequences and detect anomalies by fitting models to identify points in the extreme tails [46, 85, 142]. Isolation tree methods use ensembles of random trees to isolate observations via recursive partitioning of the data space [30, 54, 87].

Change-point detection identifies moments when the statistical properties of a sequence—including mean, variance, or distribution—change significantly. Compared to anomaly detection, it targets persistent shifts rather than isolated deviations. Popular change-point detection methods fall into three categories: parametric, non-parametric, and probabilistic. Parametric methods assume a model for the data in each segment and detect

Just Accepted

changes in its parameters [69, 103, 104, 136]. Non-parametric change-point detection aims to identify a broad range of changes with minimal assumptions about the data distribution. These methods measure dissimilarity between data segments using divergence, distance, or rank-based statistics [65, 66, 77, 88, 95, 135, 135]. Chen et al. demonstrated that graph theory can facilitate change-point detection (CPD) [28]. Probabilistic change-point detection methods include a Bayesian framework for detecting change points online [6, 121].

In time series data, seasonality refers to regular and predictable patterns that recur at specific intervals within a year (e.g., daily, monthly, or quarterly) [4]. A common approach to handling seasonality is time series decomposition, which separates a series into trend, seasonal, and irregular components [32]. The seasonal component can be estimated by averaging or smoothing observations from equivalent periods (e.g., all January values) after removing the trend. Another method incorporates seasonality into autoregressive models by adding seasonal differencing, autoregressive, and moving-average terms to capture patterns recurring every s observations (e.g., every 12 months for annual seasonality) [153]. Recently, machine learning-based models have emerged to capture seasonal patterns, combining the interpretability of classical methods with the flexibility of data-driven learning [139].

FBDetect relies on change-point detection to detect persistent shifts and seasonality detection to filter changes caused by seasonality. The key challenge FBDetect faces is that some transient anomalies can last for a while so that existing change-point detection algorithms will report them as persistent change points, and tuning sensitivity parameters will hurt their capability to capture tiny but persistent changes. FBDetect incorporates the went-away detector to filter such transient anomalies.

Stack trace sampling. Existing tools can collect stack traces for services written in C/C++ [108], Java [56, 60, 109], Python [18, 31, 43, 45, 72, 107, 117–119, 143], Go [35], Ruby [67], etc.

For a program that is compiled into machine code (e.g., those written by C/C++), its call stack is organized in a standard format, including multiple frames corresponding to subroutines in the call stack and a return address in each frame to identify the code to call the subroutine. For such a program, a profiler can follow the standard format to walk through the call stack. For a program that is written in an interpreted language (e.g., Java, Python, etc), its source code is interpreted and executed by an "interpreter" (e.g., Java Virtual Machine) at run time. Profiling the interpreter results in retrieving the interpreter's call stack, instead of the program's call stack. For these programs, we need additional help, often from the interpreter [45, 109], to map the interpreter's call stack to the program's call stack. To our knowledge, PyPerf is the first profiler capable of deriving a precise end-to-end stack trace across a Python program and the C/C++ libraries it invokes.

Root cause analysis. Prior works have attempted to localize a bug to lines of code [63], files [76, 78, 98, 101, 114, 122, 154, 156, 159], commits [19, 152, 155], or deployments [84]. They also rely on stack traces and text similarity to find the corresponding locations. Recently, several works have leveraged large language models for the same purpose [8, 29, 160]. Many of these works require a user-written bug report or at least an error message to start with. FBDetect does not require such information. Furthermore, none have tried to find the root cause of a tiny performance regression.

8 Conclusion

We have presented ServiceLab and FBDetect, Meta's pre-production and in-production performance regression detection systems. They help our developers capture tiny regressions, as small as 0.005%, in a noisy environment. They save millions of servers each year and reduce the number of performance anomalies requiring developers' attention by three to four orders of magnitude. They have also shown great potential in the new application domain of ML training, despite the new challenges introduced by these applications. Such success is due to a collective effort of novel statistical analysis, low-overhead approach to collect a large number of data points at

Just Accepted

scale, and novel algorithms to filter false positives and perform root cause analysis. In the future, we plan to continue to reduce the false positive rate as well as providing better support for new application domains.

References

[1] 2019. Google Cluster Workload Traces 2019. https://research.google/resources/datasets/google-cluster-workload-traces-2019/.

[2] 2022. Google Workload Traces 2022. https://research.google/resources/datasets/google-workload-traces-2022/.

[3] 2024. Azure Public Dataset. https://github.com/Azure/AzurePublicDataset.

[4] 2024. Seasonality. https://en.wikipedia.org/w/index.php?title=Seasonality&oldid=1244363987#cite_note-1 Page Version ID: 1244363987.

[5] Adam Aboode. 2018. *Anomaly Detection in Time Series Data Based on Holt-Winters Method*. https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-226344

[6] Ryan Prescott Adams and David J. C. MacKay. 2007. Bayesian Online Changepoint Detection. doi:10.48550/arXiv.0710.3742 arXiv:0710.3742 [stat].

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

[8] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending Root-Cause and Mitigation Steps for Cloud Incidents Using Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1737–1749. doi:10.1109/ICSE48619.2023.00149

[9] Amazon CloudWatch - Creating a Canary 2024. https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch_Synthetics_Canaries_Create.html.

[10] Samaneh Aminikhanghahi and Diane J. Cook. 2016. A Survey of Methods for Time Series Change Point Detection. *Knowledge and Information Systems* 51, 2 (8 Sept. 2016), 339–367. doi:10.1007/s10115-016-0987-z

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

[12] Approximations for Mean and Variance of a Ratio 2024. https://www.stat.cmu.edu/%7Ehseltman/files/ratio.pdf.

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

[14] Michele Basseville, Igor V Nikiforov, and others. 1993. *Detection of Abrupt Changes: Theory and Application*. Vol. 104. prentice Hall Englewood Cliffs.

[15] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson Correlation Coefficient. In *Noise Reduction in Speech Processing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–4. doi:10.1007/978-3-642-00296-0_5

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

[17] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. 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)*. USENIX Association, Carlsbad, CA, 195–212. https://www.usenix.org/conference/osdi18/presentation/berger

[18] Emery D. Berger, Sam Stern, and Juan Altmayer Pizzorno. 2023. Triangulating Python Performance Issues with SCALENE. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*. USENIX Association, 51–64. https://www.usenix.org/conference/osdi23/presentation/berger

[19] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. 2018. Orca: Differential Bug Localization in Large-Scale Services. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 493–509. https://www.usenix.org/conference/osdi18/presentation/bhagwan

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

[21] Paul Boniol and Themis Palpanas. 2022. Series2graph: Graph-based subsequence anomaly detection for time series. *arXiv preprint arXiv:2207.12208* (2022).

[22] George Edward Pelham Box and Gwilym M. Jenkins. 1994. *Time Series Analysis: Forecasting and Control* (3rd ed.). Prentice Hall PTR, USA.

[23] G. E. P. Box and David A. Pierce. 1970. Distribution of Residual Autocorrelations in Autoregressive-Integrated Moving Average Time Series Models. *J. Amer. Statist. Assoc.* 65, 332 (1970), 1509–1526. http://www.jstor.org/stable/2284333

[24] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. 2000. LOF: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 93–104.

[25] 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. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (San Jose,

Just Accepted

CA) *(USENIX ATC'13)*. USENIX Association, USA, 49–60.
[26] 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. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 49–60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson
[27] Nicolas Brousse. 2019. 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*. 1–4.
[28] Hao Chen and Nancy Zhang. 2015. Graph-Based Change-Point Detection. *The Annals of Statistics* 43, 1 (2015), 139–176. https://www.jstor.org/stable/43556511 Publisher: Institute of Mathematical Statistics.
[29] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. Automatic Root Cause Analysis via Large Language Models for Cloud Incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 674–688. doi:10.1145/3627703.3629553
[30] Zhangyu Cheng, Chengming Zou, and Jianwei Dong. 2019. Outlier detection using isolation forest and local outlier factor. In *Proceedings of the conference on research in adaptive and convergent systems*. 161–168.
[31] Sümer Cip. 2024. Yappi: Yet Another Python Profiler. https://github.com/sumerc/yappi.
[32] Robert B Cleveland, William S Cleveland, Jean E McRae, and Irma Terpenning. 1990. STL: A Seasonal-Trend Decomposition. *Journal of Official Statistics* 6, 1 (1990), 3–73.
[33] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. ACM, New York, NY, USA, 143–154. doi:10.1145/1807128.1807152
[34] Thomas Cover and Peter Hart. 1967. Nearest Neighbor Pattern Classification. *IEEE transactions on information theory* 13, 1 (1967), 21–27. doi:10.1109/TIT.1967.1053964
[35] Russ Cox and Shenghou Ma. 2013. Profiling Go Programs. https://go.dev/blog/pprof.
[36] David Daly. 2021. Creating a Virtuous Cycle in Performance Testing at MongoDB. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 33–41.
[37] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*.
[38] Wilfrid J Dixon and Frank J Massey Jr. 1957. *Introduction to Statistical Analysis*. McGraw-Hill.
[39] Wilfrid J Dixon and Kareb K Yuen. 1974. Trimming and winsorization: A review. *Statistische Hefte* 15, 2-3 (1974), 157–170.
[40] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. 2013. Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*. Association for Computing Machinery, New York, NY, USA, Article 14, 14 pages. doi:10.1145/2523616.2523627
[41] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 211–224.
[42] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, Adam Hutchin, Utku Diril, Krishnakumar Nair, Ehsan K. Aredestani, Martin Schatz, Yuchen Hao, Rakesh Komuravelli, Kunming Ho, Sameer Abu Asal, Joe Shajrawi, Kevin Quinn, Nagesh Sreedhara, Pankaj Kansal, Willie Wei, Dheepak Jayaraman, Linda Cheng, Pritam Chopda, Eric Wang, Ajay Bikumandla, Arun Karthik Sengottuvel, Krishna Thottempudi, Ashwin Narasimha, Brian Dodds, Cao Gao, Jiyuan Zhang, Mohammed Al-Sanabani, Ana Zehtabioskuie, Jordan Fix, Hangchen Yu, Richard Li, Kaustubh Gondkar, Jack Montgomery, Mike Tsai, Saritha Dwarakapuram, Sanjay Desai, Nili Avidan, Poorvaja Ramani, Karthik Narayanan, Ajit Mathews, Sethu Gopal, Maxim Naumov, Vijay Rao, Krishna Noru, Harikrishna Reddy, Prahlad Venkatapuram, and Alexis Bjorlin. 2023. MTIA: First Generation Silicon Targeting Meta's Recommendation Systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 80, 13 pages. doi:10.1145/3579371.3589348
[43] Ben Frederickson. 2024. py-spy: Sampling profiler for Python programs. https://github.com/benfred/py-spy.
[44] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried
[45] Pablo Galindo. 2022. Python Support for the Linux perf Profiler. "https://docs.python.org/3.12/howto/perf_profiling.html.
[46] Markus Goldstein and Andreas Dengel. 2012. Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm. *KI-2012: poster and demo track* 1 (2012), 59–63.
[47] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, and Chunqiang Tang. 2023. Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 325–342. https://www.

Just Accepted

usenix.org/conference/osdi23/presentation/grubic

[48] 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. 2018. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 1–14. https://www.usenix.org/conference/fast18/presentation/gunawi

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

[50] Nishant Gupta, Iyswarya Narayanan, Shivam Handa, Sayak Chakraborti, Pankit Thapar, Baohua Shan, Ariel Rao, Yuanlai Liu, Pengyuan Wang, Yuqing Wu, Qingyi Gao, Chris Chao-Chun Cheng, Sihan You, Louis Huang, Jingyuan Fan, Kenny Yu, Kevin Lin, Tengfei Mu, Parth Malani, Haiying Wang, Trey Lu, and Peter Zhang. 2024. Dynamic Idle Resource Leasing To Safely Oversubscribe Capacity At Meta. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 792–810. doi:10.1145/3698038.3698537

[51] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. 2017. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 168–183. doi:10.1145/3132747.3132774

[52] Yueming Hao, Xu Zhao, Bin Bao, David Berard, Will Constable, Adnan Aziz, and Xu Liu. 2023. TorchBench: Benchmarking PyTorch with High API Surface Coverage. arXiv:2304.14226 [cs.LG] https://arxiv.org/abs/2304.14226

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

[54] Sahand Hariri, Matias Carrasco Kind, and Robert J Brunner. 2019. Extended isolation forest. *IEEE transactions on knowledge and data engineering* 33, 4 (2019), 1479–1489.

[55] HHVM. [n. d.]. https://hhvm.com.

[56] HPROF: A Heap/CPU Profiling Tool 2024. https://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html.

[57] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and Enhancing In Situ System Observability for Failure Detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, Carlsbad, CA, 1–16. https://www.usenix.org/conference/osdi18/presentation/huang

[58] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. ACM, New York, NY, USA, 150–155. doi:10.1145/3102980.3103005

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

[60] IBM Whole-System Analysis Tool (WAIT) 2015. https://publib.boulder.ibm.com/httpserv/cookbook/Major_Tools-IBM_WholeSystem_Analysis_Tool_WAIT.html.

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

[62] Stephen C Johnson. 1967. Hierarchical Clustering Schemes. *Psychometrika* 32, 3 (1967), 241–254.

[63] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Florida) *(ICSE '02)*. Association for Computing Machinery, New York, NY, USA, 467–477. doi:10.1145/581339.581397

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

[65] Takafumi Kanamori, Is Nagoya-U Ac Jp, Shohei Hido, Masashi Sugiyama, and Cs Titech Ac Jp. [n. d.]. A Least-squares Approach to Direct Importance Estimation. ([n. d.]).

[66] Takafumi Kanamori, Taiji Suzuki, and Masashi Sugiyama. 2012. Statistical analysis of kernel-based least-squares density-ratio estimation. *Machine Learning* 86, 3 (March 2012), 335–367. doi:10.1007/s10994-011-5266-3

[67] Aman Gupta Karmani. 2024. stackprof - a sampling call-stack profiler for Ruby 2.1+. https://github.com/tmm1/stackprof.

[68] Harshad Kasture and Daniel Sanchez. 2016. Tailbench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. doi:10.1109/IISWC.2016.7581261

[69] Yoshinobu Kawahara, Takehisa Yairi, and Kazuo Machida. 2007. Change-Point Detection in Time-Series Data Based on Subspace Identification. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. IEEE, 559–564. doi:10.1109/ICDM.2007.78

[70] Maurice George Kendall. 1948. *Rank Correlation Methods*. Griffin.

Just Accepted

[71] Eamonn Keogh, Stefano Lonardi, and Bill'Yuan-chi' Chiu. 2002. Finding surprising patterns in a time series database in linear time and space. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 550–556.

[72] Robert Kern. 2024. `line_profiler`: Line-by-line profiling for Python. https://github.com/pyutils/line_profiler.

[73] Marios Kogias, Stephen Mallon, and Edouard Bugnion. 2019. Lancet: A self-correcting Latency Measuring Tool. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 881–896. https://www.usenix.org/conference/atc19/presentation/kogias-lancet

[74] T. Kohonen. 1990. The self-organizing map. *Proc. IEEE* 78, 9 (1990), 1464–1480. doi:10.1109/5.58325

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

[76] Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Kui Liu, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2019. D&C: A Divide-and-Conquer Approach to IR-based Bug Localization. (2019). arXiv:1902.02703 [cs.SE] https://arxiv.org/abs/1902.02703

[77] S. Kullback and R. A. Leibler. 1951. On Information and Sufficiency. *The Annals of Mathematical Statistics* 22, 1 (1951), 79–86. https://www.jstor.org/stable/2236703 Publisher: Institute of Mathematical Statistics.

[78] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2015. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 476–481. doi:10.1109/ASE.2015.73

[79] Nikolay Laptev, Saeed Amizadeh, and Ian Flint. 2015. Generic and Scalable Framework for Automated Time-series Anomaly Detection. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Sydney, NSW, Australia) *(KDD '15)*. Association for Computing Machinery, New York, NY, USA, 1939–1947. doi:10.1145/2783258.2788611

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

[81] 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. 2021. Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. 553–569.

[82] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. 2013. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology* 49, 4 (2013), 764–766. doi:10.1016/j.jesp.2013.03.013

[83] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. ACM, New York, NY, USA, Article 9, 14 pages. doi:10.1145/2670979.2670988

[84] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. 2020. Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 389–402. https://www.usenix.org/conference/nsdi20/presentation/li

[85] Zheng Li, Yue Zhao, Nicola Botta, Cezar Ionescu, and Xiyang Hu. 2020. Copod: copula-based outlier detection. In *2020 IEEE international conference on data mining (ICDM)*. IEEE, 1118–1123.

[86] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. 2003. A Symbolic Representation of Time Series, with Implications for Streaming Algorithms. In *Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery* (San Diego, California) *(DMKD '03)*. Association for Computing Machinery, New York, NY, USA, 2–11. doi:10.1145/882082.882086

[87] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *2008 eighth ieee international conference on data mining*. IEEE, 413–422.

[88] Song Liu, Makoto Yamada, Nigel Collier, and Masashi Sugiyama. 2013. Change-Point Detection in Time-Series Data by Relative Density-Ratio Estimation. *Neural Networks* 43 (July 2013), 72–83. doi:10.1016/j.neunet.2013.01.012

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

[90] Ashraf Mahgoub, Paul Wood, Alexander Medoff, Subrata Mitra, Folker Meyer, Somali Chaterji, and Saurabh Bagchi. 2019. SOPHIA: Online Reconfiguration of Clustered NoSQL Databases for Time-Varying Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 223–240. https://www.usenix.org/conference/atc19/presentation/mahgoub

[91] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, Puneet Agarwal, et al. 2015. Long short term memory networks for anomaly detection in time series. In *Proceedings*, Vol. 89. 94.

[92] Henry B. Mann. 1945. Nonparametric Tests Against Trend. *Econometrica* 13, 3 (1945), 245–259. http://www.jstor.org/stable/1907187

Just Accepted

[93] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. 2018. Taming Performance Variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 409–425. https://www.usenix.org/conference/osdi18/presentation/maricq

[94] Kiran Kumar Matam, Hani Ramezani, Fan Wang, Zeliang Chen, Yue Dong, Maomao Ding, Zhiwei Zhao, Zhengyu Zhang, Ellie Wen, and Assaf Eisenman. 2024. QuickUpdate: a Real-Time Personalization System for Large-Scale Recommendation Models. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 731–744. https://www.usenix.org/conference/nsdi24/presentation/matam

[95] David S. Matteson and Nicholas A. James. 2014. A Nonparametric Approach for Multiple Change Point Analysis of Multivariate Data. *J. Amer. Statist. Assoc.* 109, 505 (Jan. 2014), 334–345. doi:10.1080/01621459.2013.849605 Publisher: ASA Website _eprint: https://doi.org/10.1080/01621459.2013.849605.

[96] 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. 2020. MLPerf Training Benchmark. arXiv:1910.01500 [cs.LG]

[97] T.K. Moon. 1996. The Expectation-Maximization Algorithm. *IEEE Signal Processing Magazine* 13, 6 (1996), 47–60. doi:10.1109/79.543975

[98] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 151–160. doi:10.1109/ICSME.2014.37

[99] Mohsin Munir, Shoaib Ahmed Siddiqui, Andreas Dengel, and Sheraz Ahmed. 2018. DeepAnT: A deep learning approach for unsupervised anomaly detection in time series. *Ieee Access* 7 (2018), 1991–2005.

[100] 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. 2021. RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*.

[101] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2011. A Topic-Based Approach for Narrowing the Search Space of Buggy Files from a Bug Report. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, USA, 263–272. doi:10.1109/ASE.2011.6100062

[102] Guilherme Ottoni. 2018. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 151–165.

[103] E. S. Page. 1954. Continuous Inspection Schemes. *Biometrika* 41, 1/2 (1954), 100–115. jstor:2333009 doi:10.2307/2333009

[104] E. S. Page. 1955. A Test for a Change in a Parameter Occurring at an Unknown Point. *Biometrika* 42, 3/4 (1955), 523–527. jstor:2333401 doi:10.2307/2333401

[105] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi. 2019. IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 47–62. https://www.usenix.org/conference/atc19/presentation/panda

[106] Daehyung Park, Yuuna Hoshi, and Charles C Kemp. 2018. A multimodal anomaly detector for robot-assisted feeding using an lstm-based variational autoencoder. *IEEE Robotics and Automation Letters* 3, 3 (2018), 1544–1551.

[107] Vincent Pelletier. 2024. `pprofile`: Line-granularity, thread-aware deterministic and statistic pure-Python profiler. https://github.com/vpelletier/pprofile.

[108] perf 2024. https://perf.wiki.kernel.org/index.php/Main_Page.

[109] Perf-Map-Agent: A Java Agent to Generate Method Mappings to Use with the Linux 'perf' Tool 2024. https://github.com/jvm-profiling-tools/perf-map-agent.

[110] PHP Profiler Xenon 2024. https://github.com/facebook/hhvm/wiki/Profiling.

[111] Mia Primorac, Edouard Bugnion, and Katerina Argyraki. 2017. How to Measure the Killer Microsecond. *SIGCOMM Comput. Commun. Rev.* 47, 5 (oct 2017), 61–66. doi:10.1145/3155055.3155065

[112] Python 3.12 Preview: Support For the Linux perf Profiler 2024. https://realpython.com/python312-perf-profiler/.

[113] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 145–160. https://www.usenix.org/conference/osdi18/presentation/qin

[114] Mohammad Masudur Rahman and Chanchal K. Roy. 2018. Improving IR-Based Bug Localization with Context-Aware Query Reformulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 621–632. doi:10.1145/3236024.3236065

Just Accepted

[115] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. 2000. Efficient algorithms for mining outliers from large data sets. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 427–438.

[116] 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. 2020. MLPerf Inference Benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 446–459. doi:10.1109/ISCA45697.2020.00045

[117] Joe Rickerby. 2024. `pyinstrument`: Call stack profiler for Python. https://github.com/joerick/pyinstrument.

[118] Brett Rosen and Ted Czotter. 2024. The Python Profilers (`cProfile`). https://docs.python.org/3.8/library/profile.html.

[119] Jim Roskind. 2024. The Python Profilers (`profile`). https://docs.python.org/3.8/library/profile.html.

[120] Peter J. Rousseeuw. 1987. Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis. *J. Comput. Appl. Math.* 20 (1987), 53–65. doi:10.1016/0377-0427(87)90125-7

[121] Yunus Saatçi, Ryan Turner, and Carl Edward Rasmussen. 2010. Gaussian process change point models. In *Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10)*. Omnipress, Madison, WI, USA, 927–934.

[122] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving Bug Localization Using Structured Information Retrieval. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Silicon Valley, CA, USA) *(ASE '13)*. IEEE Press, 345–355. doi:10.1109/ASE.2013.6693093

[123] Mayu Sakurada and Takehisa Yairi. 2014. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In *Proceedings of the MLSDA 2014 2nd workshop on machine learning for sensory data analysis*. 4–11.

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

[125] Sebastian Schmidl, Phillip Wenig, and Thorsten Papenbrock. 2022. Anomaly detection in time series: a comprehensive evaluation. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1779–1797.

[126] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. 2008. *Introduction to Information Retrieval*. Vol. 39. Cambridge University Press Cambridge.

[127] 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. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 498–514.

[128] Pavel Senin, Jessica Lin, Xing Wang, Tim Oates, Sunil Gandhi, Arnold P Boedihardjo, Crystal Chen, and Susan Frankenstein. 2015. Time series anomaly discovery with grammar-based compression.. In *Edbt*. 481–492.

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

[130] Karen Sparck Jones. 1988. *A statistical interpretation of term specificity and its application in retrieval.* Taylor Graham Publishing, GBR, 132–142.

[131] spec [n. d.]. Standard Performance Evaluation Corporation. https://www.spec.org/.

[132] Akshitha Sriraman and Thomas F. Wenisch. 2018. uTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 177–194. https://www.usenix.org/conference/osdi18/presentation/sriraman

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

[134] Standard score 2025. https://en.wikipedia.org/wiki/Standard_score.

[135] Masashi Sugiyama, Taiji Suzuki, and Takafumi Kanamori. 2012. *Density Ratio Estimation in Machine Learning* (1st ed.). Cambridge University Press, USA.

[136] J. Takeuchi and K. Yamanishi. 2006. A unifying framework for detecting outliers and change points from time series. *IEEE Transactions on Knowledge and Data Engineering* 18, 4 (April 2006), 482–492. doi:10.1109/TKDE.2006.1599387 Conference Name: IEEE Transactions on Knowledge and Data Engineering.

[137] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 328–343. doi:10.1145/2815400.2815401

[138] 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. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Just Accepted

USENIX Association, 787–803.

[139] Sean J. Taylor and Benjamin Letham. 2017. *Forecasting at scale.* Technical Report e3190v2. PeerJ Inc. doi:10.7287/peerj.preprints.3190v2 ISSN: 2167-9843.

[140] The performance of Python with ‘perf‘ support is not great, and is going to get a lot worse 2024. https://discuss.python.org/t/the-performance-of-python-with-perf-support-is-not-great-and-is-going-to-get-a-lot-worse/25280.

[141] Theil–Sen Estimator 2024. https://en.wikipedia.org/wiki/Theil-Sen_estimator.

[142] Markus Thill, Wolfgang Konen, and Thomas Bäck. 2017. Time series anomaly detection with discrete wavelet transforms and maximum likelihood estimation. In *Intern. Conference on Time Series (ITISE)*, Vol. 2. 11–23.

[143] Gabriele N. Tornetta. 2024. `austin`: A Frame Stack Sampler for cPython. https://github.com/P403n1x87/austin.

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

[145] Charles Truong, Laurent Oudre, and Nicolas Vayatis. 2020. Selective Review of Offline Change Point Detection Methods. *Signal Processing* 167 (2020), 107299. doi:10.1016/j.sigpro.2019.107299

[146] Aad W Van der Vaart. 2000. *Asymptotic statistics.* Vol. 3. Cambridge university press.

[147] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. 2016. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16).* USENIX Association, Savannah, GA, 635–651. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/veeraraghavan

[148] Jeffrey S. Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (mar 1985), 37–57. doi:10.1145/3147.3165

[149] Guosai Wang, Lifei Zhang, and Wei Xu. 2017. 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).* IEEE, 25–36.

[150] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. 2018. Understanding and Auto-Adjusting Performance-Sensitive Configurations. *SIGPLAN Not.* 53, 2 (March 2018), 154–168. doi:10.1145/3296957.3173206

[151] Li Wei, Nitin Kumar, Venkata Nishanth Lolla, Eamonn J Keogh, Stefano Lonardi, and Chotirat (Ann) Ratanamahatana. 2005. Assumption-Free Anomaly Detection in Time Series.. In *SSDBM*, Vol. 5. 237–242.

[152] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating Bugs from Software Changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE ’16).* Association for Computing Machinery, New York, NY, USA, 262–273. doi:10.1145/2970276.2970359

[153] Billy M Williams and Lester A Hoel. 2003. Modeling and forecasting vehicular traffic flow as a seasonal ARIMA process: Theoretical basis and empirical results. *Journal of transportation engineering* 129, 6 (2003), 664–672.

[154] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution.* 181–190. doi:10.1109/ICSME.2014.40

[155] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. ChangeLocator: Locate Crash-Inducing Changes Based on Crash Reports. In *Proceedings of the 40th International Conference on Software Engineering* (, Gothenburg, Sweden,) *(ICSE ’18).* Association for Computing Machinery, New York, NY, USA, 536. doi:10.1145/3180155.3182516

[156] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. CrashLocator: Locating Crashing Faults Based on Crash Stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014).* Association for Computing Machinery, New York, NY, USA, 204–214. doi:10.1145/2610384.2610386

[157] Takehisa Yairi, Yoshikiyo Kato, and Koichi Hori. 2001. Fault detection by mining association rules from house-keeping data. In *proceedings of the 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, Vol. 18. Citeseer, 21.

[158] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. 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).* USENIX Association, Santa Clara, CA, 15–28. https://www.usenix.org/conference/fast17/technical-sessions/presentation/yan

[159] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to Rank Relevant Files for Bug Reports Using Domain Knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014).* Association for Computing Machinery, New York, NY, USA, 689–699. doi:10.1145/2635868.2635874

[160] Dylan Zhang, Xuchao Zhang, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2024. LM-PACE: Confidence Estimation by Large Language Models for Effective Root Causing of Cloud Incidents. 11 pages. doi:10.1145/3663529.3663858

[161] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. 2016. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA ’16).* IEEE Press, 456–468. doi:10.1109/ISCA.2016.47

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

Just Accepted

[163] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. 2018. wPerf: Generic Off-CPU Analysis to Identify Bottleneck Waiting Events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 527–543. https://www.usenix.org/conference/osdi18/presentation/zhou

[164] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 338–350. doi:10.1145/3127479.3128605

A Appendix

A.1 Leveraging the Law of Large Numbers

The Law of Large Numbers (LLN) [38] states that given a random variable x with a finite mean μ and variance σ^2, as the sample size n approaches infinity, the sample mean converges to μ. Let $Variance(\overline{x})$ denote the variance of $\overline{x}$. We have:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \quad \text{and} \quad \lim_{n\to\infty} \overline{x} = \mu \tag{3}$$

$$Variance(\overline{x}) = \sigma^2/n \quad \text{and} \quad \lim_{n\to\infty} Variance(\overline{x}) = 0. \tag{4}$$

Suppose we introduce a code change and want to determine whether its impact on a performance metric is statistically significant by comparing two groups of samples collected before and after the change. According to the LLN, as the sample size increases, we can more accurately infer the two groups' means μ_1 and μ_2. Thus, even a small difference between μ_1 and μ_2 can be detected, as reflected in Expression 1, where $\lim_{n\to\infty} \Delta_{\text{threshold}}=0$. The LLN can also explain why the noise in Figures 8 and 9 decreases as the number of servers (m) increases.

Although the LLN explains why minor regressions are detectable with a large number of samples, it is crucial to recognize that both the LLN and Expression 1 assume stationary random variables and do not account for non-stationary, transient issues like the one in Figure 1(c). Therefore, production realities are more challenging.

A.2 Calculating the Detection Threshold

While the Law of Large Numbers provides valuable intuition, a direct analysis of FBDetect would be ideal. However, its complexity renders precise analysis impractical. As an alternative, we examine a simpler yet representative problem, as described below, to gain meaningful insights.

Suppose we introduce a code change and want to determine its impact on a performance metric. Let the two groups of samples collected before and after the code change have population means μ_1 and μ_2, population variances σ_1^2 and σ_2^2, sample sizes n_1 and n_2, sample means $\overline{x_1}$ and $\overline{x_2}$, and sample variances s_1^2 and s_2^2, respectively. For simplicity, assume the two groups have identical population variances ($\sigma_1^2 = \sigma_2^2 = \sigma^2$) and identical sample variances ($s_1^2 = s_2^2 = s^2$).

We use Student's t-test [38] to examine two hypotheses:

- H0: $\mu_1 = \mu_2$ (i.e., there is no performance difference).
- H1: $\mu_1 \neq \mu_2$ (i.e., there is a performance difference).

To reject H0, the t-statistic must exceed a threshold:

$$t = \frac{\overline{x_1} - \overline{x_2}}{s\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \geq T_{\text{critical}}, \tag{5}$$

where T_{critical} is a threshold determined by n_1, n_2, and the required probability (e.g., 99%) of making a correct decision. In FBDetect, $n_1 \gg n_2$ because we prioritize detecting regressions quickly after a change, and thus do

Just Accepted

not have time to collect as many samples as were collected before the change. Thus, as an approximation, we can eliminate the $\frac{1}{n_1}$ term in Expression 5, which simplifies to:

$$t \approx \sqrt{n_2}(\overline{x_1} - \overline{x_2})/s \geq T_{\text{critical}}. \tag{6}$$

Reusing the symbol in Expression 1, let $\Delta_{\text{threshold}}$ be the minimal value of $\overline{x_1} - \overline{x_2}$ that satisfies Expression 6. We obtain

$$\Delta_{\text{threshold}} \approx \sqrt{s^2/n_2}\; T_{\text{critical}}. \tag{7}$$

This simplified analysis provides the basis for Expression 1. While it does not fully capture FBDetect's complexity, the relationship $\Delta_{\text{threshold}} \propto \sqrt{\sigma^2/n}$ generally applies to methods comparing two sample groups to detect changes.

A.3 Subroutine-level Measurements using gCPU

We demonstrate that, similar to Expression 2, subroutine-level measurements using the gCPU metric also reduce variance, enabling detection of small regressions.

We first define the following notations. Let r be a subroutine in a Linux process, with random variables X_r and X_P representing the CPU usage of r and the Linux process, respectively. Let μ_r and μ_P denote their means, and σ_r^2 and σ_P^2 denote their variances. Define $\text{gCPU}_r = X_r/X_P$.

Assume that the Linux process consists of k independent subroutines like r, so $\frac{\mu_r}{\mu_P} = \frac{1}{k}$ and $\sigma_r^2 = \frac{\sigma_P^2}{k}$. Based on the approximation for the variance of a ratio [12], we have:

$$Variance(gCPU_r) = Variance(\frac{X_r}{X_P}) \tag{8}$$

$$\approx \frac{\mu_r^2}{\mu_P^2}[\frac{\sigma_r^2}{\mu_r^2} - 2\frac{Covariance(X_r, X_P)}{\mu_r \mu_P} + \frac{\sigma_P^2}{\mu_P^2}] \tag{9}$$

$$< \frac{\mu_r^2}{\mu_P^2}[\frac{\sigma_r^2}{\mu_r^2} + \frac{\sigma_P^2}{\mu_P^2}] \tag{10}$$

$$= \frac{(k+1)}{k^2} \cdot \frac{\sigma_P^2}{\mu_P^2} \tag{11}$$

$$\approx \frac{1}{k\mu_P^2}\sigma_P^2 \tag{12}$$

$$< \frac{1}{k}\sigma_P^2. \tag{13}$$

The simplification from Expression 9 to Expression 10 holds because the covariance between X_r and X_P is positive. The simplification from Expression 12 to Expression 13 assumes $\mu_P > 1$, which typically holds in a production environment. For example, a production server typically has 80 or more cores. If half of them are utilized, $\mu_P \geq 40$. Expression 13 shows that, similar to Expression 2, subroutine-level measurements using the gCPU metric also reduce variance, enabling detection of small regressions.

Next, we show that for a small subroutine, a small regression in its gCPU directly corresponds to a small regression in its absolute CPU usage. Thus, despite being a relative metric, gCPU is appropriate for regression detection.

Let μ_r^g denote the mean of gCPU_r, where $\mu_r^g = \frac{\mu_r}{\mu_P}$. Let $h\%$ denote the change in μ_r^g after μ_r increases by Δ:

$$h\% = \tilde{\mu}_r^g - \mu_r^g = \frac{\mu_r + \Delta}{\mu_P + \Delta} - \frac{\mu_r}{\mu_P} = \frac{\Delta(\mu_P - \mu_r)}{\mu_P(\mu_P + \Delta)} \approx \frac{\Delta}{\mu_P}.$$

The last step assumes that μ_r and Δ are small relative to μ_P.

Just Accepted

In summary, we have shown that, when the gCPU of a subroutine is small, which is common in production, a small $h\%$ absolute regression in gCPU approximately corresponds to an $h\%$ relative regression in the Linux process's absolute CPU usage. Thus, gCPU is appropriate for regression detection.

A.4 Resource Waste Due to Undetectable Regressions

Let W denote the aggregate fleet-wide resource waste due to undetectable regressions smaller than the detection threshold $\Delta_{\text{threshold}}$ in Expression 1. Let m denote the fleet size (i.e., the total number of servers in the fleet), and assume the number of collected samples (i.e., n in Expression 1) is proportional to m. From Expression 1, it can be derived that the resource waste as a fraction of the fleet size is given by:

$$W/m \propto \sqrt{\sigma^2/m}.$$

This indicates that the waste fraction decreases as the fleet size increases, which is positive. However, the total waste W still grows with $\sqrt{m}$:

$$W \propto \sqrt{\sigma^2 m}.$$

In conclusion, to minimize fleet-wide waste due to undetectable small regressions, it is crucial to also reduce variance (σ^2). FBDetect significantly reduces variance by measuring CPU usage at the subroutine level rather than at the overall service level, as shown in Expression 2.

Received 21 March 2025; revised 28 August 2025; accepted 17 November 2025

Just Accepted