FBDetect: Catching Tiny Performance Regressions at Hyperscale through In-Production Monitoring

Dong Young Yoon[1] Yang Wang[1†] Miao Yu[1†] Elvis Huang[1] Juan Ignacio Jones[1]
Abhinay Kukkadapu[1] Osman Kocas[1] Jonathan Wiepert[1] Kapil Goenka[1] Sherry Chen[1]
Yanjun Lin[1] Zhihui Huang[1] Jocelyn Kong[1] Michael Chow[1] Chunqiang Tang[1]

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

Abstract

This paper presents Meta's *FBDetect* system, which advances the state of the art in performance regression detection by catching regressions as small as 0.005% in noisy production environments. FBDetect monitors around 800,000 time series covering various types of metrics (e.g., throughput, latency, CPU and memory usage) to detect regressions caused by code or configuration changes in hundreds of services running on millions of servers. 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. Beyond these individual techniques, a key strength of FBDetect over prior work is its battle-tested robustness, proven by seven years of production use, and each year catching regressions that would have wasted millions of servers if left undetected.

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

Keywords: performance regression, anomaly detection.

ACM Reference Format:
Dong Young Yoon, Yang Wang, Miao Yu, Elvis Huang, Juan Ignacio Jones, Abhinay Kukkadapu, Osman Kocas, Jonathan Wiepert, Kapil Goenka, Sherry Chen, Yanjun Lin, Zhihui Huang, Jocelyn Kong, Michael Chow, and Chunqiang Tang. 2024. FBDetect: Catching Tiny Performance Regressions at Hyperscale through In-Production Monitoring. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3694715.3695977

SOSP '24, November 4–6, 2024, Austin, TX, USA

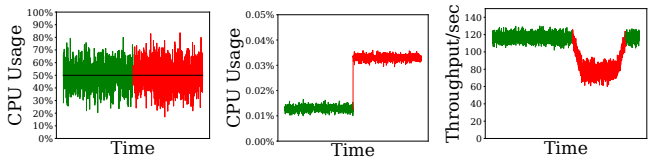
ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695977

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.

Moreover, it is essential to detect small regressions *in production*. While *pre-production* performance testing [14, 25, 67] is necessary, some issues may slip through due to the difficulty of precisely replicating complex production environments in testing. Additionally, catching tiny regressions requires a massive amount of performance data samples, which are more feasible to gather from a large production fleet than a small test environment.

Meta's *FBDetect* system performs in-production regression detection by identifying anomalies in service time-series data (e.g., CPU usage) triggered by code or configuration 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



(a) True regression but barely visible. (b) False positive due to cost shift. (c) False positive due to transient issues.

Figure 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).

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 [3] 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% 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 (§2).

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 [3] to identify anomalies, trend analysis [17] to remove seasonality, Symbolic Aggregate approXimation (SAX) [43] 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 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 [36] 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 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.

2 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 (§3). 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.[1] 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,

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

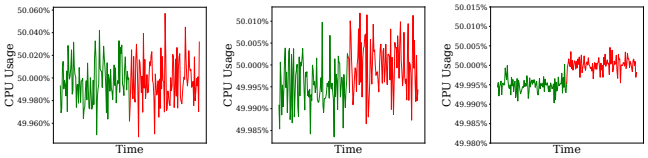
According to Expression 1, the smaller variance at the subroutine level allows detection of regressions $\frac{1}{\sqrt{k}}$ times smaller.

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.

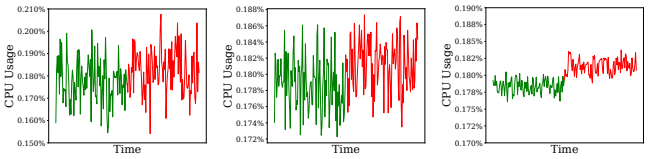
In addition to CPU metrics, FBDetect can support other subroutine-level metrics, such as latency, throughput, and error rate per RPC endpoint. However, memory and other application-level metrics require manual instrumentation if subroutine-level detection is desired.

[1] 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.



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

Figure 2. 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.

Figure 3. 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 2.

Validation via simulation. Production evaluation of our approach for detecting small regressions will be presented in §6. 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 2 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 2(c) shows that the tiny regression can be detected with sufficient samples, though sampling from 50,000,000 servers is impractical.

Figure 3 simulates subroutine-level measurements. The Linux-process-level CPU usage in Figure 2 is distributed

across k=1000 subroutines.[2] The lower variance at the subroutine level (see Expression 2) enables Figure 3 to detect the tiny regression by sampling from 1000 times fewer servers than Figure 2, 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,

3 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 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 4: (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.

FrontFaaS is Meta's serverless platform for PHP code. It runs on more than half a million servers, and tens of thousands of developers write code for it, with thousands of code commits every workday. We use the Xenon [55] profiler in the PHP runtime to sample its stack traces.

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 [30] 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

[2]The sample mean in Figure 3 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%.

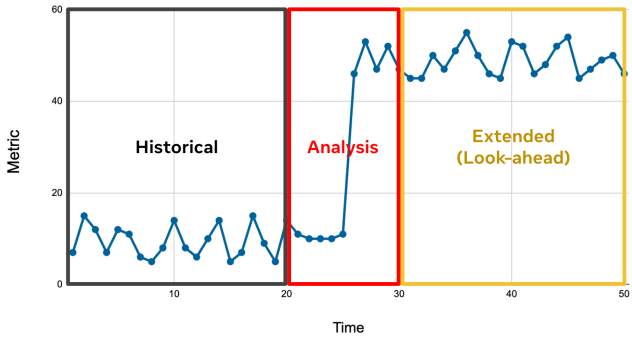


Figure 4. Time windows used in regression detection.

can annotate its stack 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 1 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.

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 [12] 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 [76] 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 1). 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 1).

Summary: The examples in Table 1 demonstrate FBDetect's ability to detect regressions across diverse workloads. Their historical, analysis, and extended windows range from hours

Name	Number of servers used	Number of servers saved yearly	Language	Leverage Stack Trace	Detection Threshold ($\Delta_{threshold}$)	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 1. 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 0.1% absolute change and a 10% relative change.

to days. These long windows allow FBDetect to gather sufficient samples to enable a small detection threshold. This highlights the importance of using 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.

4 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 [54, 55]. 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 5. 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

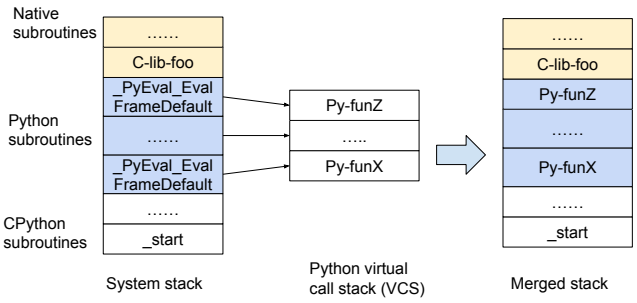


Figure 5. How PyPerf reconstructs the stack trace.

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 [8], can only approximate the time spent in C/C++ libraries since its Python-level profiling cannot reach into C/C++ code.

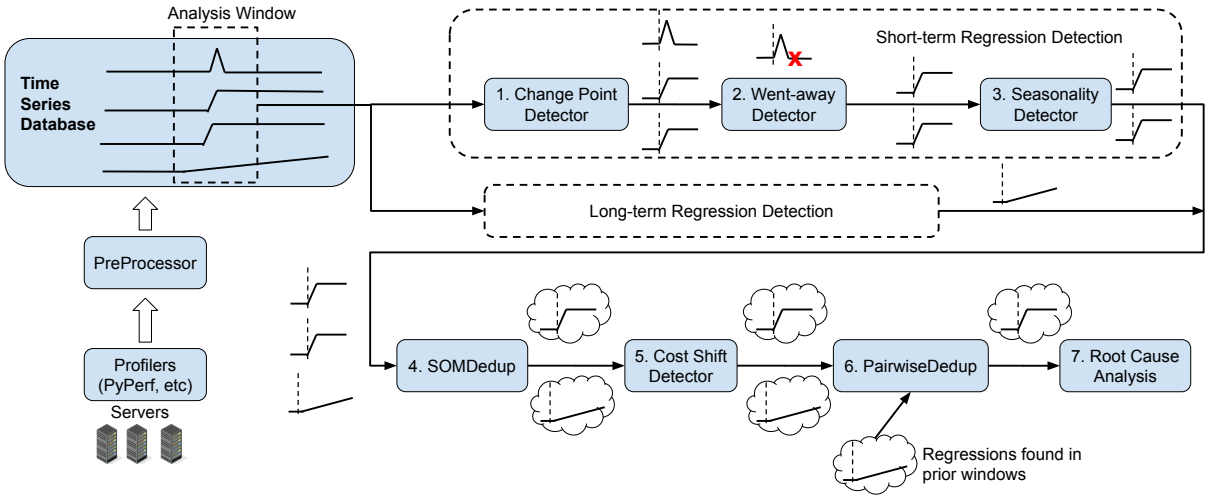


Figure 6. Workflow of FBDetect.

Although Python 3.12 introduced stack-trace collection support in October 2023 [24], 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 [56, 68], which cannot be mitigated by sampling. Second, it may interfere with Just-In-Time (JIT) optimizations, leading to further performance degradation [68].

5 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.

5.1 Overview

We follow Figure 6 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 (§4), which are then converted to subroutine-level gCPU time series. FBDetect periodically examines these time series within a recent time window to detect regressions. The "*short-term regression detection*" path in Figure 6 executes the following steps in sequence:

1. The change point detector applies change point detection [3] to identify anomalies, which are regression candidates. (§5.2.1)
2. The went-away detector filters out regressions that disappear spontaneously, like the one in Figure 1(c). (§5.2.2)
3. The seasonality detector applies trend analysis [17] to filter out regressions caused by seasonality. (§5.2.3)
4. The SOMDedup clustering algorithm performs a fast, shallow analysis to efficiently deduplicate regressions likely caused by the same change. (§5.5.1)
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. The PairWiseDedup clustering algorithm performs a slower, more thorough comparison to further deduplicate remaining regressions. (§5.5.2)
7. Finally, root cause analysis is applied to each remaining regression to pinpoint the specific code or configuration change responsible for the regression. (§5.6)

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 6.

5.2 Short-Term Regression Detection

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.

To detect short-term regressions, we use a three-step process: (1) The change-point detector identifies regression candidates. (2) The went-away detector filters out false-positive transient regressions. (3) The seasonality detector further filters out false-positive regressions caused by seasonality.

5.2.1 Change Point Detector. This detector applies the Cumulative Sum (CUSUM) [6] and Expectation Maximization (EM) [47] 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 [75] 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.

5.2.2 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 [34, 45] 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 7, 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 another change point, we consider them to be caused by different reasons. With this method, we can identify that, in Figure 7, 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) [43]. 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

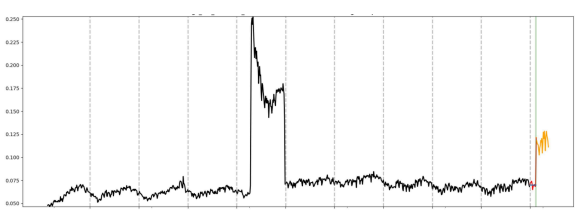


Figure 7. Catching the regression at the end.

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 [69] 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 [41] with a normality constant of 1.4826, and applies a regression coefficient (default 1.5) for sensitivity. The final regression threshold is set to $coefficient \times median \times 1.4826$.
- 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.

5.2.3 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 [17] to decompose the time series into three parts: seasonality, trend, and residual. FBDetect then removes seasonality and computes the difference between the median of the sum of trend and residual before the CUSUM change point and the median after the point. Finally, FBDetect normalizes the difference by the standard deviation of the residual and calculates the pseudo 'z-score'. 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 [10] to handle seasonality. We found that STL is superior because it is sensitive to slight changes in seasonality while being robust against sudden changes. .

5.3 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 these three steps 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 [72] 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 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.

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 subroutine, 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.
- 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 in the next section. This execution order prioritizes faster algorithms like SOMDedup to filter out regressions early, minimizing the computational load in the later, slower steps.

5.5 Regression Deduplication

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) [36] 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.

5.5.1 SOMDedup. 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 [66] 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.

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

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) [18] and hierarchical clustering [28]. 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 [62], 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.

5.5.2 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 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* [7]. We compute the coefficient between the SOURCE regression and each regression in the TARGET group, and use the maximal value.
- *Text cosine similarity* [64]. 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.

5.6 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. Without loss of generality, our description focuses on code changes but can be applied to configuration changes as well.

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 2 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 2. 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.

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

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 2. Example of gCPU changes involving subroutine B.

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.

6 Evaluation

Our evaluation answers the following questions:

- 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?

6.1 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 3 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 3, 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.

6.2 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 3), 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 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

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 (§5.2.1 and §5.3)	3.96M	1.09K	324.85k	239.67K	1.9K
After went-away detection (§5.2.2)	1/365	——	1/82	1/47	——
After seasonality detection (§5.2.3)	1/468	——	1/111	1/58	——
After threshold filtering (Table 1)	1/769	1/1.03	1/295	1/165	1/1.03
After SameRegressionMerger	1/861	1/9.6	1/500	1/320	1/7
After SOMDedup (§5.5.1)	1/3536	1/28	1/1775	1/776	1/18
After cost-shift analysis (§5.4)	1/6092	1/73	1/4010	1/776	1/18
After PairwiseDedup (§5.5.2)	1/18857	1/91	1/5240	1/1180	1/30
Total	1/7779				

Table 3. Effectiveness of individual techniques in filtering out spurious change points. All non-first-row numbers 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.

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. Assuming the same ratio (49:21) holds for the 217 reported regressions, a FrontFaaS developer would only need to investigate one false-positive regression approximately once every 13 years. 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 4). However, this would delay the timeliness of regression detection. This trade-off is challenging and requires future research.

6.3 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 2%. 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 3. 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%)*17/(61-11)=22%.

6.4 Catching Small Regressions

FBDetect is effective at catching small regressions, as shown in Table 4, 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 §6.2, false positives are mostly cost shifts, indicating an area for future research.

	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 4. 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%.

6.5 Comparison with Yahoo's EGADS

We compare FBDetect with Yahoo's EGADS [39], which offers multiple anomaly detection algorithms. The test data consists of a random set of 107 time series where FBDetect reported 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 §6.2, we assume its false negative rate is zero. FBDetect's false-positive and false-negative rates are shown in Figure 8.

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 8. 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

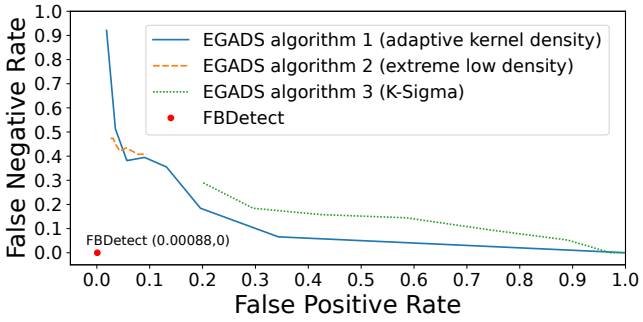


Figure 8. EGADS's algorithms cannot simultaneously reduce both false negatives and false positives.

be futile, eroding developer's trust in the algoritm. Thus, we require a false positive rate below 760/(35K+31) ≈ 0.02. Among the EGADS algorithms in Figure 8, 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.

6.6 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 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.

7 Related Work

Performance testing and monitoring. Performance regression can be detected before production [32, 46, 84] or during production [2, 25, 40, 76]. ServiceLab [14] and FBDetect are representatives for these two categories, and they are used at Meta in complementary ways. Due to their different operating environments, they differ in major ways: 1) ServiceLab controls its testing environment, such as selecting testing servers to reduce variance, whereas FBDetect have no such control and must rely on subroutine-level measurements and a larger number of samples to manage higher variance. 2) ServiceLab assumes a stationary environment and does not need FBDetect 's complex techniques, such as the went-away detector and trend analysis, for handling non-stationary conditions. 3) ServiceLab faces the challenge of faithfully mirroring interdependent production services in its testing environment, whereas FBDetect's in-production monitoring approach avoids this problem.

Change point and seasonality detection. Change point detection tries to identify points where the statistical properties of a time series change [5, 6, 11, 16, 26, 33, 44, 58, 81, 85, 86]. Seasonality detection tries to identify the presence of regular and periodic changes [17, 50, 65, 70]. Several industrial systems have adopted these methods [20, 22, 49, 73, 74]. FBDetect further introduces the went-away detector to filter out transient issues, which can account for up to 99.7% of all regressions in our production environment.

Stack trace sampling. Existing tools can collect stack traces for services written in C/C++ [53], Java [27, 54], Python [8, 15, 23, 24, 35, 52, 59–61, 71], Go [19], Ruby [31], etc. 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 [29], files [37, 38, 48, 51, 57, 63, 78, 80, 82], commits [9, 77, 79], or deployments [42]. 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 [1, 13, 83]. 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 and Future Work

We have presented FBDetect, an in-production performance-regression detection system. It saves millions of servers each year and reduces the number of performance anomalies requiring developers' attention by three to four orders of magnitude. Our contributions include the went-away detector, cost-shift detector, and several algorithms that work in concert to catch regressions as small as 0.005% in noisy production environments.

A major direction for future work is regression detection for GPU training. Currently, GPU profiling is less mature, and the synchronous training pattern significantly differs from the traditional microservice pattern. Additionally, we aim to reduce the false-positive rate for traditional services, with a focus on improving cost-shift analysis. Planned capacity changes also trigger false positives, so we plan to correlate regressions with these known changes. Finally, potential new application domains for FBDetect include detecting anomalies in site and hardware reliability.

Acknowledgments

This work involves several teams at Meta, including FBDetect, Web Efficiency, Core Data Efficiency Experience, Instagram Efficiency, Ads Serving Platform/Efficiency, MRS Foundation Serving Infra, and Infrastructure Data Science. We thank those who have contributed to FBDetect: Adwait Sathye, Ayichew Hailu. Chinnappa Pattada, Cindy Weng, David Ehrmann, Diego Plascencia, Ismail Dalgic, Jeff Song, Joshua Louie, Kevin Jeong, Nipun Gupta, Samuel Cheng, Saurabh Jain, Tola Liadi, Trung Nguyen Ba, and Varun Thangavelu. We also thank all reviewers, especially our shepherd, Ding Yuan, for their insightful comments.

References

[1] 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. https://doi.org/10.1109/ICSE48619.2023.00149

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

[3] 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. https://doi.org/10.1007/s10115-016-0987-z

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

[5] Daniel Barry and J. A. Hartigan. 1993. A Bayesian Analysis for Change Point Problems. *J. Amer. Statist. Assoc.* 88, 421 (1993), 309–319. http://www.jstor.org/stable/2290726

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

[7] 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. https://doi.org/10.1007/978-3-642-00296-0_5

[8] 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

[9] 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

[10] 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

[11] Sofiane Brahim-Belhouari and Amine Bermak. 2004. Gaussian Process for Nonstationary Time Series Prediction. *Computational Statistics Data Analysis* 47, 4 (2004), 705–712. https://doi.org/10.1016/j.csda.2004.02.006

[12] 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

[13] 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. https://doi.org/10.1145/3627703.3629553

[14] Mike Chow, Yang Wang, William Wang, Ayichew Hailu, Rohan Bopardikar, Bin Zhang, Jialiang Qu, David Meisner, Santosh Sonawane, Yunqi Zhang, Rodrigo Paim, Mack Ward, Ivor Huang, Matt McNally, Daniel Hodges, Zoltan Farkas, Caner Gocmen, Elvis Huang, and Chunqiang Tang. 2024. ServiceLab: Preventing Tiny Performance Regressions at Hyperscale through Pre-Production Testing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 545–562. https://www.usenix.org/conference/osdi24/presentation/chow

[15] Sümer Cip. 2024. `Yappi`: Yet Another Python Profiler. https://github.com/sumerc/yappi.

[16] Ian Cleland, Manhyung Han, Chris Nugent, Hosung Lee, Sally McClean, Shuai Zhang, and Sungyoung Lee. 2014. Evaluation of Prompted Annotation of Activity Data Recorded from a Smart Phone. *Sensors* 14, 9 (27 Aug. 2014), 15861–15879. https://doi.org/10.3390/s140915861

[17] 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.

[18] Thomas Cover and Peter Hart. 1967. Nearest Neighbor Pattern Classification. *IEEE transactions on information theory* 13, 1 (1967), 21–27. https://doi.org/10.1109/TIT.1967.1053964

[19] Russ Cox and Shenghou Ma. 2013. Profiling Go Programs. https://go.dev/blog/pprof.

[20] David Daly, William Brown, Henrik Ingo, Jim O'Leary, and David Bradford. 2020. The Use of Change Point Detection to Identify Software Performance Regressions in a Continuous Integration System. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (Edmonton AB, Canada) *(ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 67–75. https://doi.org/10.1145/3358960.3375791

[21] Wilfrid J Dixon and Frank J Massey Jr. 1957. *Introduction to Statistical Analysis*. McGraw-Hill.

[22] Matt Fleming, Piotr Kolaczkowski, Ishita Kumar, Shaunak Das, Sean McCarthy, Pushkala Pattabhiraman, and Henrik Ingo. 2023. Hunter: Using Change Point Detection to Hunt for Performance Regressions. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering* (Coimbra, Portugal) *(ICPE '23)*. Association for Computing Machinery, New York, NY, USA, 199–206. https://doi.org/10.1145/3578244.3583719

[23] Ben Frederickson. 2024. `py-spy`: Sampling profiler for Python programs. https://github.com/benfred/py-spy.

[24] Pablo Galindo. 2022. Python Support for the Linux `perf` Profiler. "https://docs.python.org/3.12/howto/perf_profiling.html.

[25] 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.usenix.org/conference/osdi23/presentation/grubic

[26] Manhyung Han, La The Vinh, Young-Koo Lee, and Sungyoung Lee. 2012. Comprehensive Context Recognizer Based on Multimodal Sensors in a Smartphone. *Sensors* 12, 9 (17 Sept. 2012), 12588–12605. https://doi.org/10.3390/s120912588

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

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

[29] 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. https://doi.org/10.1145/581339.581397

[30] 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. https://doi.org/10.1145/3132747.3132749

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

[32] 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. https://doi.org/10.1109/IISWC.2016.7581261

[33] 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. https://doi.org/10.1109/ICDM.2007.78

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

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

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

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

[38] 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. https://doi.org/10.1109/ASE.2015.73

[39] 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. https://doi.org/10.1145/2783258.2788611

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

[41] 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. https://doi.org/10.1016/j.jesp.2013.03.013

[42] 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

[43] 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. https://doi.org/10.1145/882082.882086

[44] 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. https://doi.org/10.1016/j.neunet.2013.01.012

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

[46] 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

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

[48] 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. https://doi.org/10.1109/ICSME.2014.37

[49] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. 2021. Identifying Software Performance Changes across Variants and Versions. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 611–622. https://doi.org/10.1145/3324884.3416573

[50] Marc Nerlove. 1964. Spectral Analysis of Seasonal Adjustment Procedures. *Econometrica* 32, 3 (1964), 241–286. http://www.jstor.org/stable/1913037

[51] 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. https://doi.org/10.1109/ASE.2011.6100062

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

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

[54] 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.

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

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

[57] 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. https://doi.org/10.1145/3236024.3236065

[58] Sasank Reddy, Min Mun, Jeff Burke, Deborah Estrin, Mark Hansen, and Mani Srivastava. 2010. Using Mobile Phones to Determine Transportation Modes. *ACM Trans. Sen. Netw.* 6, 2, Article 13 (March 2010), 27 pages. https://doi.org/10.1145/1689239.1689243

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

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

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

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

[63] 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. https://doi.org/10.1109/ASE.2013.6693093

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

[65] Christopher A. Sims. 1974. Seasonality in Regression. *J. Amer. Statist. Assoc.* 69, 347 (1974), 618–626. http://www.jstor.org/stable/2285991

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

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

[68] 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.

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

[70] J. J. Thomas and Kenneth F. Wallis. 1971. Seasonal Variation in Regression Analysis. *Journal of the Royal Statistical Society. Series A (General)* 134, 1 (1971), 57–72. http://www.jstor.org/stable/2343974

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

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

[73] Martin Valdez-Vivas, Caner Gocmen, Andrii Korotkov, Ethan Fang, Kapil Goenka, and Sherry Chen. 2018. A Real-time Framework for Detecting Efficiency Regressions in a Globally Distributed Codebase. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 821–829. https://doi.org/10.1145/3219819.3219858

[74] Owen Vallis, Jordan Hochenbaum, and Arun Kejariwal. 2014. A Novel Technique for Long-Term Anomaly Detection in the Cloud. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*. USENIX Association, Philadelphia, PA. https://www.usenix.org/conference/hotcloud14/workshop-program/presentation/vallis

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

[76] 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

[77] 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. https://doi.org/10.1145/2970276.2970359

[78] 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. https://doi.org/10.1109/ICSME.2014.40

[79] 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. https://doi.org/10.1145/3180155.3182516

[80] 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. https://doi.org/10.1145/2610384.2610386

[81] Kenji Yamanishi and Jun-ichi Takeuchi. 2002. A Unifying Framework for Detecting Outliers and Change Points from Non-Stationary Time Series Data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '02)*. Association for Computing Machinery, New York, NY, USA, 676–681. https://doi.org/10.1145/775047.775148

[82] 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. https://doi.org/10.1145/2635868.2635874

[83] 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. https://doi.org/10.1145/3663529.3663858

[84] 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. https://doi.org/10.1109/ISCA.2016.47

[85] Yu Zheng, Yukun Chen, Quannan Li, Xing Xie, and Wei-Ying Ma. 2010. Understanding Transportation Modes Based on GPS Data for Web Applications. *ACM Trans. Web* 4, 1, Article 1 (Jan. 2010), 36 pages. https://doi.org/10.1145/1658373.1658374

[86] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning Transportation Mode from Raw Gps Data for Geographic Applications on the Web. In *Proceedings of the 17th International Conference on World Wide Web* (Beijing, China) *(WWW '08)*. Association for Computing Machinery, New York, NY, USA, 247–256. https://doi.org/10.1145/1367497.1367532

A Appendix

Following the practice at SOSP'24, we declare that the appendix was not peer reviewed by the SOSP committee.

A.1 Leveraging the Law of Large Numbers

The Law of Large Numbers (LLN) [21] 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. Lete $Variance(\overline{x})$ 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 2 and 3 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 [21] 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 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 [4], 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.

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.