Exploring Statistical Change Point Detection Techniques for Performance Anomaly Detection at Mozilla

Mohamed Bilel Besbes

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

July 2026

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mohamed Bilel Besbes**

Entitled: **Exploring Statistical Change Point Detection Techniques for Performance Anomaly Detection at Mozilla**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Tse-Hsun Chen

_____ Examiner
Dr. Shin Hwei Tan

_____ Supervisor
Dr. Diego Elias Damasceno Costa

Approved by _____
Dr. Joey Paquet, Chair
Department of Computer Science and Software Engineering

_____ 2026

_____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Exploring Statistical Change Point Detection Techniques for Performance Anomaly Detection at Mozilla

Mohamed Bilel Besbes

Software performance regressions can have significant business consequences, making automated detection a critical component of modern continuous integration and delivery pipelines. At Mozilla, performance anomaly detection is handled by Perfherder, Mozilla's performance engineering management system that relies on a Student's T-test-based approach to flag regressions across hundreds of daily code changes. However, a preliminary analysis of one year of Mozilla performance data reveals that at least 12.47% of generated alert groups are false positives, while approximately 6.8% contain regressions that the automated system misses, both of which could impose costs on performance engineering teams and end users.

This thesis presents a large-scale empirical study evaluating 25 change point detection (CPD) methods spanning offline, online, and hybrid methods, as well as 15 ensemble approaches, as alternatives to Mozilla's current method. To enable rigorous evaluation, we construct a ground-truth dataset of 174 performance time series manually annotated by eleven Mozilla performance engineers, representing one of the first large-scale, practitioner-annotated CPD benchmarks for performance engineering. We also extend the *TCPDBench* evaluation tool with previously unrepresented method families. Our results show that while offline and hybrid CPD methods improve recall over Mozilla's method, they do so at a significant precision cost. Ensemble voting strategies alleviate this tradeoff as an offline methods' ensemble peaks at an F1-score of 78.4%, and a hybrid methods ensemble paired with Mozilla's method achieves 77.2%, both surpassing the baseline F1-score of 70.6% while

maintaining acceptable false alert rates. These findings are validated through a practitioner survey completed by twelve Mozilla engineers, who unanimously endorse the ensemble voting approach as the most viable candidate for deployment. Finally, we deploy the proposed candidate and report lessons learned to inform both researchers and practitioners about the practical challenges of transitioning from statistical baselines to ensemble-based anomaly detection at scale.

Dedication

This work is dedicated to the loving memory of *Hamdi Besbes*. Also, this work was completed on the unceded traditional territory of the *Kanien'kehá:ka* Nation, the custodians of *Tiohtià:ke*/Montréal, to whom I extend my respect and gratitude.

Acknowledgments

Completing my MASc. degree has been one of the most fulfilling experiences of my life, made possible by the guidance, collaboration, and encouragement of many wonderful people. Above all, I am profoundly grateful to my supervisor, Dr. Diego Elias Damasceno Costa, whose unwavering support, patience, and thoughtful feedback challenged me to grow into a better researcher and a more confident thinker. I am thankful to my industrial supervisors, Gregory Mierzwinski and Suhaib Mujahid, for their dedication to my progress and for the many conversations that sharpened my thinking throughout this work. I also wish to thank Dave Hunt, performance engineering team leader at Mozilla, for welcoming me through the Mitacs internship and for his continued investment in my research. This work was made possible through the support of Mitacs and the Mitacs Accelerate Program. I am grateful for the opportunity to have been part of a program that fostered both my academic and professional growth. A heartfelt thank you goes to Natalia, the program coordinator, whose attentiveness and guidance were a constant source of reassurance. I also want to express my sincere appreciation to my colleagues in the REALISE Lab for cultivating an environment where curiosity and collaboration thrived. In particular, thank you to Rachna, Geneviève, Khaled, Zackariya, Kawsar, Yasmine, Haya, and Adam. Your advice, warmth, and conversations made the day-to-day of research genuinely enjoyable. Lastly, and most deeply, I am grateful to my parents, siblings, and friends, whose love and belief in me never wavered. You have been my foundation through every difficult moment, and this achievement belongs to you as much as it does to me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Introduction

Slow and poorly optimized software can reduce customer satisfaction, increase operational costs, and drive users away from using a software product [1]. It has been reported that a 1-second delay can cost *Amazon* 1.6 Billion USD in sales [2]. In fact, a report by *Google*, *Deloitte*, and *55* showcase that a 0.1-second site responsiveness improvement results in 8% more user traffic and 10% more revenue [3]. To ensure software performance complies with user expectations, large software companies employ a rigorous process for testing performance [4], often under the umbrella of *performance engineering* [5].

In large software development companies such as Mozilla, developers commit hundreds of code changes every day. Measuring and monitoring the impact of such changes on the software performance is inherently challenging. Designing reliable performance tests is difficult, as tests must be both representative of real-world workloads and reproducible across executions [4, 6]. Software performance is known to be influenced by external factors (e.g., hardware non-determinism), which introduce noise that is hard to control [7–9]. As a result, distinguishing genuine performance regressions from measurement noise is a major challenge in performance engineering, particularly since performance changes often manifest subtly at runtime and may go unnoticed during code review [10–12].

Therefore, reliable performance testing requires human orchestration and dedicated tooling to manage the performance engineering workflow [13–16]. A notable example is *MongoDB*, which uses the E-Divisive change point detection algorithm to automatically flag potential regressions across hundreds of benchmarks [17]. Mozilla follows a similar model through *Perfherder* [18], its own performance engineering management system. It offers a wide selection of performance tests across multiple environmental configurations. It analyzes performance measurement history to detect potential regressions using a statistical change point detection approach [19]. Detected regressions are surfaced as *performance alerts*, which are manually triaged by performance engineering experts known as *Performance Sheriffs* and dispatched to the responsible development teams for remediation.

Despite this structured process, change point detection is inherently imperfect as no method can fully eliminate false alerts or capture all regressions [20]. A preliminary analysis of Mozilla performance data reveals that Mozilla's method produces a non-negligible proportion of both failure modes. While the rates may appear modest in isolation, at the scale of Mozilla's continuous integration process where hundreds of revisions are tested daily across a large set of tests, even a small proportion of false or missed alerts translates to meaningful overhead for Performance Sheriffs and potential quality risk for end users. This motivates a systematic investigation into whether alternative statistical methods can reduce the rate of false and missed alerts in this industrial context.

Our thesis answers the following research questions:

- **RQ1: To what extent does Mozilla's method produce false alerts and miss real ones?**

  We answer this research question in Chapter 3 by conducting an empirical study of Mozilla's performance engineering workflow through presenting its underlying mechanism as well as presenting quantitative insights on it based on data collected from live Mozilla systems. This sets the foundation to quantitatively describe the limitations of the current system, establishing the motivation for the thesis's contribution area. To this end, we begin the characterization of Mozilla's method's limitations by

quantifying the scale of false and missed alerts through a preliminary analysis of one year of Mozilla performance data.

- **RQ2: How effective are CPD methods in detecting performance changes at Mozilla?**

  Through Chapter 4, we answer this question by evaluating a broad set of CPD methods spanning offline, online, hybrid, and ensemble approaches against a practitioner-annotated ground-truth dataset of Mozilla performance time series to ultimately identify the most promising candidates for deployment in Mozilla's performance engineering pipeline. To conduct the evaluation, we build upon *TCPDBench* [21], an established CPD benchmarking toolset, which we extend with 15 previously unrepresented methods to cover online and hybrid families alongside the existing offline ones. We further construct a ground-truth dataset of 174 performance time series, annotated manually by eleven Mozilla practitioners, to benchmark CPD methods against Mozilla's method in a realistic industrial setting.

- **RQ3: How do experimental CPD findings translate into practice at Mozilla?**

  We report through Chapter 5 a practitioner validation study combining a workshop presentation and a follow-up survey with Mozilla practitioners, answering RQ3. Since strong benchmark performance alone is insufficient to justify deployment in an industrial setting, we complement our quantitative evaluation with a workshop and survey involving twelve Mozilla performance engineers. This allows us to validate whether the methods that perform best on our benchmark are also considered suitable for deployment by the practitioners who would use them, and whether the assumptions underlying our evaluation reflect the realities of Mozilla's performance testing workflow. Finally, we deploy the solution set endorsed by practitioners and report the challenges encountered upon deploying it.

1.2 Research Statement

Motivated by the scale and complexity of continuous performance regression detection and the critical cost that false alerts and missed detections impose on performance engineering teams and end users, the goal of this thesis is to investigate whether statistical change point detection methods can reliably improve upon Mozilla's current alerting approach in an industrial CI/CD setting. We state our research statement as follows:

> Performance regression detection in large-scale continuous integration systems is an inherently imperfect process, where both false alerts and missed detections carry real operational costs. This thesis systematically evaluates statistical change point detection methods as alternatives to Mozilla's current Student's t-test-based approach. Specifically, it quantifies the limitations of the existing method, benchmarks 25 CPD methods and 15 ensemble strategies against a practitioner-annotated ground-truth dataset of 174 performance time series, validates the most promising candidates through a survey of twelve Mozilla performance engineers to assess their suitability for deployment in Perfherder, and reports lessons learned and challenges from applying the experimental setup findings into Mozilla's systems.

1.3 Thesis Contribution

In this thesis, we provide the following contributions:

- We investigate academic literature in order to identify alternatives for the currently used change point detection process that is based on Student T-Test.
- We publish a dataset of 5,655 time series composed of performance measurements obtained from production-scale Mozilla infrastructure and we publish it as part of a paper [22].
- We provide a dataset of 174 time series from Mozilla systems that are manually annotated by eleven Mozilla experts.

- We evaluate 25 change point detection methods given the collected baseline of annotations, and we leverage these methods through voting system-based approaches.
- We report insights from feedback received from twelve industrial experts on the research contribution through a survey aiming to present experimental findings and to evaluate their applicability in real-life setups.
- We report challenges encountered upon deploying a solution coming from our research experimental setup into a real-life environment.

1.4 Thesis Organization

In this section, we provide a thesis overview and highlight the main themes of each chapter.

Chapter 2: Related Works This chapter reviews the body of research in relation to the thesis, organized around four interconnected areas. It covers software performance engineering practices such as noise handling, cost optimization, and performance signatures, illustrating them through industrial tools. It then examines change point detection methods, distinguishing offline from online approaches and surveying their applications in software engineering in performance and beyond. Finally, the chapter critically examines how existing practitioner validation studies confirm detected issues but rarely involve practitioners in evaluating the research assumptions underlying the evaluation framework itself, a gap that the thesis directly addresses.

Chapter 3: Practices of Software Performance Testing This chapter investigates Mozilla's performance engineering workflow and identifies a contribution area. Initially, it describes the methodology and introduces the key concepts underpinning Mozilla's performance testing workflow. We proceed with a data analysis of our collected dataset, which comprises 5,655 performance time series, 17,989 alerts, and 3,912 alert summaries drawn from Mozilla live systems, and report their distribution across repositories, platforms, and

triage categories. We then detail Mozilla's current change point detection method, a four-step pipeline combining Student T-test analysis with magnitude thresholding and manual triage by Performance Sheriffs. Finally, we conduct a preliminary analysis of false and missing alerting, finding that at least 12.47% of alert summaries are false and that 6.8% contain at least one manually created alert, motivating the need for improved detection methods.

Chapter 4: Comparing Change Point Detection Methods for Software Performance Regression This chapter presents a large-scale empirical evaluation of 25 CPD methods against a practitioner-annotated ground-truth dataset of 174 Mozilla performance time series. To construct the ground truth, eleven experienced Mozilla engineers annotated time series through a custom-deployed web platform, with each series assigned to multiple annotators for reliability. Methods spanning offline, online, hybrid, and ensemble approaches were evaluated within the TCPDBench benchmarking framework across multiple hyperparameter configurations. Results show that while most individual methods improve recall at the cost of precision, ensemble approaches effectively balance this tradeoff and surpass Mozilla's current method in overall F1-score.

Chapter 5: An Industrial Validation of Experimental Results This chapter presents a practitioner validation study conducted through a workshop and survey distributed to Mozilla practitioners, collecting responses from twelve experienced practitioners. The survey validated two key experimental assumptions and gathered practitioner assessments of the proposed CPD methods and voting strategies. Practitioners strongly prioritized minimizing missed alerts over false alerts, and while largely satisfied with the current method's precision, broadly agreed its recall is unacceptably low. Voting strategies received the strongest endorsement for deployment, with practitioners accepting the precision trade-off in exchange for meaningful gains in recall.

Chapter 6: Discussions and Limitations This chapter discusses the broader implications of our findings for both researchers and practitioners, and reports the lessons learned

from integrating and deploying the proposed ensemble-based solution into Mozilla's performance engineering pipeline. It then examines the threats to the validity of our findings across construct, internal, and external dimensions, acknowledging the assumptions and design choices that may affect how our results should be interpreted.

Chapter 7: Conclusion & Future Work This chapter summarizes the contributions of the thesis and reflects on the key findings across all research questions. It reaffirms that ensemble voting strategies represent the most viable path forward for performance anomaly detection at Mozilla, and discusses the practical considerations that emerged during deployment. The chapter concludes by outlining several promising directions for future research, including the incorporation of non-statistical CPD methods, generalization beyond Mozilla's context, and approaches to reduce the delayed issue effect inherent in batch-based detection systems.

Related Publications: The works presented in this thesis are as follows:

- Accepted paper: **Mohamed Bilel Besbes**, Diego Elias Costa, Suhaib Mujahid, Gregory Mierzwinski, Marco Castelluccio. A Dataset of Performance Measurements and Alerts from Mozilla. Accepted in 16th ACM/SPEC International Conference on Performance Engineering (ICPE'25).
- Work in progress: **Mohamed Bilel Besbes**, Gregory Mierzwinski, Suhaib Mujahid, Philipp Leitner, Alexander Serebrenik, Diego Elias Costa. Exploring Statistical Change Point Detection Techniques for Performance Anomaly Detection at Mozilla. Work in progress to be submitted to ACM Transactions on Software Engineering and Methodology (TOSEM).

The following work does not take part of the current thesis but it has been worked on throughout the Masters:

- Work in progress: Kawsar Ahmed Bhuiyan, **Mohamed Bilel Besbes**, Rachna Raj, Adam Al Assil, Diego Elias Costa. Beyond Compliance: A Large Scale Study on

the Completeness and Consistency of the GitHub SBOMs. Work in progress to be submitted to Empirical Software Engineering (EMSE).

Chapter 2

Literature Review

This chapter surveys the body of research most relevant to the contributions of this thesis, organized around four interconnected areas. First, we review work on software performance engineering, covering the practices, tools, and workflows that organizations use to monitor and maintain the performance of software systems in industrial and research settings in Section 2.1. Second, we examine the literature on change point detection, covering the definition and taxonomy of CPD methods as well as their application to software engineering problems beyond performance in Section 2.2. Third, we review statistical CPD methods applied specifically to software performance data, covering both offline and online detection approaches and their adoption in industrial settings in Section 2.3. Fourth, we review work on bridging research and practice in software engineering, examining how prior studies have engaged with industry practitioners to assess the relevance and deployability of proposed solutions in Section 2.4. Finally, Section 2.5 summarizes the chapters.

2.1 Software performance engineering

Smith and Williams [23] define *Software Performance Engineering* as a systematic, quantitative approach to constructing software systems that meet performance objectives, beginning early in the software development process to model the performance of the proposed architecture and high-level design.

Performance engineering in software systems involves a set of recurring practices observed across both industry and research. A central concern is **noise handling**, as performance measurements are inherently non-deterministic due to hardware variability and cloud infrastructure contention. To address this, Daly [24] propose Canary tests to assess testbed stability before trusting detected changes, and recommend retriggering performance tests multiple times to obtain reliable measurements. Similarly, Laaber et al. [9] advocate repeated test executions across multiple instances and apply conservative outlier removal to filter extreme anomalies caused by transient infrastructure issues. Maricq et al. [25] further propose an iterative outlier detection approach that excludes the least representative servers using MMD-based dissimilarity rankings to maintain a representative testbed. Mozilla's approach to minimize noise is to run tests multiple times on the same revision and on the same configuration, and subsequently average these runs' measurements. A second recurring concern is **cost optimization**, as running performance tests on every commit is rarely feasible at scale. Daly [24] addresses this by running the performance workflow every 2 to 24 hours rather than per commit, while Reichelt and Kühne [26] propose selecting only specific performance tests to run instead of executing the full test suite. Huang et al. [27] further introduce a Performance Risk Analysis method that prioritizes testing on high-risk commits and reduces or skips tests on low-risk ones. Mozilla addresses this concern by running performance testing intermittently on a handful of revisions, and subsequently run testing more intensively on the untested revisions only if performance anomalies are detected throughout the revisions that are already tested. A third common practice involves the use of **performance signatures**, which are curated sets of performance metrics that capture the essential behavioral characteristics of a system under test. Malik et al. [28] adopt this concept to track performance evolution in a structured and reproducible way, while Ingo [29] treat performance metrics as time series specific to test environments, modeling each series independently to enable real-time incremental analysis. Mozilla adopts a common concept to abstract test measurements coming from the same testing setup.

Despite the existence of common practices and concepts being shared between software performance engineering works, the term *Software Performance Engineering* could still be

used as an umbrella term presenting a wide range of practices and toolset all aiming to monitor and enhance software performance. Li et al. [30] state that there is a lack of standard methodologies for performance evaluation even within the scope of experimental computer science, noting that existing studies typically employ ad hoc approaches rather than rigorous means, with some evaluators focusing only on metrics while others only highlight benchmarks, making it impossible to obtain a one-size-fits-all methodology for evaluating software system performance.

MongoDB has a performance engineering workflow to identify regressions so they could be addressed before they hit the end user. Daly et al. [17] reports that MongoDB designed and maintains a framework called *Distributed System Infrastructure*, or *DSI* for short to run hundreds of performance benchmarks per day. The framework is modular, extensible, and integrable into the Continuous Integration system built on Evergreen [31] but also supports manual triggering. For scale, DSI had 200 unique tests that could run on 20 different MongoDB configurations as of March 2020.

Reichelt and Kühne [26] propose a multi-step method called *Performance Analysis of Software Systems* (PeASS), for detecting changes causing performance issues. It builds up a comprehensive knowledge base of changes affecting the performance of a software system by analyzing the version history of a repository using its unit tests. It also uses regression test selection for saving measurement time and subsequently expediting root cause isolation.

Mühlbauer et al. [15] propose the *active revision sampling* approach, which aims at tracking and understanding a system's performance history by approximating the performance behavior of a software system across all of its revisions. It measures the performance of selected revisions step by step to create a model that shows how performance changes over time. To decide which revisions to measure next, it uses Gaussian Process models to find where the model is most uncertain.

Lee et al. [16] proposes a performance anomaly detection and analysis *framework Performance Anomaly Detector* (PAD) that has several components such as a continuous performance monitor and a differential profiler, a technique that compares performance data from two program runs to identify which parts of the code caused performance changes.

Based on the performance engineering workflow analysis for DataStax, Fleming et al. [32] introduce *Hunter*, an open-source tool that uses change point detection to identify statistically significant shifts in performance metrics. Designed to reduce manual triage, Hunter analyzes benchmark data from a central datastore and has been adopted across distributed systems for its usability and effectiveness in detecting regressions and improvements.

Ferme and Pautasso [33] present *BenchFlow*, a framework for holistic and continuous software performance assessment. It automates the benchmarking process by orchestrating workload generation, test execution, and performance metric collection in realistic, containerized environments. It also supports evaluating performance across different deployment configurations and process engine implementations, enabling reproducible, end-to-end performance experiments. The framework integrates into DevOps pipelines and facilitates performance regression detection by emphasizing repeatability and controlled variability in performance measurements.

The *AutoPerf* framework is introduced by Alam et al. [34]. It is a tool designed to automate regression testing through the integration of zero-positive learning (ZPL), autoencoders, and hardware telemetry. *AutoPerf* leverages hardware performance counters (HWPCs) to gather lightweight, fine-grained runtime information from parallel programs.

Unlike other studies, we provide in-depth insights related to the functioning of Mozilla's performance engineering workflow by providing detailed analysis of its output throughout one.

2.2 Change Point Detection Methods

Aminikhanghahi and Cook [20] define a change point as a transition between different states in the process that generates time series data. Change points are defined as abrupt variations in time series data that may represent transitions occurring between states.

Basseville and Nikiforov [35] define a change point detection method as any algorithm that uses sequential or batch decision rules, including statistical ones, to identify the moment at which the underlying distribution of a time series shifts from one regime to another. They

further divide these methods into offline ones and online ones, where offline methods operate on a fixed, finite sample to either test whether a change occurred or estimate when it did, while online methods process observations sequentially and flag change points as soon as sufficient evidence of a change accumulates.

Change point detection is used in software engineering for a diverse set of purposes, most notably application failure flagging. For example, Artemov and Burnaev [36] apply change point detection methods for failure detection in large-scale software-intensive systems, where runtime metrics such as request counts and queue sizes exhibit quasi-periodic trends and long-range dependence. They propose an ensemble of weak change point detectors combined with an optimal trend estimation algorithm to detect such failures.

Chitsazian et al. [37] use change point detection in the context of *Just-In-Time Software Defect Prediction*, where the statistical properties of commit-level data streams shift over time due to evolving development processes, a phenomenon known as concept drift. They apply the Page-Hinkley sequential change point detection test to spot defects on unlabeled incoming data.

Pham et al. [38] formulate microservice crash detection as a change point detection problem. They propose a technique that employs Multivariate Bayesian Online Change Point Detection to model dependencies across metrics jointly, and show on three benchmark microservice systems that accurately identifying the change point is critical for the downstream root cause analysis task. Their proposed approach consistently outperforms simpler anomaly detectors.

2.3 CPD Methods in Software Performance

A substantial body of research has examined the usage of statistical CPD methods for anomaly detection in software performance data. Daly et al. [17] reports that MongoDB relies on detecting performance changes through a statistical method called E-divisive meanings [39], where the system generates a type of alert once a change is detected based on threshold comparisons. This implementation reduces the number of false positive alerts

from 99% to 40%. The same statistical method is used by Ingo [29] to identify performance regressions in benchmark data.

CUSUM (Cumulative Sum Control Chart) method is employed by Mühlbauer et al. [15] to identify change points in the performance evolution of six real-world software systems. It detects abrupt and substantial performance changes by computing the cumulative deviation from a target value across revisions. Another work by Lee et al. [16] explores the use of CUSUM and Shewhart control charts to identify performance anomalies in database systems during continuous integration. The approach relies on detecting both large and subtle performance shifts introduced by code changes. It is found that CUSUM can reliably detect small performance changes while maintaining a low false alarm rate comparable to Shewhart. We adopt these methods as part of the online methods in our thesis.

Malik et al. [28] use Control Chart and PCA Weights for CPD in performance tests time series originating from seven distinct performance tests designed for different purposes such as CPU stress testing, load generation-based testing, etc. PCA Weights achieves a significantly good F1-score of 82% on average across all used time series.

Jayathilaka et al. [40] reports the efficiency of using CPD methods such as PELT and Binary Segmentation for anomaly detection in the context of Cloud platform applications' performance. The evaluation showcases that the methods effectively detect sustained workload changes and help attribute anomalies to root causes, demonstrating their efficiency in real-world performance monitoring.

Garnett et al. [41] propose embedding BOCPD within a Gaussian Process regression framework by treating change point locations as hyper parameters and marginalizing over them via nonstationary covariance functions. Their approach is evaluated through predictive accuracy and log-likelihood, demonstrating consistent improvements over HMM-based baselines across multiple datasets.

Duplyakin et al. [42] proposed the usage of Robust Segmentation Algorithm upon introducing the use case of performance regression root cause analysis in large-scale systems. This method is robust to outliers and noise, avoiding false detections from sporadic anomalies. It is also efficient for large-scale, sequential data, enabling rapid analysis across many

time series.

Ramakrishnan and Kaur [43] propose a technique for proactively detecting performance anomalies by leveraging the inverse relationship between throughput and response time established by Little's Law. Their approach utilizes Individuals and Moving Range (XmR) control charts to automatically flag early-warning signals when throughput drops below a lower control limit while response times simultaneously exceed an upper limit.

Our thesis consolidates such CPD methods that follow several statistical methods' families such as offline and online detection into the established and openly available *TCPDBench* toolset for anyone to utilize for software performance data specifically.

Complementing these statistical foundations, Machine Learning-based change point detection is also adopted. Chen et al. [44] presents an approach that automatically predicts whether a specific code commit will cause a test to manifest performance regressions is developed. It builds random forest classifiers which combine multiple tree predictors, trained on all prior commits to identify potential regressions, achieving high predictive accuracy with an average AUC of 0.86.

In order to flag performance regressions through *Autoperf*, Alam et al. [34] uses autoencoder neural networks and k-means clustering, allowing the tool to accurately diagnose more performance bugs than previous state-of-the-art approaches.

Despite the high predictive accuracy reported by these ML-based systems, our thesis focuses exclusively on statistical CPD methods for three primary reasons. First, statistical methods do not require the large-scale labeled training sets that ML models necessitate, making them more adaptable to new projects. Second, they offer superior transparency and interpretability, allowing engineers to understand exactly why a regression was flagged.

2.4 Bridging Research and Practice in Software Engineering

Grechanik et al. [45] present *FOREPOST* which is an automated, black-box testing system designed to identify performance bottlenecks by learning from application execution traces. Its effectiveness is validated by industry experts who manually confirm its top 30

identified bottlenecks as genuine. Notably, the system uncovers a resource-intensive bug in a Visitor pattern implementation that, when fixed, results in a 7% improvement in overall application performance.

Zhao et al. [46] introduce *Diff-Design Structure Matrix*, a modeling technique evaluated through an empirical study of 192 performance issues from five Apache projects, using solutions already accepted by industry practitioners. To ensure data integrity, two researchers independently verify the issues to exclude false positives and focus strictly on intentional performance optimizations. This empirical grounding validates the system's ability to model the complex structural changes practitioners use to resolve architectural bottlenecks. By analyzing the Return on Investment of these expert-approved fixes, the study provides a framework for practitioners to prioritize optimization strategies based on measurable effort and performance gains.

Ramakrishnan and Kaur [43] evaluate the effectiveness of their change point detection method through a 23-day production deployment on a large-scale, mission-critical case management system. During this evaluation period, the system successfully identifies all four documented slowness incidents, achieving 100% recall and 100% precision when configured to require two consecutive intervals of violation to filter out transient noise.

While prior industrial validations confirm detected issues through expert review [45], practitioner-accepted fixes [46], or production deployment [43], none explicitly involve practitioners in evaluating the research assumptions underpinning the evaluation framework itself. Our thesis goes further as we consult Mozilla performance practitioners not only to assess the deployment readiness of our proposed methods, but also to validate research assumptions such as the tolerance margin and error-weighting assumptions central to our evaluation design which is an aspect largely absent from existing industrial validation efforts in performance engineering.

2.5 Chapter Summary

The chapter summarizes related work showing that statistical methods are a common practical choice for performance anomaly detection over machine learning-based approaches. The related literature also reveals that existing practitioner validation efforts rarely involve practitioners in scrutinizing the evaluation framework itself, a gap that our validation study aims to address. Together, these observations motivate and contextualize the three core contributions of this thesis, developed in the chapters that follow.

Chapter 3

Practices of Software Performance Testing

Modern software projects rely heavily on continuous integration and delivery pipelines to maintain and improve product quality [47, 48]. Among the many quality dimensions that need to be monitored, performance is particularly critical, as regressions can directly impact end-user experience and product reliability [49, 50]. *Mozilla*, the organization behind the Firefox browser, operates one of the most mature and large-scale open-source performance engineering workflows in the industry, making it a compelling case study for understanding the challenges and limitations of automated performance anomaly detection in practice.

This chapter characterizes Mozilla's performance engineering workflow in depth, analyzes data collected from Mozilla's live systems, and identifies a specific area where a research contribution can be made. We structure our investigation into two phases: a data collection phase, in which we gather performance alerts, time series measurements, and bug reports from Mozilla systems over a one-year period as described in Section 3.1, and a use case analysis phase, in which we examine the properties of the collected data and investigate the inner workings of the alerting workflow as described in Section 3.2. Furthermore, Section 3.3 presents a preliminary analysis of false and missing alerts, motivating the contribution area explored in subsequent chapters. Finally, Section 3.4 summarizes the

chapter.

3.1 Methodology

In order to identify a contribution area, we structure our methodology into two main phases, as illustrated in Figure 3.1.

Data collection. The first phase covers three steps. We begin with a *use case familiarization* step, in which we develop an understanding of Mozilla's performance anomaly detection workflow through **informal communication** via the *Matrix* channel[1] and email exchanges with Mozilla engineers. This allows us to identify four main sources of workflow understanding: *Perfherder*[2], *Wiki Mozilla*[3], the *Treeherder API Documentation*[4], and the *Treeherder Source Code*[5]. We then proceed with *alerts collection*, extracting all performance alerts from the Mozilla API [51] covering the period from May 2023 to May 2024, which corresponds to the one-year retention window of Mozilla systems. We additionally recheck alerts belonging to summaries stuck in a *Still Processing* status, as these have been deprioritized by Mozilla Performance Sheriffs or they have been created around the time we conducted the initial data collection, ensuring our collected alerts reflect the most up-to-date state of the data. Finally, we perform *measurements & bugs extraction*, where we identify the unique time series associated with the collected alerts and extract their features, retaining only time series linked to at least one alert. We cross-reference these measurements with the alerts data, discarding any time series that cannot be matched, and similarly extract the features of the unique bugs associated with the collected alerts.

Use case analysis. The second phase also comprises three steps. The *data analysis* step examines the collected dataset to surface key properties of the system. We then conduct a *workflow investigation*, deepening our understanding of how Mozilla Performance Sheriffs interact with alerts and what drives their triage decisions. We do so by mainly

[1] https://matrix.org/
[2] https://treeherder.mozilla.org/perfherder/
[3] https://wiki.mozilla.org/
[4] https://treeherder.mozilla.org/docs/
[5] https://github.com/mozilla/treeherder

investigating the *Treeherder Source Code*[6]. This informs the final step, *contribution area identification*, in which we synthesize the findings from both phases to pinpoint where a meaningful contribution can be made.

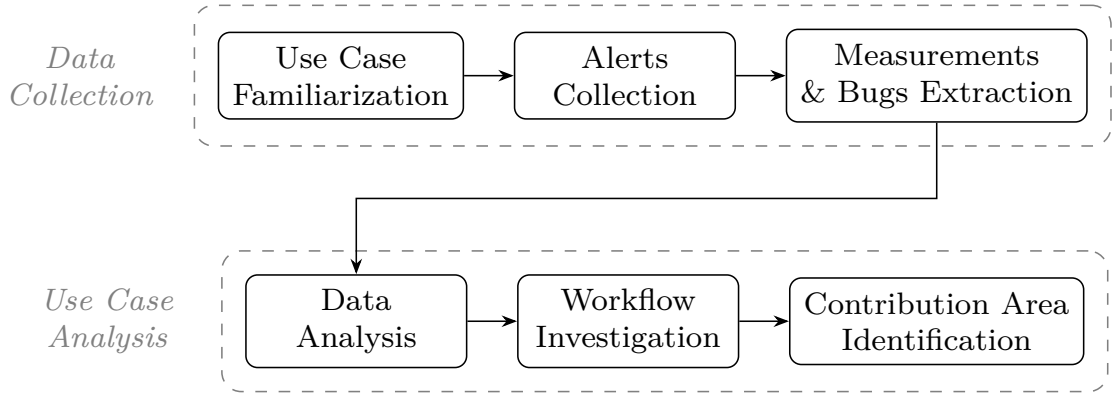


Figure 3.1: Methodology overview

3.2 Mozilla Performance Engineering Use Case Analysis

3.2.1 Use Case Concepts & Definitions

This section contains the description of the concepts that are specific to Mozilla's performance testing workflow.

Platform: It is the hardware setup combined with the software configuration on which tests can run. For example, Windows itself as an operating system is not a platform, but *Windows 11* running on *x64* processors is a platform. Mozilla has a wide range of platforms to make sure that the regression testing covers most of hardware setups for both Desktop and mobile versions of its products.

Performance Test: It presents a performance indicator. An example performance test is *First Visual Change* [52] which measures the time it takes to load the first visual component in the web page, reported in milliseconds. Other tests measure memory usage for example.

Performance Test Suite: A performance test suite at Mozilla encompasses a collection

[6]https://github.com/mozilla/treeherder

of performance tests bundled together into a single framework or tool.

Signature time series: It is a concept to abstract a series composed of performance measurements that are specific to one test running in one framework running on one project running on one platform, and other characteristics. Figure 3.2a showcases an example time series from Perfherder.

It is cost-consuming to run the performance testing workflow on every single code revision. So by default, performance testing runs on one code revision at regular intervals (e.g., every **four hours** for the case of Autoland), not necessarily all the revisions pushed to Mozilla's codebase. Therefore, the time series is composed of performance measurements of only some of the revisions.

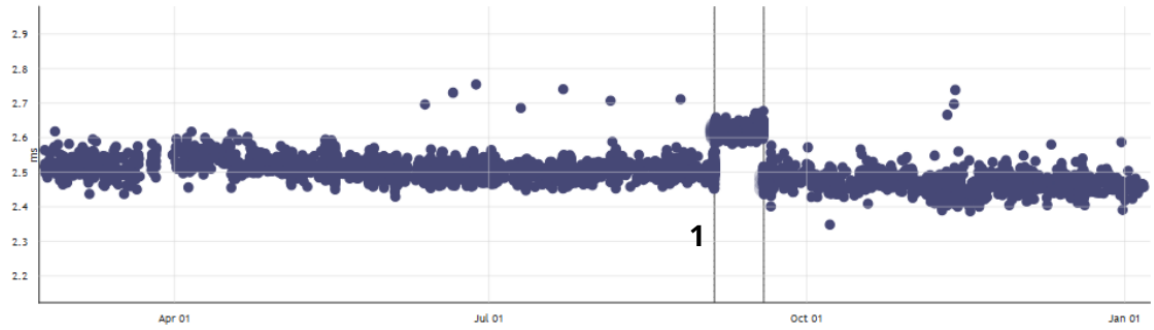
Perfherder[18] enables performance sheriffs to visualize the time series of the given signature. Figure 3.2a represents the time series visualization on Perfherder, the platform that Mozilla performance Sheriffs use in their workflow.

Revision: A *revision*, also known as *commit*, is an immutable snapshot of a project's file tree at a specific moment, uniquely identified, and associated with metadata including the author identity, timestamp, and a descriptive message[53].

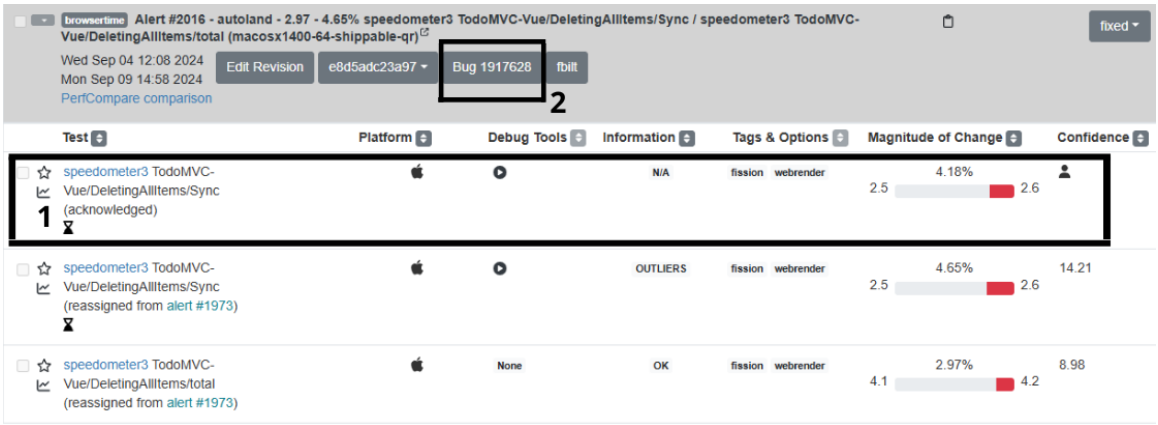
Revision datum: A revision datum D represents the set of measurement values of running testing of a specific signature on one specific revision.

Alerts: An alert is a potential performance anomaly that was triggered by Mozilla *Perfherder* or by performance Sheriffs. The alerts data characterize the performance alert and contains the related alert summary, the alert's status, the time series ID, the result of the t-test, whether the alert was created manually or not, the noise profile of that alert if there is any, and other attributes.

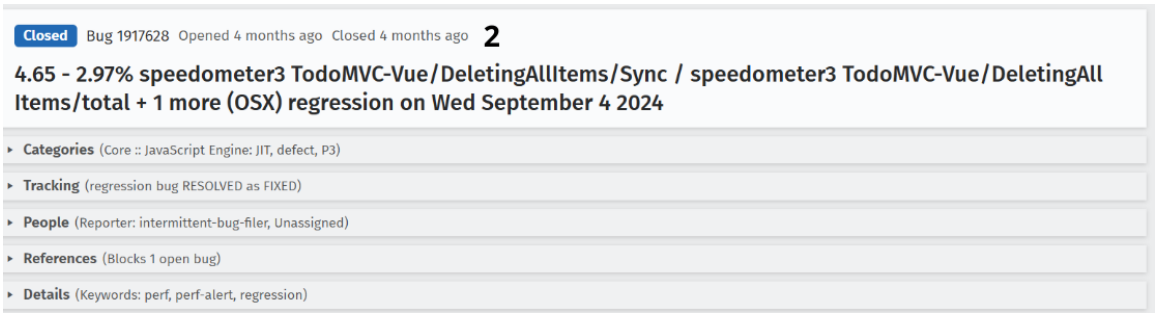
Alert Summaries: An alert summary is a grouping of alerts on the same software revision and other characteristics. It is important to note that, alert summaries are the ones usually validated by Performance Sheriffs. The alert summary data contains details about its related software revision, the triage due date, the assignee among the performance Sheriffing team that will take a look at the alert summary, its associated bug, its status, and other attributes. Depending on its status, we classify an alert summary as part of the



(a) Time series example



(b) Alert summary example



(c) Bug example

Figure 3.2: Illustrative example of the data as seen by Performance Sheriffs

following categories:

- **True Alert summaries:** Represent validated performance regressions. This category includes alerts with statuses such as *fixed, improvement, reassigned, backedout, downstream,* or *wontfix*, where *wontfix* alert summaries could correspond to valid regressions spotted in performance measurements but not reflected in the product itself

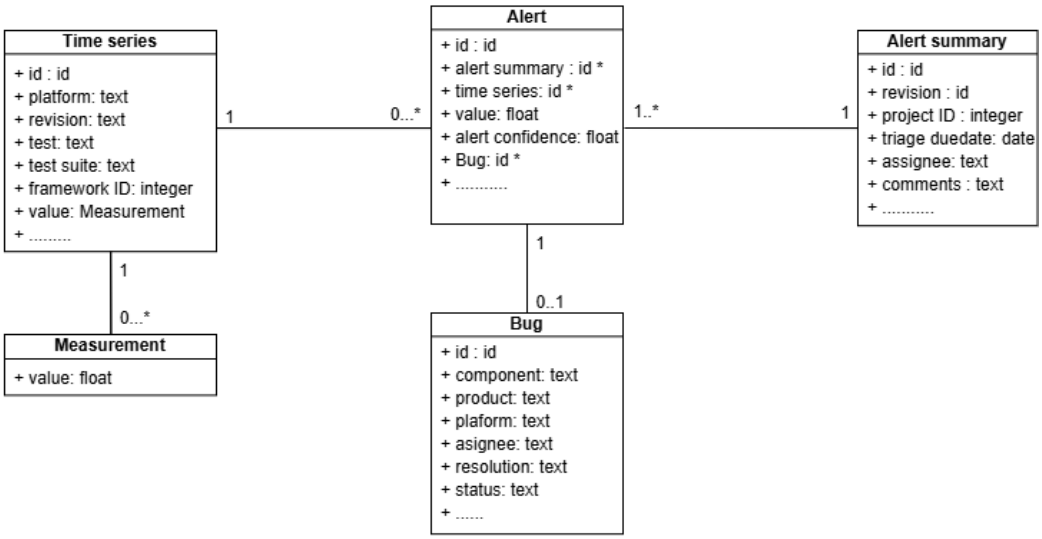


Figure 3.3: Structure of the relationships between the data entities

due to configuration issues.

- **False Alert summaries:** Do not represent real performance issues and often result from noise or irrelevant environmental factors. We identify these summaries by their *invalid* status.

- **Uncertain Alert summaries:** Summaries where the validity remains undetermined and requires further review. These are labeled with statuses such as *investigating* or *untriaged*

Figure 3.2b showcases an example of a performance alert summary. with its associated bug presented in Figure 3.2c. Figure 3.3 showcases the relationships between different entities related to it.

Performance Sheriffing: It is the role assigned to performance engineers in charge of analyzing data and performance metrics from testing frameworks to detect regressions, uncover root causes, and create bug reports to manage and resolve identified performance issues[54].

Backfilling: Mozilla employs a cost-saving strategy within its performance testing workflow in which tests are initially executed on a subset of code revisions at fixed sampling intervals. For instance, every four hours on the Autoland repository. When a performance

regression is detected in one of these sampled revisions and an alert is raised, a backfilling mechanism is triggered to retroactively schedule test runs on intermediate revisions that were not part of the original sample. This process serves to narrow the interval between the last known well-performing revision and the first revision exhibiting degraded performance, thereby enabling engineers to more precisely identify the revision responsible for introducing the regression.

Retriggering: As the nature of performance testing is non-deterministic due to unexpected environmental changes, especially those related to hardware performance, each performance test gets executed multiple times on the same revision and platform. This repetition helps reduce noise and statistical variance, allowing for a more stable and reliable estimation of the system's true performance characteristics.

3.2.2 Analysis of Performance Testing Practices

As part of the workflow investigation explained in Section 3.1, we collect performance measurements from Mozilla systems and reported them as a dataset in our previous work[22]. Table 3.1 showcases some of the statistics of our dataset which contains a total of 5,655 performance time series, covering performance tests of 186 different test suites across 5 different software platform families. On average, each performance time series contains 2,124 measurements associated with 1,867 software revisions. In total, the dataset contains ≈12 million measurements, and only 0.35% of performance measurements are associated with a performance alert.

Analyzing the performance alerts, we report a total of 17,989 alerts, 8,788 of which correspond to alerts from Speedometer3/TP6 test suites, the two of the most important Mozilla performance test suites. When grouped, the alerts correspond to 3,912 unique alert summaries. Most of these summaries are associated with two repositories, Autoland [55] and Mozilla Beta [56], respectively, representing 84.4% and 12.06% of the total alert summaries.

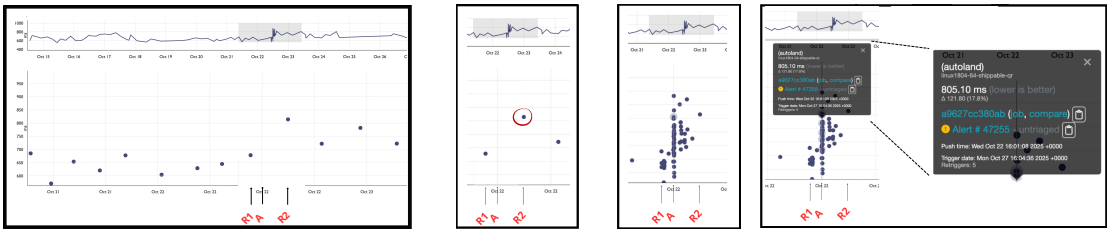
In case a performance regression is identified, it gets associated to a bug to be fixed. It is worth noting that out of the 3,912 existing alert summaries, only 633 have one associated bug out of the 482 unique bugs. Table 3.2 contains the breakdown of bugs by severity and

Table 3.1: Dataset statistics by repository.

Repository	Time Series	Alerts	Custom Alert Labels			Alert Summaries	Bugs
			True	False	Uncertain		
Autoland	3833	13593	1779	431	1109	3319	438
Mozilla Beta	1477	3597	317	45	110	472	25
Firefox Android	342	796	105	12	1	118	25
Mozilla Central	2	2	2	0	0	2	2
Mozilla Release	1	1	0	0	1	1	1
Total	**5655**	**17989**	**2203**	**488**	**1221**	**3912**	**482**

Table 3.2: Characteristics of the 482 performance bugs in the dataset

Characteristic	Level	# of Bugs
Bug Severity	S2	26
	S3	125
	S4	33
	Unknown	298
Bug Status	NEW	35
	ASSIGNED	3
	REOPENED	1
	RESOLVED	427
	VERIFIED	16



(a) Regular intermittent performance evaluation (b) Anomaly detection (c) Backfilling & Retrigger-calization (d) Fine-granular anomaly loging

Figure 3.4: Alert creation example

status. The severity of the bug decrease in magnitude as the numerical value increases.

3.2.3 Mozilla performance anomaly detection workflow

The goal of Mozilla's Performance Testing workflow is to capture and remediate performance regressions as part of the continuous integration and delivery process. Mozilla's performance engineering workflow is orchestrated via a Mozilla-specific system called *Perfherder* [18], the performance alerts management platform used at Mozilla. To mitigate performance test costs, performance tests are run at fixed time intervals, varying per project. For example, in the *Autoland* project [55], performance tests are run every four hours, always testing the latest working revision of the software project. The performance measurements are organized into time series and analyzed to identify performance anomalies. Any performance anomaly detected prompts the creation of a *performance alert*. If multiple performance alerts are created for the same revision, they are grouped into *performance alert summaries*, to facilitate analysis by the performance Sheriffs.

To elaborate on the performance anomaly detection, we present an illustrative example in Figure 3.4. Specifically, Figure 3.4a shows historical performance measurements of a target system, spanning three days of measurements. A noticeable performance change is observed between revisions *R1* and *R2* through running a Student-T-Test based analysis on performance measurements, which is highlighted in Figure 3.4b. Given that multiple revisions may exist between R1 and R2, Perfherder starts a process called *backfilling* & *retriggering*. This approach of isolating performance regressions is well-established in the

performance testing community, having been adopted in performance engineering industrial settings over a decade ago [57].

These processes ensure that any revisions pushed between *R1* and *R2* are performance-tested and measured **multiple times** as performance testing requires multiple testing runs on the same setup in order to reduce the effect of noise coming from hardware nondeterminism [11, 58]. Once the new runs are done, the Student-T-Test-based analysis is run again so the culprit revision, which is A in our case, is identified. Figure 3.4c shows the measurements created by backfilling and retriggering between R1 and R2. Once the culprit revision is identified, an alert associated with it is created as shown in Figure 3.4d.

Mozilla's method of change point detection follows the pipeline described in Figure 3.5. The method is comprised of four steps:

Step 1. Select target measurements: Each performance measurement is associated with a project revision, and is chronologically sorted based on revision dates. To investigate a performance anomaly related to *R2*, the method selects a set of measurements from revisions pushed before R2 (*pre*) and a set of measurements of revisions pushed after R2 and the measurement of R2 revision itself (*post*). It is customary to use 12 to 24 measurements for the *pre* set and 12 measurements for the *post* set.

Step 2. Compare *pre* and *post* sets using Student T test metrics: The method then compares both *pre* and *post* sets using a Student T-test [19]. Instead of relying on the usual p-values, the method instead reports on the computed T-value, as a threshold of interest. If the T-value is strictly above the critical threshold of 7, the revision is marked as a *suspected* anomaly. Otherwise, the given revision is discarded as a suspected performance anomaly, and the system moves on to test other revisions. In our example, the change introduced by A has a T-value equal to 8.237, exceeding the change detection threshold.

Step 3. Compare change magnitude: The previous step's *suspected* anomaly incurs further investigation because the performance measurements before it drastically differ in scale compared to it and ones subsequent to it. Therefore, the magnitude of change between the two sets of measurements defined earlier is computed and compared to a critical value of 2%. In case it surpasses that threshold, an anomaly is confirmed at *R2* such as the example

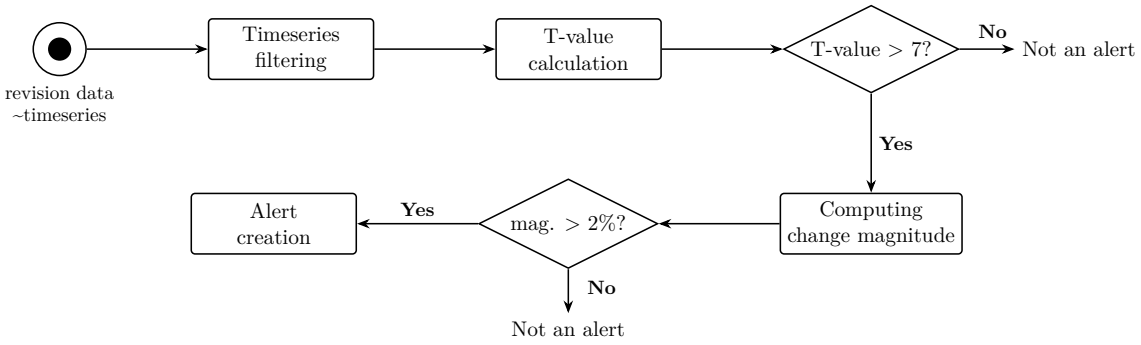


Figure 3.5: Simplified workflow of Mozilla's alert creation process

in Figure 3.4b and an alert is created and further grouped into an alert summary. The practice of double-step change point verification has also been reported in SAP HANA's regression detection use case [59].

Step 4. Manual Triage. Alert summaries are manually investigated by Performance Sheriffs, who conduct a root cause analysis to determine whether the alert corresponds to a genuine performance regression. When a regression is confirmed, the sheriffs file a bug report and assign it to the appropriate development team. They subsequently monitor the issue until it is properly addressed.

In summary, the process of detecting and precisely locating performance anomalies requires intermittent performance evaluation periodically, which creates the measurements used to evaluate performance changes, prompting the detection of a performance anomaly, as shown in Figure 3.4b. Depending on the magnitude of the performance change, it could be further analyzed to assign it to the exact revision that introduced the performance change and create an alert for it, as shown in Figure 3.4d. Subsequently, we refer to the described CPD method as *Mozilla's method* or simply as the *current method*.

3.3 A Preliminary Analysis on Missing and False Alerting

We assess the frequency with which alerts are falsely generated or valid anomalies are missed using the dataset described in Section 3.2.2.

False alerting analysis False alerting is manifested as alert summaries that appear on Perfherder, which do not correspond to actual performance anomalies. They could be caused by infrastructure issues or by high noise levels compared to the size of the change that was detected. Such summaries have to be triaged by performance sheriffs, and a high rate of false alerts cost time, effort, and may lead to practitioners distrusting the alert system. Bug # 1977385[7] on BugZilla is associated with a false alert summary, which required approximately four business days of investigation by an engineer, ultimately revealing that the flagged issue is not an issue to begin with.

Our preliminary analysis of these alert summaries indicates that 56.3% are classified as true alerts, 12.47% as false, and 31.2% as uncertain. This means that at least one eighth of generated performance alert summaries are false.

Missing alerting analysis Performance alerts can also be manually created by Performance Sheriffs when the automated tooling misses an issue. A manually created alert signals that the underlying performance change was not captured by Perfherder and that the regression was significant enough to be noticed and flagged by a human engineer. Notably, a small observed rate of such human-detected misses may plausibly understate the true prevalence of undetected issues, particularly when practitioners place high trust in the automated detection system [60–62]. Our analysis reveals that 6.8% of alert summaries contain at least one manually created alert, suggesting that missed detections, while infrequent, remain a non-trivial concern.

3.4 Chapter Summary

This chapter presented an in-depth investigation of Mozilla's performance engineering workflow, combining data collection from live Mozilla systems with a structured analysis of the alerting process. We described the two-phase methodology used to collect and analyze performance alerts, time series measurements, and bug reports spanning a one-year period across five repositories, yielding a dataset of 5,655 performance time series, 17,989 alerts,

[7] https://bugzilla.mozilla.org/show_bug.cgi?id=1977385

and 3,912 alert summaries. We detailed the core concepts underpinning Mozilla's workflow, including signature time series, alert summaries, backfilling, and retriggering, and described the four-step change point detection method at the heart of Perfherder's automated anomaly detection.

Our preliminary analysis of the collected data revealed two non-trivial limitations of the current approach. First, at least 12.47% of alert summaries are classified as false, representing wasted triage effort and a potential source of alert fatigue among Performance Sheriffs. Second, 6.8% of alert summaries contain at least one manually created alert, indicating that the automated system misses a non-negligible number of genuine regressions. Together, these findings point to the alert generation step as a meaningful area for improvement. The next chapter builds on these observations and proposes a method aimed at improving the accuracy of performance anomaly detection in Mozilla's workflow.

Chapter 4

Comparing Change Point Detection Methods for Software Performance Regression

Having characterized the limitations of Mozilla's method in chapter 3, we now turn to the question of whether alternative change point detection (CPD) methods can provide more reliable performance regression detection. While Mozilla's method achieves high precision, our preliminary analysis revealed that it misses a non-trivial number of genuine regressions and still generates a meaningful share of false alerts. This motivates a systematic investigation of whether existing CPD methods from the literature can reduce false and missed alerts relative to Mozilla's method in Mozilla's performance engineering context.

To this end, we conduct a large-scale empirical study evaluating 25 CPD methods spanning offline, online, and hybrid approaches, as well as 15 ensemble configurations, on a practitioner-annotated ground-truth dataset of 174 Mozilla performance time series. A key contribution of this chapter is the construction of that ground-truth dataset itself: since Mozilla's existing alert labels cannot serve as a reliable reference, we design and deploy a dedicated annotation platform, recruit eleven experienced Mozilla engineers, and collect multi-annotator change point labels through a structured annotation process as described

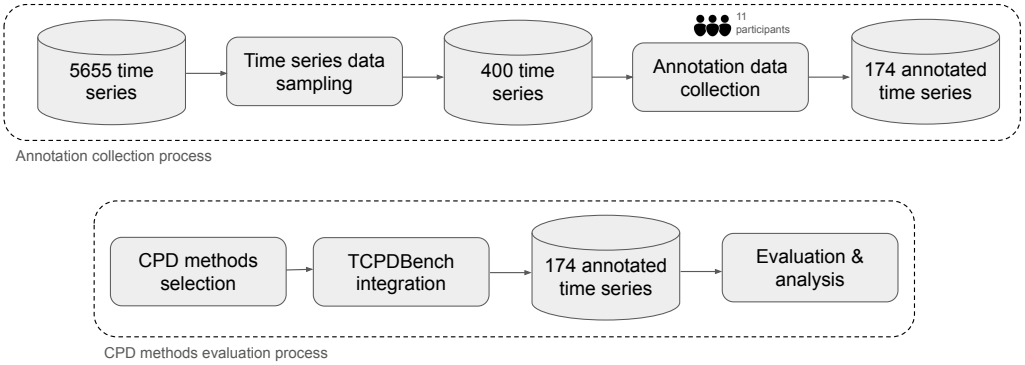


Figure 4.1: Experimental methodology overview

in Section 4.1. We then evaluate all CPD methods against this ground truth under multiple hyperparameter configurations, comparing their precision, recall, and F1-score as stated in Section 4.2. Section 4.3 concludes with a synthesis of the key findings.

4.1 Methodology

After exploring the limitations of Mozilla's method, we conduct an experimental study to explore alternative CPD methods. The goal is to determine whether CPD methods can provide greater reliability than Mozilla's method by minimizing false and/or missing alerts. Figure 4.1 illustrates the experimental methodology used to collect and annotate the data, select the alternative CPD methods, and execute the experiments.

4.1.1 Dataset Sample Selection

To evaluate the CPD techniques in our thesis, we require a reliable ground-truth dataset. Mozilla's existing alert summaries cannot be used directly for this purpose for two reasons. First, using alerts produced by Mozilla's method as a ground-truth risks confirmation bias, potentially reinforcing its assumptions and limitations. Second, the data associated with *backfilling* and *retriggering*, two processes that launch multiple tests upon finding a potential alert to determine the exact culprit revision, makes it impossible to distinguish original measurements from those produced by the latter processes. Since no such gold-standard labeling exists in Mozilla's performance pipeline, we need to construct one. This requires

collecting practitioner annotations on selected time series, providing a reliable reference against which different change point detection methods can be compared. Guided by the Mozilla performance engineering team, we select time series from the two most reliable testing subsets, which are called *Speedometer3* and *TP6*, respectively. By selecting time series associated with these subsets among the entire published dataset of 5655 time series, we obtain 2851 time series. From those, we randomly sample 400 for the annotation process to maintain a 95% confidence level and a 5% margin of error.

In the performance time series dataset, some revisions are associated with multiple performance measurements (up to 20 measurements for a single revision in certain cases) while others have only one measurement per revision. To simplify the annotation process for human annotators, we represent each revision with exactly one measurement in the annotation platform. When a revision has multiple associated measurements, we aggregate them by averaging, so that each revision is displayed as a single representative value. The annotation process is described in detail in Section 4.1.4.

4.1.2 Annotation Platform

To conduct the time-series annotation process, we need a tool that allows participants to interactively explore the time series and annotate potential performance changes. For that aim, we deploy an open-source application called *AnnotationChange*[1]. This tool is a web platform created by van den Burg and Williams [21] and is used in previous studies to collect time-series annotations from participants [63, 64]. *AnnotationChange* also has a simpler but familiar interface to Mozilla's time series dashboard, which facilitates annotator onboarding by using a familiar setup. Figure 4.2 shows an example of the annotation interface. The tool offers Zooming/sliding options, enabling annotators to zoom in and out and to move in four directions within the time series plot. It also automatically manages the assignment of annotation tasks among annotators. It prioritizes time series that have already received at least one annotation and continues assigning them to additional annotators until the specified maximum number of annotators per time series is reached. The system also

[1] https://github.com/alan-turing-institute/AnnotateChange

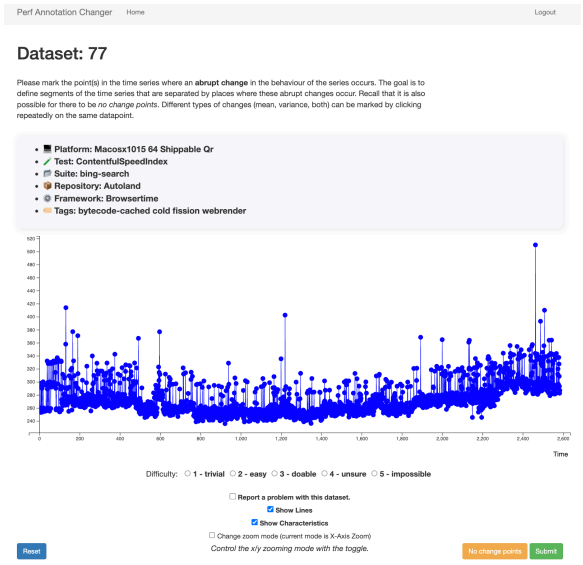


Figure 4.2: Annotation user interface example

prevents the same annotator from labeling the same time series more than once.

4.1.3 Pilot Study

Data annotation is inherently subjective, and ensuring that all participants share a consistent understanding of the task is critical for producing a reliable ground truth [65–67]. Additionally, prior to deploying the annotation tool to domain practitioners, we want to validate that the interface provides sufficient contextual information and supports the interactions needed for accurate labeling. To this end, we conduct a pilot study to refine the annotation tool and interface through iterative rounds of trials and feedback before the main annotation study begins. In this pilot study, two Mozilla practitioners and two researchers participate in a total of five improvement iterations spanning 17 days. Participant feedback is centered around three major improvements: 1) the inclusion of performance test specifications related to time series, 2) the ability to support both x-axis and y-axis zoom, and 3) the ability to support tailored tutorial content.

(1) **Performance test specifications.** Participants ask us to display the specifications of the performance related to the time series to be annotated. Therefore, for a given time series, we present its attributes, including the associated test, test suite, platform,

repository, signature number, and tags. Adding such attributes for every time series would provide enough context for the Mozilla annotator to supply an informative set of annotations per time series.

(2) **Enhanced zoom capabilities.** The platform originally had x-axis zooming capability, but it does not enable users to do y-axis zooming, which is deemed essential for participants to distinguish the exact change point in some time series.

(3) **Tailored tutorial content.** The tutorial provides background information on the study's purpose and the origin of the time series data. After each tutorial annotation task, participants are shown the expected change-point locations, along with explanations, to ensure they correctly understand the annotation procedure and interface functionality. Based on practitioner feedback, we tailor examples that gradually increase the complexity of the annotation process to check the annotators' understanding and to convey the functionality of the interface controls.

4.1.4 Data Annotation Process

We include the performance time series in the annotation platform without any prior alert-related labels. The goal of this process is to annotate these time series by identifying performance changes that should be notified to performance sheriffs. The resulting annotations will then be used as ground truth for evaluating the CPD methods. To increase the reliability of the annotated ground-truth data, we configure the annotation platform to assign five annotators for each performance time series.

Participants are recruited through an internal email sent to members of Mozilla's performance engineering team. The invitation includes a video explaining the study's motivation, the goals of the annotation process, and brief instructions for using the annotation platform. Upon registration, participants are required to complete a mandatory tutorial before annotating any of the time series used in this study.

A total of eleven Mozilla practitioners volunteered to join the data annotation process. Table 4.1 presents the demographic information of the participants, including their roles,

Table 4.1: Demographic information about the participants, including their roles, overall software engineering experience, and familiarity with the Mozilla performance workflow.

		Years of Experience		
Role	**Academic level**	**Engineering**	**Mozilla Performance**	**Perfherder**
Perf Test Engineer	Masters	10 years	7 years	7 years
Perf Sheriff & Perf Testing	Masters	13 years	13 years	5 years
SE Intern	Bachelor	2 years	0.5 years	0.5 years
Engineering Manager	College	25 years	5 years	5 years
Perf Engineer	Bachelor	24 years	7 years	7 years
Perf Engineer	Bachelor	24 years	3 years	3 years
Perf Sheriff	High School	6 years	5 years	5 years
Perf Sheriff & Perf Test Engineer	Masters	8 years	2 years	2 years
Perf Sheriff	Bachelor	6 years	5 years	5 years
Perf Sheriff & Perf Test Engineer	Bachelor	14 years	4 years	4 years
Average	—	**13.2 years**	**5.1 years**	**4.3 years**

academic backgrounds, and experience with software engineering and Mozilla's performance engineering workflow. The demographics survey was distributed after the annotation phase concluded. Out of the 11 participants, 10 completed the survey, as one participant had left the organization before the survey was circulated. Participants have extensive experience with performance engineering, with an average of 13 years of software engineering experience, five years of experience with Mozilla's performance engineering workflow, and four years of experience using *Perfherder*.

The dataset deployed in the platform initially contained 400 instances of performance time series. During the one-month annotation period, we observe that completing annotations for all 400 time series with multiple annotators would be infeasible given the time-intensive nature of the task and the restricted pool of eligible engineers. We therefore choose to concentrate annotations on a smaller subset, prioritizing the original goal of five annotators, rather than having fewer annotators on a large sample of time series. Consequently, we concentrate the annotation on 174 time series, allowing annotators to focus on a smaller set while increasing the number of annotations per time series.

4.1.5 Selecting Change Point Detection methods

Using the annotated dataset, we aim to evaluate the performance of various CPD methods. We select the CPD methods by surveying the academic literature and identifying

approaches commonly used for detecting changes in time series data [68–71]. In addition, we include Mozilla's method. For the methods that unavailable on *TCPDBench*, we implement and integrate them into *TCPDBench* [72], the benchmarking tool designed to evaluate and compare CPD techniques on time series datasets.

Integrating these methods into *TCPDBench* ensures that they can be evaluated under a consistent experimental setup. Furthermore, the extended version of *TCPDBench* developed in this thesis is fully reproducible and publicly available, allowing future research to easily reuse the implementations and replicate our evaluation [73].

We choose a broad range of CPD methods in an attempt to explore various setups and select the best for Mozilla's performance measurements. We detail the selection process per CPD method category. Table 4.2 summarizes all methods chosen in this thesis.

Offline methods. Offline CPD methods analyze the entire time series at once to identify points at which statistical properties change [74]. These methods are particularly useful when all data has already been collected, and changes need to be detected retrospectively with high accuracy [75]. We begin with the set of offline CPD methods proposed in the study introducing *TCPDBench* [21]. Following a preliminary analysis, we exclude the following methods: *Energy Change Point* [76], *Bayesian Online Change Point Detection* [77], and *Prophet* [78] due to implementation issues and lack of compatibility with our experimental setup. The final set of methods considered in our evaluation is summarized in Table 4.2. In addition, we incorporate the *E-Divisive* method, which has been used in production settings such as at MongoDB [17].

Online methods Online methods operate in a streaming setting, where observations are processed sequentially as they arrive. At each time step, the method evaluates the incoming data point in the context of previously observed data and, in some cases, reassesses earlier points within a retained history window to determine whether a change has occurred, without access to future observations [20, 77]. The objective is to detect changes as quickly as possible after they occur, such as in cases of anomaly detection in vital infrastructure

Table 4.2: Summary of Change Detection Methods

Category	Abbrev.	Full Name	Implementation	# H-param. Combos
Offline	AMOC	At Most One Change [82]	*changepoint* R [83]	126
	BinSeg	Binary Segmentation [84]	*changepoint* R [83]	252
	CPNP	Nonparametric Change Point Detection [85]	*changepoint* R [83]	28
	KCPA	Kernel Change-Point Analysis [86]	*ecp* R [87]	18
	MongoDB	E-Divisive [17]	*spa* Python [88]	12
	PELT	Pruned Exact Linear Time [89]	*changepoint* R [83]	126
	RFPOP	Robust Functional Pruning Optimal Partitioning [85]	*robustFPOP* R [90]	4
	SegNeigh	Segment Neighborhoods [91]	*changepoint* R [83]	252
	WBS	Wild Binary Segmentation [92]	*wbs* R [93]	12
	Zero	No Change Points	Custom	–
Online	CUSUM	Cumulative Sum Control Chart [94]	Custom	750
	EWMA	Exponentially Weighted Moving Average [95]	Custom	108
	Page Hinkley	Page Hinkley Test [94, 96]	*river* Python [97]	432
	ADWIN *	Adaptive Windowing [98]	*river* Python [97]	180
	BOCPD *	Bayesian Online Changepoint Detection [77]	*ocp* R [99]	81
	Shewhart	Online Shewhart Control Chart [100]	Custom	27
	KSWIN *	Kolmogorov-Smirnov Windowing [69]	*river* Python [97]	36
	CVMWIN *	Cramér-von Mises Windowing [68]	*alibi-detect* Python [101]	48
	SPRT	Sequential Probability Ratio Test [102]	Custom	27
	ODummy	Dummy Drift Detector [103]	*river* Python [97]	–
Hybrid	CVM	Cramér-von Mises Test [71]	`scipy.cramervonmises_2samp` [104]	792
	WELCH	Welch's t-Test [70]	`scipy.ttest_ind` [105]	792
	MWU	Mann-Whitney U Test [106]	`scipy.mannwhitneyu` [107]	792
	LEVENE	Levene's Test [108]	`scipy.levene` [109]	792
	KS	Kolmogorov-Smirnov Test [110]	`scipy.ks_2samp` [111]	792

***** Detection may be attributed to an earlier point within the retained observation window.

metrics [79, 80] where quick detection is needed, at the cost of being less precise than offline methods [75, 81]

Hybrid CPD Methods. In addition to the literature-selected methods, we propose a set of hybrid CPD methods inspired by Mozilla's method. Each hybrid method evaluates a candidate revision comparing two local windows of measurements, one preceding and one following the revision, using a statistical test method, which is itself a hybrid method. As shown in Figure 4.3, the statistical test and effect size measure are interchangeable components, controlled by parameters α and τ respectively. The default window sizes and magnitude threshold match those of Mozilla's method, and the p-value is set to **0.05**, following common practice [112–114].

TCPDBench integration. For offline methods, we retain those from the original replication package. We newly introduce nine online detection techniques, as well as the hybrid methods described in Section 4.1.5, which share the same parameter space as Mozilla's method (critical value, window sizes). We also replicate Mozilla's method [72] and validate it against 30 Mozilla performance time series cleaned of backfilling runs, confirming

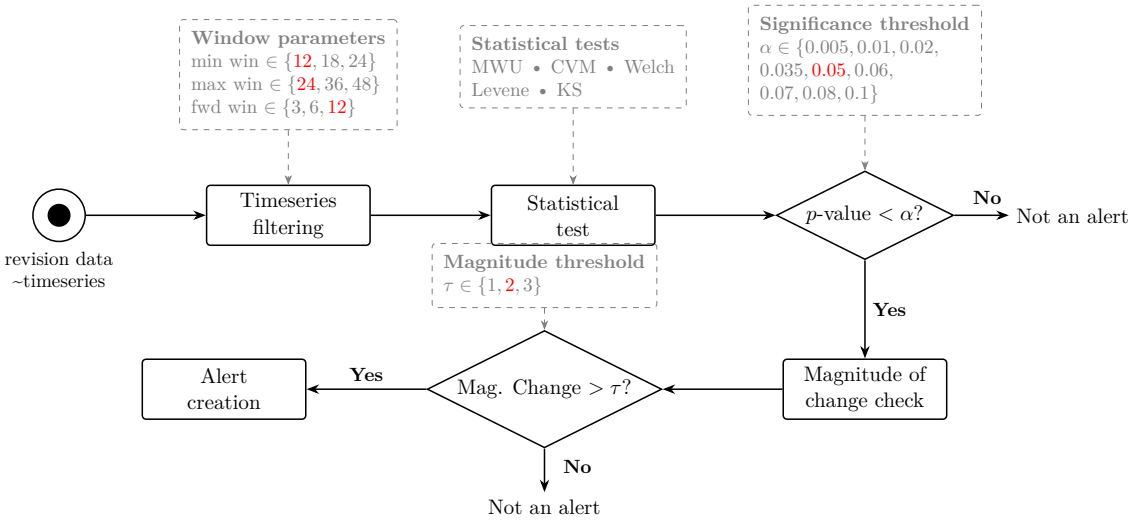


Figure 4.3: Hybrid CPD methods pipeline breakdown. We highlight the varying statistical tests and hyperparameter search space. Default values are highlighted in red.

that the results match Perfherder exactly. We also introduce five hybrid detection techniques. Finally, we configure TCPDBench to include all 174 annotated time series and their ground-truth change point labels.

4.1.6 Evaluation Metrics

Change point detection can be cast as a binary classification problem, where each time step $t \in [1, T]$ is classified as either a change point or a non-change point [20, 89]. Because change points typically constitute a small fraction of all time steps, however, standard accuracy is heavily skewed toward the majority class and is therefore uninformative. We instead evaluate methods using **precision**, the proportion of predicted change points that correspond to true change points, and **recall**: the proportion of true change points that are successfully detected. We combine both into the $\mathbf{F_1}$**-score**, which weights precision and recall equally, reflecting our goal of minimizing both false positives and false negatives.

Change point detection inherently involves some degree of ambiguity in the exact location of a change: a detection that is off by a few time steps may still be practically meaningful. It is therefore common to define a margin of error M around each true change point location, within which a predicted change point is counted as a true positive [74, 89].

We use $M = 5$ throughout.

Our setting introduces an additional challenge: each time series is annotated by multiple users, and annotators may disagree on the number or location of change points. Following the framework proposed by Martin et al. [115] and adopted by van den Burg and Williams [21], we adapt precision and recall to account for multiple ground-truth annotations. Let T_k denote the set of change points annotated by user k, for $k = 1, \ldots, K$, let X denote the set of predicted change points, and define

$$TP(A, B) = \{\tau \in A \mid \exists\, x \in B \text{ such that } |\tau - x| \leq M, \text{ with each } x \in B \text{ matched to at most one } \tau\}.$$

Letting $T^* = \bigcup_{k=1}^{K} T_k$ be the union of all annotations, precision is the fraction of predictions matched by at least one annotator

$$P = \frac{|TP(T^*, X)|}{|X|} \tag{1}$$

Recall is the average, across all annotators, of how many of their annotations were detected:

$$R = \frac{1}{K} \sum_{k=1}^{K} \frac{|TP(T_k, X)|}{|T_k|} \tag{2}$$

Averaging recall across annotators, rather than pooling all annotations, ensures that each annotator's perspective contributes equally [21].

$$F_1 = \frac{2PR}{P + R} \tag{3}$$

As commonly adopted in change point detection in time series [21, 59], we include $t = 1$ as a trivial change point in every T_k and in X to prevent division by zero when an annotator or method reports no change points,

4.1.7 Experiment Setup

Our experimental evaluation considers two distinct setups corresponding to the nature of the CPD methods: *offline* and *online*. Offline methods operate on complete time series and produce change points after observing the entire dataset, whereas online methods process data sequentially and make decisions at each time step without access to future observations. As a result, offline methods are evaluated on full time series [74], while online methods are evaluated in a streaming fashion, where data points are revealed incrementally [20, 77].

Each CPD method considered in this thesis exposes a set of hyper-parameters that control its behavior. As these parameters vary across methods (e.g., window sizes, thresholds, penalty terms), we perform hyperparameter tuning by defining, for each method, a corresponding search space of possible configurations. The size of these search spaces, expressed as the number of hyperparameter combinations explored per method, is reported in Table 4.2. This tuning process allows us to systematically explore the parameter space, assess the sensitivity of each method to its configuration, and identify settings that yield optimal performance.

To evaluate the impact of hyperparameter tuning, we report results under three complementary evaluation setups, following prior work [21]:

- **Default Configuration:** For a given method, we use its default hyperparameter configuration and compute the average metric values across all datasets.
- **Best Configuration:** For a given method, we perform hyperparameter tuning by evaluating all configurations in its search space and computing the average F1-score across all datasets for each configuration. We then select the configuration that achieves the highest average F1-score. The corresponding precision and recall values associated with this configuration are reported as *Best Precision* and *Best Recall*.
- **Upper-Bound Performance:** For a given method, we select, for each dataset independently, the hyperparameter configuration that yields the best performance. We then average these best-per-dataset metric values across all datasets for each metric separately. This setup represents an upper bound on performance when optimal

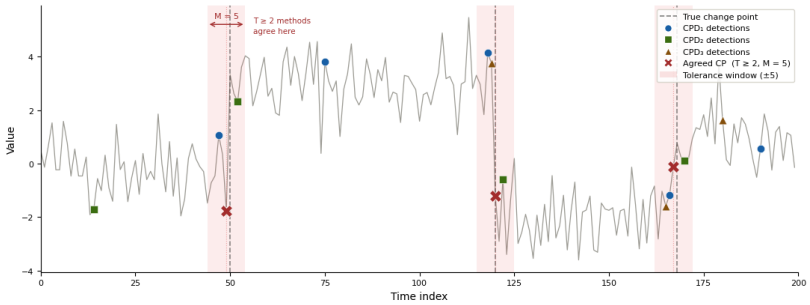


Figure 4.4: CPD agreement example: consensus threshold T of 2, tolerance margin M of 5

hyper-parameters are chosen with full knowledge of the dataset.

4.1.8 Ensemble of Methods

We want to evaluate the effectiveness of combining multiple CPD methods using an ensemble voting strategy. The intuition is that performance changes reported by multiple distinct CPD methods are more likely to be true positives. Since different methods may not flag the same event at the exact same index, we employ a *tolerance margin* M, which defines a window within which detections from different methods are treated as referring to the same event. Then, we define a *consensus threshold* T, which specifies the minimum number of methods that must agree on for a performance change to be issued. Figure 4.4 illustrates this on a simulated time series with three true change points: three CPD methods each produce their own set of detections, and a collective agreement is declared only where at least $T = 2$ methods detect a change within a tolerance window of $M = 5$, with the resulting agreed change point assigned to the average index of the contributing detections.

4.2 Results

This section presents the assessment of the data collected during the annotations process as well as the results of applying CPD methods to identify the annotated changes using offline, online, hybrid, and ensemble CPD methods. The results are compared against the performance obtained with the Mozilla's method as well as dummy baselines specific to each CPD method family.

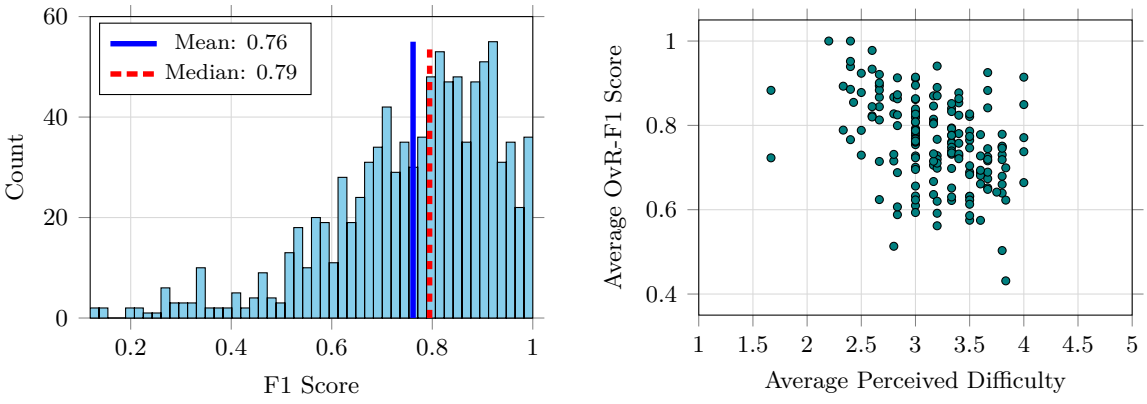
4.2.1 Annotation Data Quality Assessment

Our data annotation experiment lasted for approximately one month. In this period, eleven Mozilla annotators participated in the experiment, annotating 174 timeseries. We present some general statistics in Table 4.3. Annotators took, on average, 5.5 minutes to finish a task, with half the tasks completed in around 1 minute. A task is the action of annotating one time series by one annotator. On average, an annotator marks 6 to 7 change points per timeseries, though 61 tasks received no annotations whatsoever, meaning those timeseries were deemed free of any performance anomaly. Some annotators are notably more active than others, ranging from 1 to 2 change points per task on average to as many as 14, possibly reflecting differences in sensitivity to subtle changes. Annotators also assigned a difficulty level to each time series among five levels ranging from trivial to impossible: 69.7% were considered at least doable, while in 30.3% of cases, annotators were unsure or found it impossible to distinguish changes from noise.

Regarding annotator agreement, for every change point P flagged by an annotator, we check whether other annotators of the same timeseries agree on a change point within a 5-point radius. Out of 5781 annotated changes, 1275 (22.05%) were not agreed upon by any other annotator, while 1415 (24.48%) were agreed upon by all annotators. We define the One-versus-Rest F1 score (OvR F1) as the F1-score of an annotator's change points against the annotations of all other annotators of the same timeseries, serving as a measure of collective agreement. Figure 4.5a shows that most tasks score closer to 1 than to 0, confirming general agreement among annotators, with a mean OvR F1 of 0.76 and a median of 0.79. Figure 4.5b further shows that harder timeseries tend to yield lower OvR F1 scores (Pearson r = -0.42), suggesting that perceived difficulty negatively correlates with annotation agreement. Overall, a mean OvR F1 of 0.76 and the strong correlation between difficulty and agreement collectively indicate that the annotation data is of sufficient quality for our evaluation purposes: annotators reliably agree on unambiguous change points, and disagreements are largely confined to inherently difficult time series rather than reflecting systematic inconsistency.

Metric	Value
# of Mozilla annotators	11
# of tasks	975
Average annotations per task	6.3
# of timeseries	174
Avg. seconds per task	323
Median seconds per task	65

Table 4.3: General statistics



(a) Distribution of OvR F1-score

(b) OvR F1-score vs Difficulty

Figure 4.5: OvR F1-score related statistics.

4.2.2 Offline Methods' Results

Table 4.4 reports F1-score, precision, and recall for all offline methods across the three groups of configurations. Mozilla's method with an F1-score of 70.6%, a precision of 99.2%, and a recall of 57.7% serves as the reference point, with cells highlighted in green indicating values that exceed its corresponding metric. We also include the *Zero* baseline, which flags no change points, to help characterize the dataset. While this method trivially achieves perfect precision, its recall is notably non-zero as in 25.9% of the time series, at least one annotator indicates that no changes are present.

Across the board, CPD methods are more sensitive to performance changes than Mozilla's method. Under the *Default* configuration, nearly all methods outperform Mozilla's method's recall of 57.7%, meaning they miss fewer true regressions. However, this comes at a steep precision cost as most methods issue substantially more false alerts, with

Table 4.4: Performance Metrics for Offline Methods. We highlight in the results that surpass Mozilla's method on the corresponding evaluation metric.

	Default Configuration			Best Configuration			Upper-bound Performance		
Method	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall
Zero (Baseline)	0.407	1.000	0.268	–	–	–	–	–	–
Mozilla's method	0.706	0.992	0.577	–	–	–	–	–	–
BinSeg	0.675	0.859	0.594	0.699	0.749	0.711	0.809	1.000	1.000
CPNP	0.694	0.634	0.819	0.724	0.668	0.840	0.754	0.701	1.000
KCPA	0.618	0.864	0.513	0.728	0.697	0.830	0.835	0.957	0.844
MongoDB	0.734	0.789	0.731	0.740	0.763	0.760	0.774	0.828	0.779
PELT	0.644	0.750	0.626	0.660	0.703	0.692	0.752	0.810	1.000
RFPOP	0.701	0.613	0.884	0.703	0.616	0.884	0.714	0.629	0.949
SegNeigh	0.659	0.792	0.604	0.659	0.703	0.691	0.787	1.000	1.000
WBS	0.513	0.400	0.864	0.666	0.615	0.789	0.730	0.698	1.000
AMOC	0.541	0.922	0.406	0.542	0.922	0.407	0.582	1.000	0.433

precision values as low as 40% for `WBS` and 61% for `RFPOP`. This could reflect a fundamental behavioral difference which is that CPD methods are designed to be sensitive to data shifts, whereas Mozilla's method is deliberately calibrated for precision. In the aggregated F1-score, only `MongoDB` improves the F1-score in this setting achieving 73.4%, capturing 73.1% of all annotated regressions while keeping false alerts rate at 21%.

There are real performance gains when methods are tuned. The gains of moving from *Default* to *Best* configurations are substantial and consistent across all methods, ranging from modest improvements to substantial increases in performance. For example, `WBS`'s F1-score improves from 51.3% to 66.6%, and `KCPA`'s F1-score improves from 61.8% to 72.8%.

Once tuned, three methods achieve an overall improvement over Mozilla's method as `CPNP`, `KCPA`, and `MongoDB` methods achieve an F1-score of 72.4%, 72.8%, and 74% respectively. Among these, `MongoDB` is the most robust of the evaluated methods and outperforms the Mozilla's method's F1-score in both *Default* and *Best* configurations. Tuning generally improves recall at the cost of precision, as five out of nine methods see a drop in precision, notably with `KCPA`, which shows the most pronounced drop, moving from 86.4% to 69.7% from *Default* to *Best* configuration. Concerning recall, all methods except for *WBS* see an increase. So, performance generally improves by having a better precision at the slight cost

of a lower recall.

Upper-bound performance of CPD methods shows significant gains in recall, but most fail to reach the precision of Mozilla's method. Under *Upper-Bound Performance*, which selects the best possible configuration per time series, almost all methods surpass Mozilla's method, with *KCPA* reaching an F1-score of 83.5% and *BinSeg* reaching 80.9%. All metrics increase going from *Best* configuration to *Upper-Bound Performance*, with an improvement of 10.15% for F1-score, 18.46% for precision, and 21.12% for recall. It is worth noting that several methods achieve a precision or recall of 100% under *Upper-Bound Performance*: a perfect recall as seen in methods such as *BinSeg*, and *CPNP* reflects over-reporting, where the method flags so many change points that all true regressions are inevitably captured. Conversely, a perfect precision which is as seen in *AMOC*, *BinSeg*, and *SegNeigh* reflects under-reporting, where so few points are flagged that every detection happens to be correct.

How effective are Offline CPD methods in detecting performance changes?

Offline CPD methods show greater sensitivity to performance changes, leading to better recall (fewer missed alerts) at the cost of worse precision (more false alerts). When tuned to performance measurements, `CPNP`, `KCPA`, and `MongoDB`, show better overall performance than Mozilla's method, improving the F1-score by 2.55%, 3.12%, and 4.82% respectively.

4.2.3 Online Methods' Results

We present the results of the online method evaluation in Table 4.5. Once again, the main baseline is the performance of Mozilla's method, and we also include the *ODummy* baseline, which flags change points at fixed intervals of 300 time steps. The method, hence, serves as a sanity check for online methods operating without any statistical grounding, yielding an F1-score of 26.3%.

Out of the box, most online methods surpass the trivial baseline. Under the *Default* configuration, most methods perform better than the *ODummy* baseline in terms

Table 4.5: Performance Metrics for Online Methods. For colour interpretation, see Table 4.4. We highlight in the results that surpass Mozilla's method on the corresponding evaluation metric.

	Default Configuration			Best Configuration			Upper-Bound Performance		
Method	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall
ODummy (Baseline)	0.263	0.325	0.287	–	–	–	–	–	–
Mozilla's method	0.706	0.992	0.577	–	–	–	–	–	–
ADWIN	0.272	0.285	0.310	0.299	0.384	0.285	0.485	0.617	0.472
CVMWIN	0.203	0.148	0.497	0.225	0.155	0.632	0.319	0.245	0.770
BOCPD	0.633	0.577	0.803	0.671	0.731	0.677	0.799	0.939	0.987
CUSUM	0.220	0.165	0.520	0.313	0.370	0.331	0.518	0.622	0.664
EWMA	0.407	1.000	0.268	0.408	1.000	0.269	0.510	1.000	0.461
KSWIN	0.281	0.286	0.334	0.302	0.357	0.304	0.455	0.574	0.475
Page Hinkley	0.291	0.347	0.288	0.387	0.682	0.295	0.525	0.845	0.490
Shewhart	0.296	0.332	0.459	0.323	0.405	0.405	0.406	0.511	0.667
SPRT	0.205	0.147	0.512	0.226	0.137	0.979	0.249	0.166	0.979

of F1-score with the only three performing worse with F1-scores as low as 22% for CUSUM, 20.5% for SPRT, and 20.3% for CVMWIN. However, F1-score scores of most of the other methods remain close to that of the dummy baseline. This is not a sign of fundamental weakness in the underlying statistical tests, but rather a consequence of high sensitivity to parameter initialization, a pattern consistent with what we observe in offline methods. *BOCPD* is the clear exception, reaching an F1-score of 63.3% without any tuning and a recall of 80.3% already surpassing Mozilla's method's recall, meaning it catches more true regressions out of the box at the cost of lower precision.

Tuning improves results considerably, but no online method closes the gap with Mozilla's method. After tuning, all methods improve over their *Default* results, with some methods seeing dramatic gains such as *Page Hinkley* and *CUSUM*. Yet despite these gains, no online method surpasses Mozilla's method's F1-score of 70.6% under *Best* configuration. *BOCPD* remains the strongest candidate with an F1-score 67.1%, missing fewer annotated regressions as recall is 67.7%, but still issuing 26.9% of false alerts. However, *BOCPD* remains substantially lower than that of Mozilla's method. Contrary to the offline methods, tuning online methods yields better precision at the cost of recall which significantly decreases for four out of nine evaluated methods, namely for *BOCPD*, *ADWIN*, *Page Hinkley*, and *Shewhart*.

The Upper-Bound performance reveals a ceiling on tuning as even with optimal parameters, online methods cannot match Mozilla's method. Under *Upper-Bound Performance*, only one online method surpasses Mozilla's method's F1-score, which is *BOCPD* as it reaches its highest achievable F1-score of 79.9%, yet this remains below the best offline *Upper-Bound Performance* result ,which is *KCPA* at an 83.5% F1-score, indicating that the performance ceiling for online methods lies below that of their offline counterparts. This means that additional tuning effort yields diminishing returns. This could be interpreted that online methods are not held back by poor calibration, but by the inherent constraint of committing to decisions about change points as soon as data gets streamed in [20]. It is also worth noting that *ADWIN*, *KSWIN*, and *CVMWIN* retain some retrospective capacity by attributing detections to earlier positions within an observation window.

EWMA method presents an outlier. Upon investigating *EWMA*'s experiments results, it either produces no alerts at all or simultaneously floods with false positives *and* misses many true regressions, yielding F1-scores lower than issuing no alerts whatsoever. The configurations that avoid this erratic behavior are inherently conservative, explaining the perfect precision but low recall consistently observed across all configurations. This also makes *EWMA* an outlier in terms of variance across configurations. One plausible explanation is that the hyperparameter space itself, despite covering 108 distinct configurations as per Table 4.2, does not expose a regime in which *EWMA* behaves neither too conservatively nor too erratically, pointing to a fundamental sensitivity of the method to hyperparameter choices beyond the ranges explored.

Table 4.6: Performance Metrics for Hybrid Methods. For colour interpretation, see Table 4.4. We highlight in the results that surpass Mozilla's method on the corresponding evaluation metric.

Method	Default Configuration			Best Configuration			Upper-Bound Performance		
	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall
Mozilla's method	0.706	0.992	0.577	–	–	–	–	–	–
CVM	0.411	0.295	0.908	0.622	0.569	0.792	0.749	0.729	0.987
MWU	0.403	0.287	0.902	0.643	0.708	0.642	0.777	0.831	0.983
KS	0.396	0.277	0.901	0.631	0.627	0.711	0.757	0.739	0.981
LEVENE	0.369	0.335	0.544	0.493	0.567	0.517	0.658	0.762	0.854
WELCH	0.451	0.343	0.885	0.671	0.647	0.787	0.750	0.723	0.978

How effective are online CPD methods for identifying performance regressions?

Online CPD methods prioritize issuing alerts in real time, and this shows an inherent limitation in their effectiveness. Most evaluated online CPD methods have lower precision and recall, compared to Mozilla's method. The exception is `BOCPD`, which shows comparable out-of-the-box and tuned performance, showing an improvement of 17.33% in recall at a cost of 26.31% in precision loss when fine-tuned.

4.2.4 Hybrid Methods' Results

We report in Table 4.6 the results of all hybrid methods, including the Mozilla's method as the main baseline.

Out of the box, hybrid methods are more sensitive than Mozilla's method but suffer from the same precision problem as other CPD methods. Under *Default* configuration, nearly all methods already surpass Mozilla's method's recall of 57.7%, with *CVM*, *MWU*, and *KS* exceeding 90% recall. Similarly to the offline CPD methods, there is a steep precision cost, with values as low as 27.7% for *MWU* and 29.5% for *CVM*. *LEVENE* is the weakest method overall, failing to surpass Mozilla's method's recall of 54.4% and yielding a low precision of 33.5%. No hybrid method surpasses Mozilla's method's F1-score of 70.6% at *Default*.

Tuning improves F1-score consistently, but precision remains the bottleneck

for most variants. After tuning, all methods improve their F1-score over the default configuration, yet none of them surpass Mozilla's method's F1-score under the *Best* configuration. The core issue is that, despite meaningful precision gains through tuning, like in the example of *MWU*, which improve from 28.7% to 70.8%, the methods remain too "liberal" in flagging change points to achieve a competitive overall F1-score.

The Upper-Bound Performance exposes meaningful calibration headroom for hybrid methods. Under *Upper-Bound Performance*, several methods surpass Mozilla's method, with *MWU* reaching 77.7% and *CVM* reaching 74.9% F1-scores. A large gap between *Best* configuration and *Upper-Bound Performance* results is observed for most methods. For instance, *CVM* improves by 12.7 points from *Best* configuration to *Upper-Bound Performance*, which suggests that better per-series calibration strategies could substantially close the precision gap.

How effective are hybrid CPD methods for identifying performance regressions?

Most variants detect more true regressions than Mozilla's method, confirming their sensitivity to performance changes. However, their precision is too low for practical use without tuning. The results with fine-tuning suggest that the precision problem is a calibration challenge rather than a fundamental limitation of the methods.

4.2.5 Ensemble Voting Results

Approach. The results of the individual CPD methods reveal a clear tradeoff: most methods tend to produce a high false positive rate with an acceptable false negative rate, whereas Mozilla's method exhibits high precision and low recall. This complementary behaviour motivates the design of an ensemble voting framework that aggregates the outputs of multiple methods into a collective decision, enabling more robust detection than any single method alone. We apply this framework to all three families of methods evaluated in this section: offline, online, and hybrid.

To construct the voting ensemble, we select the four best-performing offline methods

Table 4.7: Voting strategies results using offline & online methods. We highlight in the results that surpass Mozilla's method on the corresponding evaluation metric. The best F1-score within each voting strategy is in **bold**.

		Offline Methods			Online Methods		
Strategy	Consensus (C)	F1	Precision	Recall	F1	Precision	Recall
Baseline	Mozilla's method	0.706	0.992	0.577	0.706	0.992	0.577
Ensemble	C=1	0.681	0.582	0.888	0.477	0.411	0.702
	C=2	0.782	0.770	0.830	**0.491**	0.812	0.386
	C=3	**0.784**	0.864	0.753	0.413	0.977	0.275
	C=4	0.754	0.940	0.661	0.408	1.000	0.269

based on their F1-score under the *Best* configuration, excluding lower-performing methods to avoid diluting the ensemble's detection quality. We adopt the same rationale for the online methods. Eventually, the offline methods participating in the voting are *BinSeg*, *CPNP*, *KCPA*, and *MongoDB*, and the online methods participating are *BOCPD*, *EWMA*, *Page Hinkley*, and *Shewhart*. For both offline and online methods, we apply a voting ensemble with equal weights, a.k.a. *Ensemble* voting. In the ensemble of hybrid methods, where the complementarity with Mozilla's method is most pronounced, we also experiment with a mode of *Mozilla or Ensemble* strategy.

Results. Tables 4.7 and 4.8 report the results of the voting strategies across all method families under increasing consensus thresholds, with Mozilla's method serving as the baseline reference. Table 4.8 presents the *Ensemble* and *Mozilla or Ensemble* strategies applied to hybrid methods, where the latter gives Mozilla's method an unconditional vote before applying the consensus threshold to the remaining methods. Table 4.7 presents the *Ensemble* voting strategy applied independently to the offline and online method families.

Offline methods form the strongest ensemble, surpassing Mozilla's method without requiring its participation. In Table 4.7, each row corresponds to a consensus level C, where a point is flagged only if at least C methods agree. Lower C values favor recall while higher ones favor precision. The offline ensemble peaks at an F1-score of 78.4% at C=3, surpassing Mozilla's method's F1-score of 70.6% by 7.8 percentage points, and achieved without including Mozilla's method in the ensemble at all. Precision and

Table 4.8: Voting strategies results using hybrid methods. We highlight in the results that surpass Mozilla's method on the corresponding evaluation metric. Voting strategies results. The best F1-score within each voting strategy is in **bold**.

Strategy	Consensus (C)	Results		
		F1	Precision	Recall
Baseline	Mozilla's method	0.706	0.992	0.577
Ensemble	C=1	0.592	0.493	0.863
	C=2	0.724	0.718	0.795
	C=3	**0.768**	0.868	0.726
	C=4	0.747	0.950	0.645
	C=5	0.680	0.983	0.552
	C=6	0.577	0.999	0.434
Mozilla or Ensemble	C=1	0.593	0.494	0.865
	C=2	0.722	0.706	0.802
	C=3	**0.772**	0.855	0.738
	C=4	0.763	0.944	0.669
	C=5	0.726	0.979	0.606

recall are well-balanced at this threshold, scoring 86.4% and 75.3% respectively, reflecting a meaningful improvement over any individual offline method. As the threshold increases to C=4, precision continues to rise to 94.0% but recall drops to 66.1%, confirming the expected tradeoff at high consensus thresholds. The ensemble benefits from the diversity of the four selected methods, where their disagreements filter out incidental detections and their agreements reinforce genuine regressions. It is important to note that offline ensembles and hybrid ones have comparable results, both peaking in terms of F1-score at 78.4% and 77.2% respectively.

Online methods do not form an effective ensemble. The online voting ensemble peaks at a modest F1-score of 49.1% at C=2, far below Mozilla's method. Even at C=1, the ensemble reaches only 47.7%, lower than any offline configuration. At C=4, precision reaches 100% but recall collapses to 26.9%, reflecting an under-reporting behavior.

Hybrid methods combined with Mozilla's method through voting produce the best balance between precision and recall. Table 4.8 shows that both voting strategies surpass Mozilla's method at moderate thresholds, with *Ensemble voting (C=3)* peaking

at F1-score of 76.8% and *Mozilla or Ensemble (C=3)* at an F1-score of 77.2%. As the agreement threshold increases, precision rises and recall falls monotonically, with F1-score peaking before declining as the precision gain no longer compensates for the recall loss. *Mozilla or Ensemble* consistently outperforms *Ensemble* voting at the same threshold, and the advantage becomes most apparent at high thresholds as *Mozilla or Ensemble (C=5)* still substantially outperforms *Ensemble (C=6)* by 14.9 percentage points in F1-score. This confirms that preserving Mozilla's method's high-precision detections unconditionally prevents the steep recall collapse that *Ensemble* voting suffers at high consensus thresholds.

Ensemble methods help overcome the limitations of individual CPD methods. Taken in isolation, hybrid methods do not surpass the best offline method *MongoDB* with an F1-score 74.0% after tuning, and their precision remains a persistent weakness. The ensemble resolves this by combining the high recall of hybrid methods with the high precision of Mozilla's method through voting, achieving a balance that no individual method reaches. Similarly, the offline ensemble demonstrates that aggregating diverse methods filters noise effectively even without Mozilla's method's participation.

4.3 Chapter Summary

This chapter presented a large-scale empirical evaluation of 25 CPD methods and 15 ensemble configurations applied to Mozilla performance time series. A core contribution of this work is the construction of a practitioner-annotated ground-truth dataset of 174 time series, assembled through a structured annotation process involving eleven experienced Mozilla engineers, which provides a reliable reference that is free from the confirmation bias inherent in using Mozilla's method's own outputs as labels.

Our evaluation across offline, online, and hybrid method families reveals a consistent pattern: most CPD methods are more sensitive to performance changes than Mozilla's method, achieving higher recall at the cost of precision. Among individual methods, offline methods are the strongest, with `MongoDB` being the most robust, surpassing Mozilla's method's F1-score in both default and tuned configurations. Online methods, constrained

by their sequential nature, fail to match Mozilla's method even after tuning, with `BOCPD` being the only competitive candidate. Hybrid methods show high sensitivity but suffer from low precision unless carefully calibrated.

Ensemble voting proves to be the most effective strategy overall. The offline ensemble peaks at an F1-score of 78.4%, surpassing Mozilla's method by 7.8 percentage points without requiring its participation, while the hybrid *Mozilla or Ensemble* strategy achieves an F1-score of 77.2% by preserving Mozilla's method's high-precision detections unconditionally. These results demonstrate that combining complementary methods through voting effectively resolves the precision-recall tradeoff that limits individual methods. The next chapter discusses the practitioners perspective on these findings and how they apply in practice.

Chapter 5

An Industrial Validation of Experimental Results

Chapter 4 established that ensemble voting strategies, particularly those built on offline and hybrid CPD methods, can meaningfully improve detection performance relative to Mozilla's method. However, empirical results alone are insufficient to determine whether a proposed solution is suitable for deployment in an industrial setting. Practitioner input is essential to validate the assumptions underlying the evaluation framework, assess the practical acceptability of the proposed methods, and identify which solutions are considered ready for integration into Mozilla's performance engineering pipeline.

To this end, as stated in Section 5.1, we conduct a practitioner validation study consisting of a workshop presentation as detailed in Section 5.2 and a structured follow-up survey targeting Mozilla's performance engineering team with details in Section 5.3. The study pursues two complementary goals. First, we seek to validate two key assumptions of our experimental evaluation in Section 5.4. Second, we present the performance of Mozilla's method alongside the hybrid CPD methods and voting strategies to collect practitioner ratings and deployment preferences in Section 5.5. Finally, Section 5.6 summarizes the findings of the chapter.

5.1 Survey Overview

We present the method and results of our practitioner validation study, which consists of a workshop and a survey study to present the results of our experimental comparison of CPD methods to Mozilla's performance engineering team. Participants are asked to participate in a survey study that has two main complementary goals:

(1) **Validate parameters of the experimental study:** In our evaluation study, we assume two important parameters: (a) the applicability of the tolerance margin of $M = 5$ revisions used when matching predicted change points to ground-truth annotations, and (b) the importance of false positives and false negatives in the CPD result comparison through the F1 score.

(2) **Validating the proposed solutions:** We present the performance of Mozilla's method alongside the hybrid CPD methods and voting strategies. Given the comparable performance of the offline and hybrid voting methods, we opt to proceed with hybrid methods as they are easier to integrate and follow the same hybrid structure of the Mozilla's method. Therefore, our survey focuses on validating the feasibility of the hybrid solutions, focusing on the two most promising voting strategies.

We present the core findings of our study during a scheduled Mozilla team meeting to a group of 21 practitioners.

The presentation cover four topics: (i) Mozilla's method and its limitations, (ii) the practitioner-annotation baseline and evaluation framework, (iii) the hybrid CPD methods and their results, and (iv) the two voting strategies. To ensure broad and informed participation, we prepare two supporting artifacts: a structured follow-up survey, circulated through a

channel with **91 members**, which includes screenshots and descriptions from the presentation, and a report detailing the key concepts and results.

Table 5.1: Summary of the follow-up survey questions and answer formats.

Category	Question	Answer Type
Background	**Q1.** Do you consent to participate in this follow-up survey?	Tick box
	Q2. What roles do you have at Mozilla?	Free text
	Q3. How many years have you been working in software development?	Free text
	Q4. How many years have you been working in a performance-related area at Mozilla?	Free text
	Q5. For how many years have you been using Perfherder?	Free text
	Q6. Did you take part in the workshop presentation (Feb 19, 2026)?	Yes / No
Research assumptions validation	**Q7.** The tolerance margin of 5 points is acceptable when evaluating detection methods.	Likert scale (1–5)
	Q8. (Optional) Please elaborate on the reasons for your answer above.	Free text
	Q9. A reliable detection system should minimize false and missing alerts equally.	Likert scale (1–5)
	Q10. Minimizing false alerts should be prioritized due to validation effort.	Likert scale (1–5)
	Q11. Minimizing missing alerts should be prioritized due to user impact.	Likert scale (1–5)
	Q12. (Optional) Please elaborate on the reasons for your answer above.	Free text
Result validation	**Q13.** The current method's performance in minimizing false alerts is acceptable.	Likert scale (1–5)
	Q14. The current method's performance in minimizing missing alerts is acceptable.	Likert scale (1–5)
	Q15. (Optional) Please elaborate on your assessment of the current method's performance.	Free text
	Q16. How do you grade the overall performance of the statistical methods presented?	MOS scale
	Q17. How do you grade the overall performance of the voting strategies presented?	MOS scale
	Q18. What group of techniques shows adequate performance for deployment on Perfherder?	Multiple selection
	Q19. (Optional) Please elaborate on your answer above.	Free text
Other	**Q20.** Would you like to suggest, question or comment anything related to this study?	Free text
	Q21. Would you like to be notified about a pre-print of the study?	Free text (email)

5.2 Survey Instrument Design

The survey comprises 21 questions organized into four categories (Table 5.1), with an estimated completion time of 10 to 15 minutes. The Background category (Q1–Q6) collects demographic information and consent. The Research Assumptions Validation category (Q7–Q12) asks practitioners to rate two key assumptions: the tolerance margin and the equal weighting of false and missing alerts. The Result Validation category (Q13–Q19) asks practitioners to grade Mozilla's method, the CPD methods, and voting strategies. The Other category (Q20–Q21) collects open-ended feedback and optional contact information. Each category includes an optional free-text field for elaboration, and only the consent question is mandatory.

The survey employs three rating methodologies:

- **Likert-scale agreement questions** for statements about research assumptions and the acceptability of Mozilla's method performance [116].
- **MOS-style quality-assessment questions**, where practitioners rate each technique on a five-point scale ranging from *Poor* to *Excellent* [117]. In this survey, we asked participants to rate the CPD methods and voting strategies.
- **Optional free-text boxes** accompany each question group so that respondents can elaborate on their ratings.

5.3 Survey Demographics

Participants have one week to fill out the survey, and we receive responses from 12 practitioners, nine of whom attended the presentation. Table 5.2 presents their demographic profiles. On average, respondents have 11.9 years of software engineering experience, 4.3 years of Mozilla performance engineering experience, and 4.9 years of Perfherder experience. The participant pool mirrors that of the annotation study (Table 4.1), composed predominantly of Performance Sheriffs and Engineers, the practitioners who interact daily with Perfherder.

Table 5.2: Demographic information about the survey participants.

ID	Role	Years of experience		
		Engineering	**Performance**	**PerfHerder**
P1	Performance Tech Lead	20	8	16
P2	Performance Engineer	27	7.5	7.5
P3	Engineering Manager	20	7	7
P4	Performance Tools Engineer	7	1	0
P5	JavaScript Engine Developer	12	7	7
P6	Engineer	12	0	0
P7	Performance Sheriff & Performance Test Engineer	15	4	4
P8	Performance Sheriff	7	5	5
P9	Performance Sheriff	6	5	5
P10	Performance Sheriff and Performance Test	7	2	2
P11	performance test/tools engineer	8	4	4
P12	Performance Test Engineer	2	1	1
	Average	**11.9**	**4.3**	**4.9**

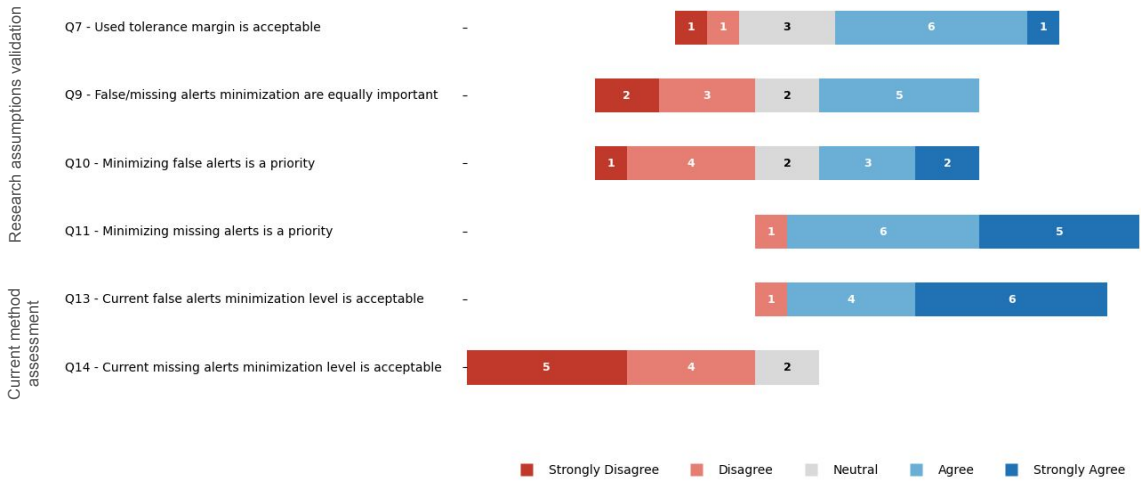


Figure 5.1: Likert scale questions results

5.4 Results on Research Assumptions

Validity of the Tolerance Margin. We use the tolerance margin of $M = 5$ to evaluate the predicted change points against ground-truth annotations. *Q7* in Figure 5.1 shows that the majority of respondents find this margin acceptable: 7 out of 12 agree or strongly agree that a margin of five points is acceptable in practice. Most practitioners consider a window of 5 revisions to be a reasonable operational tolerance for evaluating detection accuracy within Perfherder's workflow, and that a smaller margin would be considered overly strict and potentially unfair in practice.

> *"A sensitivity analysis with different margin values would help justify this choice. But in the given example, even human annotators could disagree. Expecting a perfect match would be too harsh."* — P11

Validating Penalization-Related Assumptions The second assumption concerns the equal weighting of false positives (false alerts) and false negatives (missing alerts) in the F1 score. Practitioners express more nuanced views on this assumption, captured through *Q9* to *Q11*.

Respondents are split when asked to agree that both false alerts and missing alerts are equally important (Q9). This balance suggests that practitioners hold no strong consensus on this trade-off, with some viewing it as context-dependent. However, when asked to assess each error type independently, responses to *Q10* and *Q11* reveal a more pronounced asymmetry in practitioners' priorities. Minimizing missing alerts (*Q11*) is strongly endorsed, with eleven out of twelve respondents agreeing or strongly agreeing with the emphasis on higher technique recall.

> *"If a compromise needs to be done between the two, there should be no missed alerts, since that is what finally impacts the end user."* — P7

Another practitioner introduces an important nuance, distinguishing between the severity of individual regressions:

> *"Missing a major regression is a big deal. Missing a tiny regression is less important, and may be overall better than spending a lot of time investigating false alarms."* — P5

In contrast, minimizing false alerts yields more divided opinions (*Q10*), with participants equally split on agreeing and disagreeing with the premise of increasing the technique's precision.

This tension reflects the inherent difficulty of the trade-off and helps explain the seemingly neutral stance in *Q9* alongside the stronger preference patterns observed in *Q10* and *Q11*. Nevertheless, the cost of false alerts is not dismissed entirely.

> *"False alerts come with significant cost and reduce confidence in the system, which is problematic."* — P1

Assessment of the Mozilla's method performance. The higher focus on minimizing false negatives is also reflected in the practitioners' assessment of Mozilla's method. Respondents are largely satisfied with its ability to minimize false alerts (*Q13*), with ten out of twelve agreeing or strongly agreeing, a sentiment well-grounded in Mozilla's method's 99.2% precision reported in Section 4.2. However, Mozilla's method's handling of missing alerts (*Q14*) is rejected as nine out of twelve respondents disagree or strongly disagree with being acceptable in practice. Multiple practitioners point directly to the ∼42% miss rate implied by this recall figure.

> *"Current method has a number of missed alerts that almost equals the number of correct alerts. If there are regressions among those missed alerts, it's a very high number."* — P7

> *"The current method is great at avoiding false alerts, but that comes at the cost of missing real regressions. A 42% change slipping through undetected is a clear sign the system is too conservative."* — P11

Validating Research Assumptions Practitioners strongly lean toward minimizing missed alerts (11 out of 12 endorse this), driven by direct user impact, though the cost of false alerts is acknowledged, as it erodes engineering confidence. Finally, practitioners approve of Mozilla's method's near-perfect precision (99.2%) but strongly agree that the method's recall (57.7%) should be improved.

5.5 Results on the Perception of CPD methods

Ratings of Proposed Methods. The left subfigure of Figure 5.2 reports the MOS ratings collected in *Q16* and *Q17* for the standalone statistical methods and voting strategies, respectively. The voting strategies (*Q17*) are rated considerably more favorably than standalone methods. *Ensemble voting (C=3)* receives the highest mean score across all evaluated techniques, with 8 respondents rating it *Good* and 2 *Excellent*, consistent with its strong recall improvement over the baseline. *Mozilla or Ensemble voting (C=3)* and *Mozilla or Ensemble voting (C=4)* follow closely, both receiving predominantly *Good* ratings, reflecting appreciation for strategies that recover missed detections while preserving much of Mozilla's method's precision. *Ensemble voting (C=4)* received a relatively lower rating, a profile that practitioners appear to find overly cautious, given the operational priority they place on minimizing missed regressions.

Among the standalone methods, ratings vary more than the voting strategies. *WELCH* earns a mean MOS score of 3.18, comparable to *Ensemble voting (C=4)*. However, responses concentrate around *Fair* and *Good* with no *Excellent* ratings, and its best-configuration F1-score of 67.1% still falls short of Mozilla's baseline of 70.6%. *KS* and *MWU* are rated predominantly *Fair*, reflecting limited confidence in methods that improve recall without a proportional gain in overall F1-score. *LEVENE* receives the lowest ratings (2.18 out of 5) among all evaluated techniques, consistent with its F1-score of only 49.3% and recall of 51.7%, making it the weakest individual method on both dimensions.

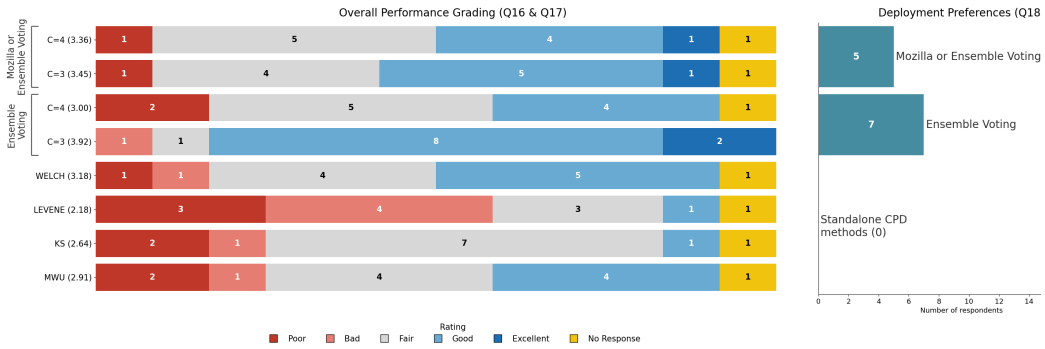


Figure 5.2: MOS Performance Grading (Q16 & Q17) and Deployment Preferences (Q18)

Deployment Preference. Practitioners were asked which group of techniques they consider adequate for deployment on Perfherder as shown in the right subfigure of Figure 5.2. Respondents clearly converge on voting strategies: seven select *Ensemble voting* and five select *Mozilla or Ensemble voting*, while no respondent endorses any standalone CPD method. Two respondents do not provide any response about their preference. The free-text responses illustrate some reasoning behind these preferences, as practitioners emphasize the importance of a technique with a better recall:

> *"I believe Mozilla's top priority should be detecting as many regression alerts as possible. This means minimizing missed alerts, even if it results in more false positives. Sheriffs are responsible for evaluating these alerts, confirming whether they are valid regressions or marking them as invalid. Although this may increase their workload, it ultimately leads to better performance quality and a more reliable product."* — P10

Indeed, relative to Mozilla's method, the ensemble strategy gains 27.9 percentage points in recall at the cost of a 13.8% reduction in precision, a trade-off that most practitioners consider acceptable given the operational priority placed on minimizing missed regressions.

> *"I favor systems which lead to higher correct alerts. For dealing with the false positives, we can improve our tooling, technology, and process to minimize the cost."* — P2

Practitioner's perspective on CPD results in practice. Voting strategies receive substantially higher practitioner ratings and endorsement for deployment than standalone methods, with *Ensemble voting (C=3)* earning the strongest endorsement across all evaluated techniques. Practitioners accept the precision trade-off that voting strategies introduce, viewing the significant gains in recall as essential given the operational cost of missed regressions.

5.6 Chapter Summary

This chapter presented a practitioner validation study designed to assess the research assumptions and proposed solutions from the preceding chapter through the lens of Mozilla's performance engineering team. The study combined a workshop presentation delivered to practitioners with a structured follow-up survey completed by twelve of them.

On the question of research assumptions, practitioners broadly validated the tolerance margin of $M = 5$ revisions as a reasonable operational threshold. Regarding the evaluation metric, a clear asymmetry emerged: while practitioners hold no strong consensus on equal weighting of false and missing alerts, eleven out of twelve strongly endorse minimizing missing alerts as the higher priority, driven by the direct impact of undetected regressions on end users. This is reflected in their assessment of Mozilla's method, which is praised for its near-perfect precision of 99.2% but criticized for its recall of 57.7%, with nine out of twelve practitioners finding its miss rate unacceptable for production use.

On the question of proposed solutions, voting strategies received substantially stronger endorsement than standalone CPD methods. *Ensemble voting (C=3)* earned the highest ratings overall, and no respondent endorsed any standalone method for deployment. Practitioners accepted the precision trade-off introduced by ensemble voting, viewing the 27.9 percentage point gain in recall as operationally justified. Taken together, these findings confirm both the validity of our experimental framework and the practical relevance of the proposed ensemble-based solutions, providing strong motivation for their integration into Mozilla's performance engineering pipeline.

Chapter 6

Discussions and Limitations

This chapter reflects on the findings of the preceding chapters through three lenses. Section 6.1 discusses the implications of our results for both researchers and practitioners, and reports the lessons learned from deploying the proposed ensemble-based solution into Mozilla's performance engineering pipeline. Section 6.2 examines the threats to the validity of our findings across construct, internal, and external dimensions.

6.1 Discussion

We discuss the findings of our thesis work across three subsections. We first discuss the practical implications of our benchmarking results for CPD method selection and tuning, then examine how practitioners perceive the proposed improvements, and finally reflect on the limitations to consider when interpreting our conclusions.

6.1.1 Implications for Researchers

Our results highlight several methodological considerations for researchers working on CPD benchmarking and evaluation.

The Importance of Method Calibration. First, the large gap between the *Default* and *Upper-Bound Performance* configurations observed across all method families reveals that many methods underperform, not due to fundamental limitations but due to poor default

calibration. This is consistent with the findings of van den Burg and Williams [21], who report similar gaps between default and oracle configurations in their benchmark and note that methods such as KCPA and RFPOP only become competitive after tuning. These results imply that benchmarking CPD methods under fixed default parameters risks systematically misrepresenting their true capabilities, and that investing in parameter selection is critical for obtaining the best performance of these CPDs.

Accounting for inter-annotator disagreement in CPD benchmarks. A broader implication of our annotation study is that collapsing multi-annotator labels into a single ground truth discards a meaningful signal. The disagreement we observe is not random noise but reflects genuine ambiguity in performance data, as evidenced by its correlation with perceived difficulty as stated in Section 4.2.1. In our thesis work, we account for this by adopting a multi-annotator evaluation framework following Martin et al. [115]. Future CPD benchmark construction efforts should similarly model and report inter-annotator disagreement explicitly.

Survey design and response consistency Practitioners differ on whether to prioritize precision or recall in change point detection methods. When asked abstractly in *Q9* whether false and missing alerts deserve equal weight, respondents were evenly split. Yet when asked independently in *Q10* and *Q11* whether each error type should be prioritized, eleven out of twelve strongly endorsed minimizing missed alerts, suggesting a general lean toward recall in practice. One possible explanation is that respondents had already been exposed to Mozilla's method's performance profile, specifically its 99.2% precision and 57.7% recall prior to filling the survey, which may have anchored their responses toward the known weakness of the system. Regardless, this divergence highlights that practitioner priorities are not uniform. Future studies evaluating change point detection methods should therefore report not only the standard F1 score but also recall-weighted variants such as the F2 score, alongside precision-weighted alternatives, to ensure results can be meaningfully interpreted across different operational contexts and priority profiles.

6.1.2 Implications for Practitioners

Our findings carry several actionable implications for practitioners responsible for deploying and maintaining performance alerting systems.

Ensemble methods are a good starting point. The practitioner survey showed that no standalone CPD method was endorsed for deployment, whereas ensemble voting strategies were deemed as a good compromise between precision and recall. This consensus reflects the quantitative results, where the offline ensemble peaked at 78.4% F1-score, and Mozilla's method or Ensemble strategy reached 77.2%, both substantially outperforming any individual method. Practitioners should therefore treat ensemble voting not as an experimental extension but as the primary deployment candidate when upgrading existing CPD pipelines. For offline ensembles, we suggest starting with `BinSeg`, `CPNP`, `KCPA`, and `MongoDB` as the initial set of methods to experiment with, as this combination achieved the best F1-score in our evaluation. For hybrid ensembles, we recommend experimenting with all available methods except `LEVENE`, which yields the worst F1-score, precision, and recall among all hybrid methods evaluated and is therefore unlikely to contribute meaningfully to the ensemble's detection quality.

Explicitly quantify the precision-recall tradeoff. Beyond method selection, our results suggest that the precision-recall tradeoff should be treated as an explicit organizational policy rather than an implicit consequence of threshold defaults. While survey respondents were evenly split when asked abstractly whether both error types deserve equal weight, eleven out of twelve strongly endorsed minimizing missed alerts when asked independently, driven by the direct user impact of undetected regressions. Organizations should explicitly encode this priority in their alerting configurations rather than relying on default parameter choices.

The hidden issue of false negatives. Finally, practitioners should be cautious about interpreting low manual alert creation rates as evidence that their automated system is performing well. In our preliminary analysis, only 6.8% of alert summaries contained manually

created alerts suggesting missed detections, a figure that may appear reassuring in isolation. However, our evaluation of Mozilla's method reveals a recall of only 57.7%, meaning that roughly four in ten annotated regressions go undetected. This discrepancy is consistent with prior work showing that practitioners tend to under-report automation failures when they place high trust in the automated system [60–62]. The low rate of human-detected misses therefore likely understates the true prevalence of undetected regressions, and practitioners should not rely on manual creation rates alone as a proxy for detection coverage.

6.1.3 Lessons Learned From the Integration and Deployment

The integration began after consolidating the results of the practitioner survey, and remains an ongoing process. In the process, we face a series of challenges that require changes in the original chosen method. We discuss these challenges in the remainder of the section.

Constrain the methods to the same window size. For better explainability, we opt to deploy a voting system in which all methods use the same data window size, so that all individual methods analyze the same data. This design choice was motivated by practitioners contributing to the voting system deployment, who emphasize that when all methods operate on the same window, the p-values they produce are directly comparable, ensuring that the ensemble consolidates conclusions drawn from the same observations, rather than mixing results across different data ranges. It is worth noting that the results reported in this thesis and presented in the practitioner survey were obtained without this constraint, meaning that the deployed configuration differs from the evaluated one in this regard.

Restricting tolerance margin further. Although most respondents did not object to a tolerance margin of M=5 revisions, it proved too large in practice. Specifically, a senior engineer involved in the voting system integration pointed out that a margin of 5 corresponds to a full day's worth of performance measurements, which represents too wide a window given that multiple regressions can be introduced within a single day. We therefore adopted M=1 instead based on that feedback, as a wider window risks merging two closely

occurring regressions into a single alert, making it harder for teams to identify and address each issue separately.

Levene instability. For hybrid ensembles, we recommend experimenting with all available methods except `LEVENE`. Beyond yielding the weakest F1-score, precision, and recall among all hybrid methods evaluated, it exhibited implementation instability during integration testing, producing unit test failures across multiple configurations. These issues further motivated its exclusion from the deployed ensemble.

Recurring adjacent alerts. During integration, several methods began generating alerts from adjacent revisions. Upon investigation, the root cause was traced to p-values of exactly zero returned by multiple methods for consecutive revision pairs. This behavior stems from a subtle asymmetry between T-value and p-value comparison logic. In the original system, when two adjacent revisions both exceed the T-value threshold, the one with the higher T-value is selected. When switching to p-value-based comparisons, the logic is inverted: a change point is flagged when the p-value falls *below* a threshold rather than above it. Since p-values are strictly positive by definition, a returned value of exactly zero indicates numerical underflow: the computed p-value is so small that it falls below the floating-point representation limit. This caused multiple adjacent revisions to be simultaneously flagged, as the deduplication logic originally designed for T-values did not account for this edge case. Notably, this issue did occur in the experimental setup on some occasions but went undetected prior to deployment. It was addressed by retaining only the last revision among those yielding a p-value of zero, effectively applying a conservative deduplication strategy for underflow cases.

6.2 Threats To Validity

We discuss the main threats to the validity of our findings, organized along three dimensions: construct, internal, and external validity.

6.2.1 Construct Validity

Several choices in our evaluation design may affect how accurately our metrics reflect real-world detection quality.

Evaluation metric. First, the F1-score treats false positives and false negatives as equally costly, an assumption that *Q9* does not explicitly reject, though the asymmetry observed in *Q10* and *Q11* suggests practitioners may operationally prioritize recall. This asymmetry may itself reflect practitioners reasoning from familiarity with the current system's known weaknesses rather than a principled stance against equal weighting, yet the possibility remains that a recall-weighted variant such as an F-beta score could yield different method rankings.

Tolerance margin. We define a detection as correct if it falls within five revisions of a ground-truth annotation. While this choice introduces some ambiguity into what constitutes a true positive, and a stricter margin could penalize methods differently, we partially mitigate this threat by validating the margin with practitioners, seven out of twelve of whom endorsed M=5 as a pragmatic operational tolerance. Furthermore, as discussed in Section 6.1, we additionally adopted M=1 during deployment integration following practitioner feedback, providing a complementary perspective on method performance under a stricter criterion.

Subjective ground-truth. Ground-truth annotation is inherently subjective: even among eleven experienced Mozilla engineers, disagreement on the exact location or existence of a change point is possible. We partially mitigate this threat by having each timeseries annotated by five distinct practitioners on average, ensuring that the ground truth reflects a collective judgment rather than a single individual's perspective. The level of agreement observed in the annotations further supports the reliability of this ground truth: out of 5781 annotated changes, 24.48% were agreed upon by all annotators, and the mean OvR F1 score across all tasks is 0.76 with a median of 0.79 as stated in Section 4.2.1, indicating that annotators generally agree on the location of change points. Residual disagreement is most pronounced on harder timeseries, where perceived difficulty correlates negatively

with agreement (Pearson r=-0.42), suggesting that subjectivity is concentrated in genuinely ambiguous cases rather than being a systematic bias across the dataset.

6.2.2 Internal Validity

CPD Method Implementations. Several factors internal to our experimental design may affect the reliability of our results. Regarding implementation fidelity, porting methods into the TCPDBench framework introduces a risk of compatibility or configuration errors. We mitigate this for Mozilla's method by validating our replication against 30 manually cleaned time series and confirming that it reproduces Perfherder's output exactly. We also perform sanity checks on third-party algorithms to detect obvious over- or under-reporting, yet subtle implementation discrepancies in less well-documented methods cannot be fully ruled out.

Limitations on the Hyper-parameter tuning. Two related design choices limit the conclusions that can be drawn from the ensemble results. First, voting experiments are conducted using only the single best-performing configuration per method rather than searching the joint hyperparameter space across all ensemble members, meaning the true optimal ensemble configuration remains unexplored. Second, the hyperparameter search spaces differ substantially across methods, ranging from as few as four combinations for RFPOP to 792 for hybrid methods, which means methods with larger grids have a structurally higher chance of identifying a well-performing configuration. This disparity makes direct cross-method comparisons under tuned settings potentially inequitable, and some methods evaluated on restricted grids may have achieved higher F1-scores had a more exhaustive search been conducted.

6.2.3 External Validity

Several factors limit the generalizability of our findings beyond the specific context in which this thesis was conducted. The thesis work is deeply embedded in Mozilla's CI/CD workflow and tooling, and the relative performance of CPD methods may differ in organizations with different code change frequencies, testing cadences, or noise profiles.

Within Mozilla itself, the ground-truth dataset is drawn exclusively from two test subsets, Speedometer3 and TP6, selected for their reliability, meaning that detection performance on more volatile or heterogeneous Mozilla test suites may diverge from what we report. The practitioner validation is subject to a related limitation as it relies on a single workshop and follow-up survey, which, while providing valuable industrial grounding, falls short of the depth achievable through more rigorous elicitation methods such as multi-round Delphi studies or individual interviews, and its conclusions should therefore be interpreted as indicative rather than definitive.

6.3 Chapter Summary

This chapter discussed the broader implications of our findings and the practical challenges encountered during deployment. For researchers, we highlighted the importance of method calibration, the need to account for inter-annotator disagreement in CPD benchmarks, and the value of reporting recall-weighted metrics alongside standard F1-scores. For practitioners, we recommended ensemble voting as the primary deployment candidate, argued for treating the precision-recall tradeoff as an explicit organizational policy, and cautioned against interpreting low manual alert creation rates as a proxy for detection coverage. The deployment lessons revealed several real-world adjustments relative to the experimental setup, including constraining methods to a shared window size, tightening the tolerance margin, removing Levene from the ensemble due to instability, and resolving a p-value underflow issue that caused adjacent duplicate alerts. Finally, we discussed the main threats to validity, acknowledging limitations in our evaluation metric, ground-truth construction, implementation fidelity, hyperparameter search design, and generalizability beyond Mozilla's specific context.

Chapter 7

Conclusion & Future Work

In this thesis, we first recorded the baseline performance of Mozilla's method for performance regressions detection, which we then used as a benchmark for comparison. We proceeded with an extensive annotation collection phase, involving performance experts to establish a ground-truth dataset of 174 time series. This dataset enabled a rigorous CPD method evaluation where we analyzed the trade-offs between 25 statistical methods and 15 ensemble strategies. Finally, we performed a validation with practitioners, whose feedback confirmed the practical utility of our findings.

Our results demonstrate that while traditional statistical methods provide a foundation for regression detection, their effectiveness in a production environment depends on their ability to minimize false positives without sacrificing sensitivity. We found that individual methods often struggle to meet these requirements. However, ensemble voting strategies, particularly those that leverage the existing high-precision baseline, achieved a significant performance increase, reaching an F1-score of 77.2%. Practitioner validation further confirmed that ensemble voting strategies are the preferred deployment candidates, with all twelve respondents endorsing voting-based approaches over standalone methods and none selecting any individual CPD method as deployment-ready. Practitioners strongly prioritized minimizing missed regressions over false alerts, reflecting the direct user impact of undetected performance degradations. The subsequent integration effort revealed additional practical considerations, including the need for a stricter tolerance margin, the removal of

unstable methods such as Levene from the ensemble, and the enforcement of a shared window size across voting participants.

While this work provides a comprehensive benchmarking of statistical CPD methods for performance anomaly detection at Mozilla, it also opens several avenues for future research. We outline the most promising directions below.

- **Non-statistical CPD methods:** The most natural extension of this work is incorporating non-statistical CPD paradigms that are intentionally scoped out. Machine learning-based methods, such as the neural network approach by Atashgahi et al. [63], operate without explicit statistical assumptions, making them potentially more robust to the non-Gaussian, heteroskedasticity nature of browser performance data. Supervised variants could leverage Mozilla's existing human-labeled alerts as training signal which is a direct asset from the dataset that statistical methods leave unexploited. Forecasting-based approaches like Prophet [78] detect change points implicitly by flagging residuals from a learned trend model, handling seasonality and calendar effects natively, which matters for CI systems with weekday/weekend load patterns. Finally, future work could evaluate *meta-ensembles* that combine statistical and ML-based detectors.

- **Generalization beyond Mozilla:** The evaluation is grounded in Mozilla's CI pipeline, and replicating the benchmark on data from other organizations such as MongoDB's Evergreen or Meta's performance infrastructure would test whether the method rankings observed in this thesis generalize, or whether they are use case-specific.

- **Longitudinal developer feedback studies:** Beyond offline benchmarking, a longitudinal study tracking how developers respond to alerts generated by different methods measuring false positive rates, time-to-triage, and regression escape rates would provide ecological validity that offline evaluations cannot capture.

- **Stability of method rankings over time:** As Mozilla's codebase and test infrastructure evolve, the ranking of CPD methods may itself drift over time. Investigating

whether method rankings remain stable across different periods motivates adaptive method selection strategies that can respond to shifts in the underlying data distribution.

- **Issue delay effect reduction:** The current anomaly detection method employed by Mozilla relies on comparing performance measurements before and after the revision under investigation, which means that the current approach requires waiting to obtain performance measurements on later code pushes to be able to automatically analyze a given code push produced earlier. The longer a bug persists in the codebase, the more difficult and costly it becomes to fix. This is known as the *delayed issue effect* [118]. In our experimentation, the online methods, which would be a good candidate for this issue mitigation, show lower performance than the current method. However, other change point detection strategies that minimizes delayed detection could be explored such as the usage of time series foundation models.

Bibliography

[1] Gunnar Kudrjavets, Jeff Thomas, and Nachiappan Nagappan. The evolving landscape of software performance engineering. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, EASE '22, page 260—261, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450396134. doi: 10.1145/3530019.3534977. URL https://doi.org/10.1145/3530019.3534977.

[2] Fast Company. How one second could cost amazon $1.6 billion in sales, 2012. URL https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales.

[3] Google, Deloitte, and 55. Milliseconds make millions. https://www.deloitte.com/content/dam/assets-zone2/ie/en/docs/services/consulting/2023/Milliseconds_Make_Millions_report.pdf, 2018. Accessed: 2025-07-01.

[4] Mark D. Syer, Weiyi Shang, Zhen Ming Jiang, and Ahmed E. Hassan. Continuous validation of performance test workloads. *Automated Software Engineering*, 24(1): 189—231, March 2017. ISSN 1573-7535. doi: 10.1007/s10515-016-0196-8. URL https://doi.org/10.1007/s10515-016-0196-8.

[5] Shravan Pargaonkar. A comprehensive review of performance testing methodologies and best practices: Software quality engineering. *International Journal of Science and Research*, 12(8):2008—2014, 2023. doi: 10.21275/SR23822111402.

[6] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. What's wrong

with my benchmark results? studying bad practices in jmh benchmarks. *IEEE Transactions on Software Engineering*, 47(7):1452—1467, 2021. doi: 10.1109/TSE.2019.2925345.

[7] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 265—276, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584065. doi: 10.1145/1508244.1508275. URL https://doi.org/10.1145/1508244.1508275.

[8] David Georg Reichelt and Stefan Kühne. How to detect performance changes in software history: Performance analysis of software system versions. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 183—188, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356299. doi: 10.1145/3185768.3186404. URL https://doi.org/10.1145/3185768.3186404.

[9] Christoph Laaber, Joel Scheuner, and Philipp Leitner. Software microbenchmarking in the cloud. how bad is it really? *Empirical Software Engineering*, 24(4):2469—2508, August 2019. ISSN 1573-7616. doi: 10.1007/s10664-019-09681-1. URL https://doi.org/10.1007/s10664-019-09681-1.

[10] Iman Kohyarnejadfard, Daniel Aloise, Michel R. Dagenais, and Mahsa Shakeri. A framework for detecting system performance anomalies using tracing data analysis. *Entropy*, 23(8), 2021. ISSN 1099-4300. doi: 10.3390/e23081011. URL https://www.mdpi.com/1099-4300/23/8/1011.

[11] Jinfu Chen and Weiyi Shang. An exploratory study of performance regression introducing code changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, page 341—352, 2017. doi: 10.1109/ICSME.2017.13.

[12] Moritz Beller, Hongyu Li, Vivek Nair, Vijayaraghavan Murali, Imad Ahmad, Jürgen

Cito, Drew Carlson, Ari Aye, and Wes Dyer. Learning to learn to predict performance regressions in production at meta. In *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*, page 56—67, 2023. doi: 10.1109/AST58925.2023.00010.

[13] Xutong Liu, Yufei Zhou, Yutian Tang, Junyan Qian, and Yuming Zhou. Human-in-the-loop online just-in-time software defect prediction: What have we achieved and what do we still miss? *Science of Computer Programming*, 244:103296, 2025. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2025.103296. URL https://www.sciencedirect.com/science/article/pii/S0167642325000358.

[14] Husari, Abdulrahman and Taherpour, Sepehr. Enhanced Techniques for Detecting Performance Abnormalities in Software Quality Assurance Processes, 2024. ISSN 1650-2884. Student Paper.

[15] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. Accurate modeling of performance histories for evolving software systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 640—652, 2019. doi: 10.1109/ASE.2019.00065.

[16] Donghun Lee, Sang K. Cha, and Arthur H. Lee. A performance anomaly detection and analysis framework for dbms development. *IEEE Transactions on Knowledge and Data Engineering*, 24(8):1345–1360, 2012. doi: 10.1109/TKDE.2011.88.

[17] David Daly, William Brown, Henrik Ingo, Jim O'Leary, and David Bradford. 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*, ICPE '20, page 67–75, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450369916. doi: 10.1145/3358960.3375791. URL https://doi.org/10.1145/3358960.3375791.

[18] Mozilla. Mozilla's perfherder dashboard, 2026. URL https://treeherder.mozilla.org/perfherder/alerts/.

[19] Student. The probable error of a mean. *Biometrika*, 6(1):1—25, 1908. ISSN 00063444, 14643510. doi: 10.2307/2331554. URL http://www.jstor.org/stable/2331554.

[20] Samaneh Aminikhanghahi and Diane Cook. A survey of methods for time series change point detection. *Knowledge and Information Systems*, 51(2):339—367, 05 2017. doi: 10.1007/s10115-016-0987-z.

[21] Gerrit J. J. van den Burg and Christopher K. I. Williams. An evaluation of change point detection algorithms, 2020. URL https://arxiv.org/abs/2003.06222.

[22] Mohamed Bilel Besbes, Diego Elias Costa, Suhaib Mujahid, Gregory Mierzwinski, and Marco Castelluccio. A dataset of performance measurements and alerts from mozilla (data artifact). In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*, ICPE '25, page 16–20. ACM, May 2025. doi: 10.1145/3680256.3721973. URL http://dx.doi.org/10.1145/3680256.3721973.

[23] Connie Smith and Lloyd Williams. New book - performance solutions: A practical guide to creating responsive, scalable software. volume 20, pages 355–358, 12 2001.

[24] David Daly. Creating a virtuous cycle in performance testing at mongodb. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '21, page 33—41, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450381949. doi: 10.1145/3427921.3450234. URL https://doi.org/10.1145/3427921.3450234.

[25] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 409–425, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/maricq.

[26] David Reichelt and Stefan Kühne. How to detect performance changes in software history: Performance analysis of software system versions. pages 183–188, 04 2018. doi: 10.1145/3185768.3186404.

[27] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 60–71, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568232. URL https://doi.org/10.1145/2568225.2568232.

[28] Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1012–1021, 2013. doi: 10.1109/ICSE.2013.6606651.

[29] Henrik Ingo. 8 years of optimizing apache otava: How disconnected open source developers took an algorithm from n3 to constant time, 2025. URL https://arxiv.org/abs/2505.06758.

[30] Zheng Li, Liam O'Brien, and Maria Kihl. Doknowme: Towards a domain knowledge-driven methodology for performance evaluation. *ACM SIGMETRICS Performance Evaluation Review*, 43(4):23–32, February 2016. ISSN 0163-5999. doi: 10.1145/2897356.2897360. URL http://dx.doi.org/10.1145/2897356.2897360.

[31] Kyle Erf. Evergreen continuous integration: Why we reinvented the wheel. https://engineering.mongodb.com/post/evergreencontinuous-integration-why-we-reinvented-the-wheel, 2016. Blog Post.

[32] Matt Fleming, Piotr Kołaczkowski, Ishita Kumar, Shaunak Das, Sean McCarthy, Pushkala Pattabhiraman, and Henrik Ingo. Hunter: Using change point detection to hunt for performance regressions, 2023. URL https://arxiv.org/abs/2301.03034.

[33] Vincenzo Ferme and Cesare Pautasso. Towards holistic continuous software performance assessment. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, page 159–164, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348997. doi: 10.1145/3053600.3053636. URL https://doi.org/10.1145/3053600.3053636.

[34] Mejbah Alam, Justin Gottschlich, Nesime Tatbul, Javier Turek, Timothy Mattson, and Abdullah Muzahid. A zero-positive learning approach for diagnosing software performance regressions. In *Advances in Neural Information Processing Systems*, volume 32, pages 11627–11639, Red Hook, NY, USA, 2019. Curran Associates Inc. doi: 10.5555/3454287.3455330.

[35] Michèle Basseville and Igor Nikiforov. *Detection of Abrupt Change Theory and Application*, volume 15. 04 1993. ISBN 0-13-126780-9.

[36] Alexey Artemov and Evgeny Burnaev. Detecting performance degradation of software-intensive systems in the presence of trends and long-range dependence. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, pages 29–36, 2016. doi: 10.1109/ICDMW.2016.0013.

[37] Zeynab Chitsazian, Saeed Sedighian Kashi, and Amin Nikanjam. Detecting concept drift for the reliability prediction of software defects using instance interpretation, 2023. URL https://arxiv.org/abs/2305.16323.

[38] Luan Pham, Huong Ha, and Hongyu Zhang. Baro: Robust root cause analysis for microservices via multivariate bayesian online change point detection, 2024. URL https://arxiv.org/abs/2405.09330.

[39] David S. Matteson and Nicholas A. James. A nonparametric approach for multiple change point analysis of multivariate data. *Journal of the American Statistical Association*, 109(505):334—345, 2014. doi: 10.1080/01621459.2013.849605. URL https://doi.org/10.1080/01621459.2013.849605.

[40] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. Detecting performance anomalies in cloud platform applications. *IEEE Transactions on Cloud Computing*, 8(3):764–777, 2020. doi: 10.1109/TCC.2018.2808289.

[41] Roman Garnett, Michael A. Osborne, and Stephen J. Roberts. Sequential bayesian prediction in the presence of changepoints. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 345–352, New York,

NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585161. doi: 10.1145/1553374.1553418. URL https://doi.org/10.1145/1553374.1553418.

[42] Dmitry Duplyakin, Alexandru Uta, Aleksander Maricq, and Robert Ricci. In datacenter performance, the only constant is change, 2020. URL https://arxiv.org/abs/2003.04824.

[43] Raghu Ramakrishnan and Arvinder Kaur. Technique for detecting early-warning signals of performance deterioration in large scale software systems. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, page 213—222, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344043. doi: 10.1145/3030207.3044533. URL https://doi.org/10.1145/3030207.3044533.

[44] Jinfu Chen, Weiyi Shang, and Emad Shihab. Perfjit: Test-level just-in-time prediction for performance regression introducing commits. *IEEE Transactions on Software Engineering*, 48(5):1529—1544, 2022. doi: 10.1109/TSE.2020.3023955.

[45] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th International Conference on Software Engineering (ICSE)*, page 156—166, 2012. doi: 10.1109/ICSE.2012.6227197.

[46] Yutong Zhao, Lu Xiao, Xiao Wang, Lei Sun, Bihuan Chen, Yang Liu, and Andre B. Bondi. How are performance issues caused and resolved?-an empirical study from a design perspective. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 181—192, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450369916. doi: 10.1145/3358960.3379130. URL https://doi.org/10.1145/3358960.3379130.

[47] Eliezio Soares, Daniel Alencar da Costa, and Uirá Kulesza. Continuous integration and software quality: A causal explanatory study, 2023. URL https://arxiv.org/abs/2309.10205.

[48] Omar Elazhary, Colin Werner, Ze Shi Li, Derek Lowlind, Neil A. Ernst, and Margaret-Anne Storey. Uncovering the benefits and challenges of continuous integration practices. *IEEE Transactions on Software Engineering*, 48(7):2570–2583, 2022. ISSN 2326-3881. doi: 10.1109/tse.2021.3064953. URL http://dx.doi.org/10.1109/TSE.2021.3064953.

[49] Wei Liu, Yi Wen Heng, Feng Lin, Tse-Hsun, Chen, and Ahmed E. Hassan. Mobile-upreg: Identifying user-perceived performance regressions in mobile os versions, 2025. URL https://arxiv.org/abs/2509.16864.

[50] Jinfu Chen and Weiyi Shang. An exploratory study of performance regression introducing code changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 341–352, 2017. doi: 10.1109/ICSME.2017.13.

[51] Mozilla api documentation, 2024. URL https://treeherder.mozilla.org/docs/.

[52] First visual change, 2026. URL https://firefox-source-docs.mozilla.org/testing/perfdocs/raptor-metrics.html#first-visual-change.

[53] D. Spinellis. Version control systems. *IEEE Software*, 22(5):108–109, 2005. doi: 10.1109/MS.2005.140.

[54] Mozilla's performance sheriffs, 2024. URL https://firefox-source-docs.mozilla.org/testing/perfdocs/perf-sheriffing.html.

[55] Mozilla. Autoland code repository, 2026. URL https://hg.mozilla.org/integration/autoland/.

[56] Mozilla. Mozilla beta release notes, 2024. URL https://www.mozilla.org/en-US/firefox/129.0beta/releasenotes/.

[57] Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated root cause isolation of performance regressions during software development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, page

27–38, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450316361. doi: 10.1145/2479871.2479879. URL https://doi.org/10.1145/2479871.2479879.

[58] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 409—425, USA, 2018. USENIX Association. ISBN 9781931971478. doi: 10.5555/3291168.3291198.

[59] Zhan Lyu, Thomas Bach, Yong Li, Nguyen Minh Le, and Lars Hoemke. Bipec: A combined change-point analyzer to identify performance regressions in large-scale database systems. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, page 808—819, 2024. doi: 10.1109/ICSME58944.2024.00084.

[60] Sara E. McBride, Wendy A. Rogers, and Arthur D. Fisk. Understanding human management of automation errors. *Theoretical Issues in Ergonomics Science*, 15(6): 545—577, 2014. doi: 10.1080/1463922X.2013.817625. URL https://doi.org/10.1080/1463922X.2013.817625. PMID: 25383042.

[61] Oscar Hernan Madrid Padilla, Alex Athey, Alex Reinhart, and James G. Scott. Sequential nonparametric tests for a change in distribution: An application to detecting radiological anomalies. *Journal of the American Statistical Association*, 114 (526):514—528, 2019. doi: 10.1080/01621459.2018.1476245. URL https://doi.org/10.1080/01621459.2018.1476245.

[62] BONNIE M. MUIR and NEVILLE MORAY. Trust in automation. part ii. experimental studies of trust and human intervention in a process control simulation. *Ergonomics*, 39(3):429—460, 1996. doi: 10.1080/00140139608964474. URL https://doi.org/10.1080/00140139608964474. PMID: 8849495.

[63] Zahra Atashgahi, Decebal Constantin Mocanu, Raymond Veldhuis, and Mykola Pechenizkiy. Memory-free online change-point detection: A novel neural network approach, 2022. URL https://arxiv.org/abs/2207.03932.

[64] Stefan Volz, Martin Storath, and Andreas Weinmann. Degrees-of-freedom penalized piecewise regression. *Information and Inference: A Journal of the IMA*, 14(1):iaaf002, 02 2025. ISSN 2049-8772. doi: 10.1093/imaiai/iaaf002. URL https://doi.org/10.1093/imaiai/iaaf002.

[65] Johann Laux, Fabian Stephany, and Alice Liefgreen. Improving task instructions for data annotators: How clear rules and higher pay increase performance in data annotation in the ai economy, 2024. URL https://arxiv.org/abs/2312.14565.

[66] Sonja Schmer-Galunder, Ruta Wheelock, Scott Friedman, Alyssa Chvasta, Zaria Jalan, and Emily Saltz. Annotator in the loop: A case study of in-depth rater engagement to create a bridging benchmark dataset, 2024. URL https://arxiv.org/abs/2408.00880.

[67] Federico Ruggeri, Eleonora Misino, Arianna Muti, Katerina Korre, Paolo Torroni, and Alberto Barrón-Cedeño. Let guidelines guide you: A prescriptive guideline-centered data annotation methodology, 2024. URL https://arxiv.org/abs/2406.14099.

[68] Gordon J. Ross and Niall M. Adams. Two nonparametric control charts for detecting arbitrary distribution changes. *Journal of Quality Technology*, 44(2):102—116, 2012. doi: 10.1080/00224065.2012.11917887. URL https://doi.org/10.1080/00224065.2012.11917887.

[69] Christoph Raab, Moritz Heusinger, and Frank-Michael Schleif. Reactive soft prototype computing for concept drift streams. *Neurocomputing*, 416:340—351, 2020. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2019.11.111. URL https://www.sciencedirect.com/science/article/pii/S0925231220305063.

[70] B. L. WELCH. The generalization of 'student's' problem when several different population varlances are involved. *Biometrika*, 34(1-2):28—35, 01 1947. ISSN 0006-3444.

doi: 10.1093/biomet/34.1-2.28. URL https://doi.org/10.1093/biomet/34.1-2.28.

[71] Harald Cramér. On the composition of elementary errors. *Scandinavian Actuarial Journal*, 1928(1):13—74, 1928. doi: 10.1080/03461238.1928.10416862. URL https://doi.org/10.1080/03461238.1928.10416862.

[72] Simon Eismann. Tcpdbench, 2022. URL https://github.com/SimonEismann/TCPDBench/tree/master.

[73] Mozilla. Replication package, 2026. URL https://github.com/mozilla/REALISE-Performance.

[74] Charles Truong, Laurent Oudre, and Nicolas Vayatis. Selective review of offline change point detection methods. *Signal Processing*, 167:107299, 2020. ISSN 0165-1684. doi: https://doi.org/10.1016/j.sigpro.2019.107299. URL https://www.sciencedirect.com/science/article/pii/S0165168419303494.

[75] Iurii Katser, Viacheslav Kozitsin, Victor Lobachev, and Ivan Maksimov. Unsupervised offline changepoint detection ensembles. *Applied Sciences*, 11(9), 2021. ISSN 2076-3417. doi: 10.3390/app11094280. URL https://www.mdpi.com/2076-3417/11/9/4280.

[76] Nicholas A. James and David S. Matteson. ecp: An r package for nonparametric multiple change point analysis of multivariate data. *Journal of Statistical Software*, 62(7):1—25, 2015. doi: 10.18637/jss.v062.i07. URL https://www.jstatsoft.org/index.php/jss/article/view/v062i07.

[77] Ryan Prescott Adams and David J. C. MacKay. Bayesian online changepoint detection, 2007. URL https://arxiv.org/abs/0710.3742.

[78] Sean J. Taylor and Benjamin Letham. Forecasting at scale. *The American Statistician*, 72(1):37—45, 2018. doi: 10.1080/00031305.2017.1380080. URL https://doi.org/10.1080/00031305.2017.1380080.

[79] Sayandeep Banerjee, Bappa Basak, Sandip Mandal, Subhajit Manna, Shuvam Chakraborty, Sujata Ghatak, and Anirban Das. Real time anomaly detection in network traffic: A comparative analysis of machine learning. *International Research Journal on Advanced Engineering Hub (IRJAEH)*, 2(07):1968—1977, Jul. 2024. doi: 10.47392/IRJAEH.2024.0269. URL https://irjaeh.com/index.php/journal/article/view/309.

[80] Zhuangbin Chen, Jinyang Liu, Yuxin Su, Hongyu Zhang, Xiao Ling, Yongqiang Yang, and Michael R. Lyu. Adaptive performance anomaly detection for online service systems via pattern sketching. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 61—72, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510085. URL https://doi.org/10.1145/3510003.3510085.

[81] Yixiao Li, Gloria Lin, Thomas Lau, and Ruochen Zeng. A review of changepoint detection models, 2019. URL https://arxiv.org/abs/1908.07136.

[82] David V. Hinkley. Inference about the change-point in a sequence of random variables. *Biometrika*, 57(1):1—17, 04 1970. ISSN 0006-3444. doi: 10.1093/biomet/57.1.1. URL https://doi.org/10.1093/biomet/57.1.1.

[83] Rebecca Killick and Idris A. Eckley. changepoint: An R package for changepoint analysis. *Journal of Statistical Software*, 58(3):1—19, 2014. URL https://www.jstatsoft.org/article/view/v058i03.

[84] A. J. Scott and M. Knott. A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, 30(3):507–512, 1974. ISSN 0006341X, 15410420. URL http://www.jstor.org/stable/2529204.

[85] Paul Fearnhead and Guillem Rigaill. Changepoint detection in the presence of outliers. *Journal of the American Statistical Association*, 114(525):169—183, 2019. doi: 10.1080/01621459.2017.1385466. URL https://doi.org/10.1080/01621459.2017.1385466.

[86] Francis Harchaoui, Zaïd Moulines, Éric, and Bach. Kernel change-point analysis. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems*, NIPS'08, page 609—616, Red Hook, NY, USA, 2008. Curran Associates Inc. ISBN 9781605609492. doi: 10.5555/2981780.2981856.

[87] David S. Matteson Nicholas A. James, Wenyu Zhang. Ecp r package. https://cran.r-project.org/web/packages/ecp/index.html, 2026. R Package.

[88] MongoDB. E divisive cpd method, 2024. URL https://pypi.org/project/signal-processing-algorithms/.

[89] R. Killick, P. Fearnhead, and I. A. Eckley. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 107(500): 1590—1598, 2012. doi: 10.1080/01621459.2012.737745. URL https://doi.org/10.1080/01621459.2012.737745.

[90] guillemr. Robust-fpop, 2019. URL https://github.com/guillemr/robust-fpop.

[91] Ivan E. Auger and Charles E. Lawrence. Algorithms for the optimal identification of segment neighborhoods. *Bulletin of Mathematical Biology*, 51(1):39—54, January 1989. ISSN 1522-9602. doi: 10.1007/BF02458835. URL https://doi.org/10.1007/BF02458835.

[92] Piotr Fryzlewicz. Wild binary segmentation for multiple change-point detection. *The Annals of Statistics*, 42(6):2243—2281, 2014. ISSN 00905364. doi: 10.1214/14-AOS1245. URL http://www.jstor.org/stable/43556493.

[93] Rafal Baranowski and Piotr Fryzlewicz. Wbs r package, 2024. URL https://cran.r-project.org/web/packages/wbs/index.html.

[94] E. S. Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100—115, 1954. ISSN 00063444. doi: 10.2307/2333009. URL http://www.jstor.org/stable/2333009.

[95] James M. Lucas and Michael S. Saccucci. Exponentially weighted moving average control schemes: Properties and enhancements. *Technometrics*, 32(1):1—12, 1990.

doi: 10.1080/00401706.1990.10484583. URL https://www.tandfonline.com/doi/abs/10.1080/00401706.1990.10484583.

[96] Raquel Sebastião and José Maria Fernandes. Supporting the page-hinkley test with empirical mode decomposition for change detection. In Marzena Kryszkiewicz, Annalisa Appice, Dominik Ślęzak, Henryk Rybinski, Andrzej Skowron, and Zbigniew W. Raś, editors, *Foundations of Intelligent Systems*, page 492—498, Cham, 2017. Springer International Publishing. ISBN 978-3-319-60438-1. doi: 10.1007/978-3-319-60438-1_48.

[97] Jacob Montiel, Max Halford, Saulo Martiello Mastelini, Geoffrey Bolmier, Raphael Sourty, Robin Vaysse, Adil Zouitine, Heitor Murilo Gomes, Jesse Read, Talel Abdessalem, and Albert Bifet. River: machine learning for streaming data in python. *J. Mach. Learn. Res.*, 22(1), January 2021. ISSN 1532-4435. doi: 10.5555/3546258.3546368.

[98] Albert Bifet and Ricard Gavaldà. Learning from time-changing data with adaptive windowing. In *Proceedings of the 2007 SIAM International Conference on Data Mining*, pages 443–448, 04 2007. doi: 10.1137/1.9781611972771.42.

[99] Andrea Pagotto. ocp: Bayesian online changepoint detection, 2019. URL https://CRAN.R-project.org/package=ocp.

[100] W. A. Shewhart. Economic quality control of manufactured product. *The Bell System Technical Journal*, 9(2):364—389, 1930. doi: 10.1002/j.1538-7305.1930.tb00373.x.

[101] Seldon Technologies Ltd. Alibi detect, 2025. URL https://docs.seldon.io/projects/alibi-detect/en/latest/index.html.

[102] A. Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2):117—186, 1945. ISSN 00034851. doi: 10.1214/aoms/1177731118. URL http://www.jstor.org/stable/2235829.

[103] Heitor Murilo Gomes, Jacob Montiel, Saulo Martiello Mastelini, Bernhard Pfahringer, and Albert Bifet. On ensemble techniques for data stream regression. In *2020 International Joint Conference on Neural Networks (IJCNN)*, page 1—8, 2020. doi: 10.1109/IJCNN48605.2020.9206756.

[104] SciPy. Cramér-von mises test implementation in scipy.stats, 2025. URL https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.cramervonmises_2samp.html.

[105] SciPy. T-test implementation in scipy.stats, 2025. URL https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html.

[106] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18 (1):50—60, 1947. ISSN 00034851. doi: 10.1214/aoms/1177730491. URL http://www.jstor.org/stable/2236101.

[107] SciPy. Mann-whitney u rank test implementation in scipy.stats, 2025. URL https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html.

[108] Morton B. Brown and Alan B. Forsythe. Robust tests for the equality of variances. *Journal of the American Statistical Association*, 69(346):364—367, 1974. doi: 10.1080/01621459.1974.10482955. URL https://www.tandfonline.com/doi/abs/10.1080/01621459.1974.10482955.

[109] SciPy. Levene test implementation in scipy.stats, 2025. URL https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.levene.html.

[110] Frank J. Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68—78, 1951. doi: 10.1080/01621459.1951.10500769. URL https://www.tandfonline.com/doi/abs/10.1080/01621459.1951.10500769.

[111] SciPy. Kolmogorov-smirnov implementation in scipy.stats, 2025. URL https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ks_2samp.html.

[112] Ronald L. Wasserstein and Nicole A. Lazar. The asa statement on p-values: Context, process, and purpose. *The American Statistician*, 70(2):129—133, 2016. doi: 10.1080/00031305.2016.1154108. URL https://doi.org/10.1080/00031305.2016.1154108.

[113] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *J. Mach. Learn. Res.*, 7:1–30, December 2006. ISSN 1532-4435.

[114] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 1–10, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. doi: 10.1145/1985793.1985795. URL https://doi.org/10.1145/1985793.1985795.

[115] D.R. Martin, C.C. Fowlkes, and J. Malik. Learning to detect natural image boundaries using local brightness, color, and texture cues. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(5):530—549, 2004. doi: 10.1109/TPAMI.2004.1273918.

[116] R.A. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):5—55, 1932.

[117] Ambika Kirkland, Shivam Mehta, Harm Lameris, Gustav Eje Henter, Eva Szekely, and Joakim Gustafson. Stuck in the MOS pit: A critical analysis of MOS test methodology in TTS evaluation. In *Proceedings of the 12th ISCA Speech Synthesis Workshop (SSW2023)*, page 41—47, 2023. doi: 10.21437/SSW.2023-7. URL https://doi.org/10.21437/SSW.2023-7.

[118] Tim Menzies, William Nichols, Forrest Shull, and Lucas Layman. Are delayed issues harder to resolve? revisiting cost-to-fix of defects throughout the lifecycle. *Empirical Software Engineering*, 22(4):1903—1935, November 2017. ISSN 1573-7616. doi: 10.1007/s10664-016-9469-x. URL https://doi.org/10.1007/s10664-016-9469-x.