Titre: Title:	A Goal-Based Adaptive Tracing Method for Detecting Performance Anomalies and Anti-Patterns in Distributed Systems
Auteur: Author:	Masoumeh Nourollahi
Date:	2025
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Nourollahi, M. (2025). A Goal-Based Adaptive Tracing Method for Detecting Performance Anomalies and Anti-Patterns in Distributed Systems [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/71012/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/71012/
Directeurs de recherche: Advisors:	Michel Dagenais, Heng Li, & Naser Ezzati-Jivan
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

A Goal-Based Adaptive Tracing Method for Detecting Performance Anomalies and Anti-Patterns in Distributed Systems

MASOUMEH NOUROLLAHI
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Octobre 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

A Goal-Based Adaptive Tracing Method for Detecting Performance Anomalies and Anti-Patterns in Distributed Systems

présentée par **Masoumeh NOUROLLAHI**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Alejandro QUINTERO, président
Michel DAGENAIS, membre et directeur de recherche
Naser EZZATI-JIVAN, membre et codirecteur de recherche
Heng LI, membre et codirecteur de recherche
Maxime LAMOTHE, membre
Ferhat KHENDEK, membre externe

DEDICATION

To my beautiful family that supported me through all my ups and downs,
I love being with you . . .

ACKNOWLEDGEMENTS

I would like to begin by expressing my heartfelt gratitude to my supervisor, Prof. Michel Dagenais, whose guidance has shaped not only my research but also my perspective on life. His wisdom, kindness, and unwavering support have been a constant source of inspiration, and I truly look up to him as both a mentor and a role model.

I am equally grateful to my co-supervisor, Prof. Naser Ezzati-Jivan, for his continuous encouragement, for opening doors to countless opportunities in research, and for patiently teaching me the right way to approach both scientific challenges and personal growth.

My sincere thanks go to our research associates, Adel Belkheiri, Arnaud Fiorini, and Fariba Fateme Faraji Daneshgar, whose advice, technical expertise, and readiness to help have greatly enriched my work.

Finally, I would like to thank all my fellow lab members for their camaraderie, collaboration, and the many moments, both professional and personal, that made this journey truly memorable.

Moreover, I would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Ciena, Ericsson, and EffciOS for funding this research project.

RÉSUMÉ

Les systèmes distribués modernes, notamment ceux basés sur les architectures à microservices, constituent aujourd'hui l'épine dorsale d'un grand nombre d'applications critiques. Ces systèmes traitent quotidiennement des millions, voire des milliards, de requêtes et doivent maintenir des niveaux élevés de performance et de fiabilité. Cependant, la complexité croissante de ces architectures rend le diagnostic de problèmes de performance particulièrement difficile. Les solutions classiques de traçage, bien que puissantes, collectent souvent un volume considérable de données dont une grande partie est non pertinente pour l'analyse recherchée. Cette collecte massive entraîne une surcharge importante, augmente les coûts de stockage et de traitement, et complique la localisation des causes profondes des problèmes.

Face à ces défis, cette thèse propose un **cadre de traçage adaptatif basé sur les objectifs**, conçu pour ajuster dynamiquement la collecte de données en fonction du but précis de l'analyse. L'idée centrale est que tous les problèmes de performance ne nécessitent pas le même type d'observation et qu'un traçage intelligent, orienté par un objectif spécifique, permet de réduire le bruit, limiter la surcharge et accélérer le diagnostic.

Objectifs et questions de recherche

Le travail de recherche s'articule autour de plusieurs objectifs :

- Concevoir un mécanisme de traçage capable d'adapter dynamiquement la configuration et la granularité des données collectées en fonction du type de problème à diagnostiquer.
- Développer une méthodologie de détection efficace pour les **antipatrons de performance logicielle (SPAs)**, qui représentent des motifs récurrents de conception ou d'implémentation dégradant les performances.
- Réduire la surcharge de traçage sans compromettre la capacité à identifier les causes profondes des anomalies.
- Évaluer l'intégration de techniques d'apprentissage automatique et semi-supervisé pour automatiser la classification et l'explication des problèmes détectés.

Méthodologie

La recherche se décline en trois axes principaux :

1. Détection d'anomalies légère via TraceLens

La première contribution, *TraceLens*, repose sur l'analyse du chemin critique d'exécution des requêtes pour identifier les événements responsables des latences. Cette approche minimise la surcharge en se concentrant uniquement sur les points ayant un impact direct sur les temps de réponse, plutôt que d'analyser l'intégralité des traces collectées.

2. Traçage adaptatif basé sur les objectifs

La seconde contribution introduit un mécanisme de traçage adaptatif, guidé par une taxonomie des SPAs. Cette taxonomie permet de définir des scénarios de traçage spécifiques selon le type de dégradation recherchée, par exemple les SPAs de communication (Blob, tâches de service excessives) ou les SPAs liés à la charge. Le système ajuste automatiquement les points de traçage, la granularité et la portée de collecte pour optimiser la pertinence des données.

3. Détection automatisée des SPAs par apprentissage machine

La troisième contribution consiste à appliquer des méthodes d'apprentissage automatique, en particulier des approches semi-supervisées, pour identifier et classifier les SPAs à partir des traces collectées. Les caractéristiques extraites incluent des métriques de charge, de latence, de volume de données échangées et de fréquence d'appels. L'utilisation d'étiquetage faible permet de limiter le besoin en annotations manuelles, tout en maintenant une performance élevée de détection.

Validation expérimentale

L'évaluation a été menée sur *DeathStarBench*, un banc d'essai open source reproduisant des scénarios réalistes de microservices. Plusieurs types de SPAs y ont été injectés afin de tester la capacité du cadre proposé à :

- Détecter différents antipatrons avec précision.
- Localiser la cause racine des anomalies dans un environnement multi-services.
- Maintenir une faible surcharge de collecte, compatible avec des déploiements en production.

Les résultats expérimentaux montrent que :

- La détection atteint jusqu'à 99% de précision pour certains types de SPAs.
- La surcharge introduite par le traçage adaptatif est aussi faible que 2,7%, ce qui est significativement inférieur aux approches de traçage exhaustif.
- Le système permet une localisation plus rapide des problèmes, réduisant le volume de données à analyser de manière substantielle.

Conclusions et perspectives

Cette thèse démontre qu'un traçage adaptatif orienté objectifs est une approche viable et efficace pour le diagnostic de performance dans les systèmes distribués modernes. Les contributions proposées permettent de réduire significativement le bruit dans les données collectées, d'améliorer la précision de détection des problèmes et de limiter l'impact sur les systèmes en fonctionnement.

Les perspectives futures incluent :

- L'extension de la taxonomie et des méthodes de détection à d'autres types d'anomalies logicielles ou matérielles.
- L'intégration du cadre dans des plateformes d'observabilité complètes (par ex. OpenTelemetry).
- L'évaluation à grande échelle dans des environnements industriels et des contextes de déploiement en continu.

Mots-clés : traçage adaptatif, traçage basé sur les objectifs, antipatrons de performance, systèmes distribués, détection d'anomalies, microservices, apprentissage automatique.

ABSTRACT

Modern distributed systems generate massive volumes of trace data, making performance diagnosis costly, intrusive, and often inefficient. Traditional tracing approaches either capture excessive irrelevant information or miss critical events necessary for root cause analysis. This thesis proposes a **goal-based adaptive tracing framework** that selectively collects trace data aligned with specific diagnostic objectives, with a primary focus on detecting **Software Performance Anti-patterns** (SPAs)—recurring design and implementation flaws that degrade system performance.

The research begins with *TraceLens*, a lightweight anomaly detection method that uses execution critical path analysis to identify latency-contributing events with minimal overhead. Building on this, the work introduces an *adaptive goal-based tracing method* that leverages an SPA taxonomy to guide tracepoint activation, tracing scope, and workload selection. This approach enables a hierarchical, iterative localization of performance issues, progressively refining trace collection toward the likely root cause.

Two main phases are explored:

1. **User-space level tracing**, using statistical analysis to detect SPAs such as traffic jams, ramps, and single-user problems.
2. **Hybrid tracing**, combining user-level and kernel-level data to capture more complex communication-related SPAs, including Blob and Excessive Service Tasks (EST).

Machine learning and semi-supervised techniques are integrated to classify SPA-related anomalies with minimal manual labeling. Experimental validation on the *DeathStarBench* microservice benchmark shows that the proposed methods achieve high detection accuracy (up to 92%) while imposing as little as 2.7% overhead, making them suitable for deployment in *CI/CD pipelines* and production environments.

This work contributes a *modular, scalable, and cost-efficient performance diagnosis framework* that can adapt to multiple goals, from SPA detection to resource monitoring and anomaly analysis, offering a practical path toward smarter and lighter observability in distributed systems.

Keywords: goal-based tracing, adaptive tracing, performance anti-patterns, distributed systems, anomaly detection, critical path analysis, hybrid tracing, microservices.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

SPA	Software Performance Anti-patterns
EST	Empty-semi-trucks
API	Application Programming Interface
CTF	Common Trace Format
CPU	Centralized Processing Unit
LTTng	Linux Trace Toolkit : next generation
PID	Process Identifier
RAM	Random Access Memory

CHAPTER 1 INTRODUCTION

Modern software systems are increasingly complex, dynamic, and distributed. In such environments, diagnosing performance issues is particularly challenging, especially when relying on traditional tracing techniques that indiscriminately collect large volumes of runtime data. While tracing is essential for understanding system behavior, it often incurs significant overhead in terms of storage, processing, and runtime performance. As systems scale, the volume and granularity of trace data can become unmanageable, rendering analysis inefficient or even infeasible.

Our initial approach to addressing this challenge was to design an *adaptive tracing* mechanism that adjusts the tracing level dynamically. In the first phase of this work, we focused on detecting performance anomalies through *critical path analysis of kernel-level traces*. This enabled the detection of resource-related anomalies. However, we encountered several limitations, most notably, the lack of visibility into application-level behaviors, which reduced the effectiveness of adaptive tracing in certain cases. These limitations prompted a broader re-evaluation of both the problem space and our solution strategy.

We concluded that the key to making tracing both efficient and useful lies in explicitly defining the *goal of tracing*. This led us to propose a **goal-based tracing** paradigm. Instead of passively collecting trace data, this approach focuses on selectively capturing only the data that is relevant to a specific analytical goal, such as detecting performance problems, identifying anomalies, or understanding system behavior. By aligning trace collection with predefined diagnostic objectives, we can significantly reduce tracing overhead while improving the relevance and interpretability of collected data.

To operationalize this idea, we further narrowed our tracing goal to focus on detecting *SPAs*, recurring design and implementation flaws that degrade system performance. These SPAs provide an ideal focus for tracing, because they encode not only the manifestation of performance issues, but also their causes and potential solutions. They are well-defined, widely documented, and represent some of the most common performance pitfalls encountered in practice.

Examples of Software Tracing Analysis Goals and Corresponding Techniques:

- Kernel-level resource monitoring → Detect resource contention and low-level anomalies
- Workload-sensitive tracing → Reveal scalability and concurrency bottlenecks

- Sampling or selective instrumentation → Reduce trace cost while preserving performance insights

Consultations with our industrial partners further reinforced this focus. More than 90% of performance problems in their systems were attributed to *configuration mistakes*, often manifesting as deployment misconfigurations or system-level tuning issues. These types of problems are frequently observable through the lens of SPAs, making them a practical and impactful tracing target. By focusing on SPA detection, we aim to support early detection of performance regressions in CI/CD pipelines and enable rapid validation of application performance in deployment scenarios.

This thesis is driven by the central question:

> *Can software tracing be made more selective and goal-oriented to reduce its overhead while still providing actionable diagnostic insights?*

To answer this, we explore the feasibility of a goal-based tracing framework that activates and collects only the trace data relevant to predefined analytical objectives. Our work investigates tracing strategies that minimize cost while maintaining diagnostic utility, and evaluates how SPAs can be identified using selective, adaptive trace data. We also explore the use of statistical and machine learning methods to interpret these traces and validate our approach through controlled experiments and fault injection scenarios. Ultimately, we aim to demonstrate that goal-based tracing can support a wide range of diagnostic goals beyond anti-pattern detection, leading to more scalable and efficient observability.

1.1 Thesis Statement and Objectives

This thesis proposes a **goal-based tracing framework** designed to reduce the overhead associated with traditional software tracing by *selectively collecting trace data* aligned with specific performance analysis goals, particularly the detection of *SPAs*. This approach enables targeted, low-cost, and effective system diagnostics.

1.2 Research Approach and Contributions

The research began with an exploration of **anomaly detection**, under the assumption that focusing on behavioral outliers could help narrow down the scope of analysis. This initial phase included studying **adaptive instrumentation** and **adaptive sampling**, which

dynamically adjust the granularity of trace collection during runtime. While these methods offered flexibility, they lacked a clear alignment with specific performance concerns and often resulted in unnecessary or irrelevant data.

To better understand the trade-off between trace volume and diagnostic accuracy, we developed a lightweight anomaly detection method called **TraceLens**. The method leverages *execution critical path analysis*, which captures only the most influential sequences of execution. Since these events lie on the critical path, they contribute the most to latencies, enabling high-precision anomaly detection with significantly reduced tracing overhead. In contrast to prior approaches that rely on full-system syscall data, TraceLens selects fewer, but more meaningful, events, using high-level features such as event counts and durations. This selective approach compensates for reduced data volume by guiding performance diagnosis in the right direction, supporting effective localization of root causes. This work also helped refine our understanding of **unsupervised anomaly detection** under constrained observability, laying the groundwork for goal-based tracing.

Recognizing the limits of generic anomaly detection for structured diagnostic reasoning, we then shifted our focus toward **SPAs**, well-known design and implementation flaws that degrade performance. Examples include the **Blob** anti-pattern, where a single component becomes overly central, and **EST (Excessive Service Tasks)**, where redundant or excessive service interactions lead to overhead. This insight led to the development of a **goal-based tracing** method that selectively activates trace points relevant to identifying such SPAs, reducing overhead while enhancing diagnostic precision.

The proposed goal-based tracing framework is implemented in **two phases**:

Phase 1: User-Space Level Tracing

In the first phase, the focus is on SPAs detectable through user-space level traces. Patterns studied include *Generic performance problems, application hiccups, the ramp, continuously violated performance requirements, including traffic jam and single user problem. Traffic jams themselves include unbalanced processing and one-lane bridge.* For this phase, **statistical methods** are used to identify deviations in trace patterns and event distributions indicative of performance issues.

Phase 2: Hybrid Tracing

The second phase expands the analysis to hybrid traces, which combine distributed user-level traces with kernel/system-call level traces. This enables the detection of more complex

performance issues that arise deeper in the execution stack. During this stage, the analysis transitions from purely statistical methods to a combination of **supervised, unsupervised, and semi-supervised learning**, using both labeled and unlabeled trace data. **Blob** and **EST** SPAs serve as reference examples to demonstrate the power and flexibility of the approach.

The implementation is evaluated through **problem injection** in a controlled test environment. This allows us to simulate performance anomalies and validate the effectiveness of our tracing method under known faulty conditions. A variety of representative use cases are tested, with particular focus on high-priority performance regressions in modern service-oriented architectures.

A key strength of the proposed method is its **flexibility**. While this thesis emphasizes anti-pattern detection, the same tracing framework can be adapted to other analytical goals, including:

- **Anomaly detection**, where unexpected deviations in behavior are flagged.
- **Resource monitoring**, such as analyzing CPU, memory, and I/O usage trends.
- **Comparative analysis**, where trace data from different runs are compared to pinpoint regressions or improvements.

By decoupling the tracing mechanism from the analytical goal, this work contributes a modular, adaptive, and cost-effective approach for system performance diagnostics in complex, distributed environments.

In summary, this thesis proposes a **goal-based tracing framework** designed to reduce the overhead associated with traditional software tracing by *selectively collecting trace data* aligned with specific performance analysis goals, particularly the detection of *SPAs*. This approach enables targeted, low-cost, and effective system diagnostics and is extensible to a variety of other performance analysis tasks.

The main research objectives addressed in this work are:

- To investigate anomaly detection using unsupervised methods with minimal trace data.
- To evaluate the feasibility of detecting SPAs through selective trace collection.
- To leverage statistical and machine learning techniques for interpreting trace data and identifying performance issues.

- To validate the proposed tracing framework through controlled experiments and problem injection.
- To demonstrate the adaptability of goal-based tracing across multiple analytical goals beyond anti-pattern detection.

1.3 Thesis Outline

The remainder of this thesis is organized as follows:

- **Chapter 2** reviews related work in performance tracing, anomaly detection, and performance anti-pattern detection.
- **Chapter 3** provides an overview of the three proposed research tracks to lay the groundwork for their detailed presentation.
- **Chapters 4 to 7** present the peer-reviewed papers that constitute the core contributions of the thesis. The first paper presents a low-overhead method to detect performance anomalies. The second paper presents a goal-based tracing method, by addressing SPA detection through user-level tracing, benefiting from statistical analysis. The third paper enhances the second work by training deep learning models that can be reused to detect SPAs in CICD pipelines. This work focuses on SPAs detectable by hybrid tracing.
- **Chapter 8** discusses limitations of this research.
- **Chapter 9** concludes the thesis with a summary of findings and directions for future work.

1.4 Publications

This thesis is based on the following peer-reviewed publications:

- Nourollahi, Masoumeh; Haghshenas, Amir; and Dagenais, Michel. "TraceLens: Early Detection of Software Anomalies Using Critical Path Analysis." Companion of the 16th ACM/SPEC International Conference on Performance Engineering. 2025. [1]
- Nourollahi Masoumeh, Ezzati-Jivan Naser, Belkheiri Adel, Faraji-Daneshgar Fatemeh, and Dagenais Michel. "Software Performance Problem Localization: A Goal-based

Adaptive Tracing Approach." Submitted to: Concurrency and Computation: Practice and Experience. 2024.

- Nourollahi Masoumeh, Ezzati-Jivan Naser, Belkheiri Adel, and Dagenais Michel. "Efficient Detection of Communication-related SPAs in Microservices." submitted to the Journal of Software: Evolution and Process. 2025.

CHAPTER 2 LITERATURE REVIEW

This chapter surveys foundational research that inspires our proposed *goal-based adaptive tracing* framework, which is designed to detect SPAs in large-scale distributed systems. The literature is organized around the key building blocks necessary to realize adaptive tracing in practice.

We begin in Section 2.1 by outlining the motivation and challenges of large-scale tracing, focusing on the cost of observability and the difficulties of applying tracing in distributed and dynamic environments.

Section 2.2 introduces core tracing techniques, with an emphasis on execution tracing and its implementation at both the user and kernel levels. We also review sampling strategies (e.g., head- and tail-based methods such as Sifter) and instrumentation approaches, including static, dynamic, and adaptive schemes.

Section 2.3 turns to methods for understanding system performance. It is divided into three parts: performance modeling (e.g., queuing models, ML/DL-based models), performance prediction (analytic and non-analytic techniques), and direct performance measurement approaches. These methods contribute to understanding the system's current and expected behavior, which is essential for selectively collecting relevant trace data.

In Section 2.4, we examine how performance issues are diagnosed and localized using trace data. This includes workflow-centric tracing systems such as Dapper and Jaeger, fault localization techniques, and hybrid approaches that combine knowledge-based models with experimental validation.

Section 2.5 focuses on software performance anti-patterns, the central diagnostic goal of our tracing framework. We review taxonomy models, detailed examples of common SPAs, and recent detection techniques, including those based on statistical analysis, modeling, and machine learning.

Finally, Section 2.6 summarizes the surveyed approaches and discusses their trade-offs. We highlight current limitations in the literature and identify research gaps that our proposed framework seeks to address by integrating performance understanding with adaptive tracing control mechanisms, all guided by the goal of SPA detection.

Figure 2.1 illustrates the two primary research directions reviewed in this chapter. The first concerns performance behavior knowledge acquisition, encompassing methods to measure,

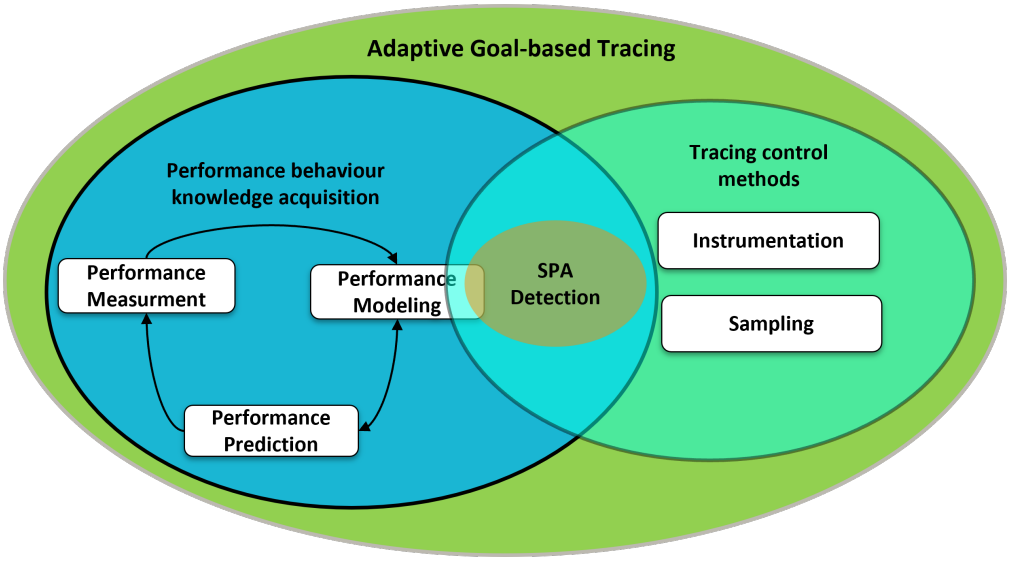

Figure 2.1 Extended view of research areas supporting goal-based adaptive tracing. The left circle includes performance behavior knowledge acquisition (measurement, modeling, prediction), while the right circle includes tracing control methods (instrumentation, sampling). Performance anti-pattern detection emerges at the intersection, informed by system behavior and enabling dynamic trace reconfiguration.

model, and predict system performance. These techniques aim to detect deviations from expected behavior and, in some cases, to localize the root causes of performance issues.

We emphasize that the detection of SPAs lies at the intersection of these two research areas. It requires insight into system behavior, via measurement and modeling, and simultaneously benefits from adaptive tracing configurations that expose or validate performance issues in context.

2.1 Motivation and Challenges in Large-Scale Tracing

Modern software architectures, built on microservices, containerized workloads, and elastic deployments, present observability challenges not found in traditional systems. Performance degradations are often emergent behaviors that span multiple services and components, and may depend on dynamic conditions such as user behavior, infrastructure variability, or workload shifts. In these environments, identifying the root cause of a performance issue is particularly difficult. Distributed tracing provides end-to-end visibility into transactions by capturing causally related spans, enabling the comparison of anomalous traces against healthy ones in terms of structure, timing, and flow.

Observability refers to the ability to infer the internal state of a system based on its externally

visible outputs, typically metrics, logs, and traces. Among these, distributed tracing offers the most detailed perspective, capturing fine-grained event sequences across asynchronous and parallel service boundaries. Several studies cite observability as a foundational capability for diagnosing and optimizing complex systems [2–7]. Observability is typically enabled through instrumentation, inserting tracepoints at key execution sites to collect and propagate trace data.

However, full-scale tracing comes at a cost. Capturing timestamps, managing context propagation, and storing large volumes of trace data introduce significant overhead. Even low-overhead instrumentation tools like LTTng or eBPF may impact performance in high-throughput production environments. Moreover, storing and analyzing the trace data, particularly in large-scale deployments, becomes computationally and economically expensive. These challenges motivate the need for tracing strategies that are both adaptive and cost-aware.

Runtime Overhead: Instrumentation, especially in user-space or distributed environments, incurs runtime costs. Capturing trace metadata, managing buffers, and propagating context information can increase latency and CPU usage. In high-load systems, even minimal overhead per request becomes significant at scale.

Data Volume and Storage: Large-scale systems can generate millions of spans per second. Persisting this data across time requires considerable storage, and transferring or indexing such volume adds network and processing load.

Analysis Bottlenecks: The utility of tracing lies not just in data collection, but in post-hoc reconstruction of execution paths, detection of anomalies, and identification of root causes. These tasks are non-trivial at scale and often delayed due to analysis bottlenecks.

Cost-Aware Tracing Needs: These factors highlight the need for tracing frameworks that dynamically adjust their fidelity and scope based on runtime needs. Rather than tracing all requests uniformly, systems must prioritize trace collection in ways that balance cost, fidelity, and diagnostic power.

2.1.1 Distributed Tracing Challenges

In distributed systems, tracing presents additional complications beyond local instrumentation. Causal relationships must be reconstructed across network boundaries, service topologies change rapidly, and instrumentation coverage is rarely complete.

Causal Context Propagation: To reconstruct the full path of a request, trace context (e.g., trace ID, span ID, parent ID) must be propagated across RPC boundaries, threads, and asynchronous calls. This requires consistent instrumentation support across all system components, a challenge in polyglot environments or when third-party libraries are involved.

Partial Observability: Not all services or components in a system may be instrumented. Some may lack trace support due to technical, performance, or compliance constraints. As a result, collected traces may be incomplete or fragmented, which complicates root cause analysis.

High Cardinality and Dynamic Topologies: Modern systems frequently deploy hundreds or thousands of ephemeral service instances that scale in and out. This dynamic infrastructure complicates trace indexing, aggregation, and correlation, especially when attempting to associate traces with resource-level metrics.

Sampling Trade-offs: To reduce overhead, head-based sampling methods (e.g., [8, 9]) randomly select requests for tracing, while tail-based sampling (e.g., [10,11]) selects based on observed latency or errors. Head-based methods may miss critical edge cases, and tail-based methods often require full trace capture prior to sampling, partly negating their overhead advantages.

Cross-Service Root Cause Localization: Even with complete traces, isolating the service or component responsible for a performance problem remains challenging. Latency or faults may propagate across services, and performance anomalies may arise due to complex interdependencies. Trace data provides necessary but insufficient context, models, heuristics, or augmented diagnostics are often required to support fault localization.

Together, these challenges underscore the need for adaptive tracing frameworks that are sensitive to runtime context, informed by diagnostic goals, and efficient enough to operate in production. The remainder of this chapter surveys the foundational techniques that underpin such frameworks, including execution tracing tools, instrumentation strategies, and trace sampling methods.

2.2 Tracing Techniques

Tracing is a powerful approach for understanding the internal behavior of distributed systems. It involves recording execution data such as function calls, system calls, scheduling events,

and inter-process communication. This information enables developers and operators to gain visibility into runtime behavior, identify bottlenecks, localize performance regressions, and verify performance-related assumptions.

To reduce the overhead and maximize the effectiveness of tracing, various techniques have been developed, including: (1) how execution events are captured, (2) how much of the data is collected (sampling), and (3) how instrumentation is configured (static, dynamic, or adaptive). In this section, we discuss key techniques for tracing modern distributed systems.

2.2.1 Execution Tracing: Tools and Spaces (User vs Kernel)

Execution tracing is a foundational technique for collecting runtime data. Unlike logging, which captures high-level user-defined messages, and profiling, which focuses on statistical summaries, tracing captures fine-grained, low-level event data, often with timestamps and context. This makes it particularly suitable for understanding performance behavior in high-throughput systems [12].

Tracers work by generating events at defined instrumentation points in the system. These events can be generated in either the user-space or kernel-space. Gebai and Dagenais [13] provides an extensive survey of commonly used tracing tools. In this thesis, we use the LTTng tracer, an open-source and low-overhead tracing suite developed at Polytechnique Montréal. LTTng supports both kernel-space and user-space tracing and is well-suited for large-scale systems with high data throughput [12].

User-space tracing: In user-space tracing, tracepoints must be inserted into the application code [13, 14]. This can be done statically, by modifying the source code prior to compilation, or dynamically, by injecting tracepoints at runtime [13, 15–17]. Static instrumentation is easier to manage but requires source code access and recompilation [14, 18]. LTTng-UST provides libraries and agents for tracing applications written in C, C++, Java, and Python [12, 14].

Kernel-space tracing: Kernel-space tracing allows the monitoring of operating system events such as thread context switches, scheduler activity, file I/O, system call latency, and memory management [13, 19, 20]. Unlike user-space tracing, it does not require any modification to application code [13, 18]. Kernel-space events are critical for analyzing behaviors like blocking, preemption, and contention, especially in scenarios where performance anomalies originate outside of user logic [20–22].

In this work, we primarily utilize kernel-space tracing due to its generality, low configuration

effort, and ability to expose system-level bottlenecks. Nevertheless, we remain open to incorporating user-space tracepoints when application-specific metrics are required for detailed diagnosis.

2.2.2 Sampling Methods (Head, Tail, Sifter, etc.)

Given the potentially massive volume of trace data in modern systems, sampling is often employed to reduce overhead and storage requirements. There are two dominant strategies: head-based and tail-based sampling.

Head-based sampling: This approach makes a sampling decision at the beginning of a trace. For example, Dapper [8] and Facebook Canopy [9] use random sampling to decide whether to capture an incoming request trace. These techniques are efficient and impose a minimal runtime cost, but risk missing rare or anomalous behaviors.

Tail-based sampling: Tail-based methods defer the sampling decision until after execution, allowing a selection based on observed metrics such as latency, page faults, or response size. Examples include latency-based sampling in Desfossez et al. [10], memory threshold-based sampling in Daoud et al. [23], and performance anomaly detection in Las-Casas et al. [11]. These methods capture high-value traces, but often incur more overhead due to the need to instrument all requests initially.

Representative sampling: [24] proposed a weighted sampling technique using hierarchical clustering (PERCH) to balance representativeness and diversity. Their method forms a tree of trace clusters and selects samples across different branches to avoid over-representation of common behaviors. However, its offline nature limits applicability in live production systems.

To overcome this, the same authors later proposed Sifter [11], a scalable online sampling method. Sifter uses prediction error from dimensionality-reduced trace data as a proxy for anomalous behavior. It maintains a fixed memory budget and guarantees $O(1)$ computational cost per trace. This makes it suitable for live systems with dynamic workloads. Sifter biases sampling toward rare or anomalous traces and demonstrates strong effectiveness on datasets like HDFS, DeathStarBench, and production microservices.

Partial sampling: [25] propose Partial Sampling, an adaptive technique that assigns independent sampling rates to individual spans based on attributes and resource constraints. This fine-grained approach minimizes trace volume but can result in fragmented traces. To

compensate, they introduce an estimation algorithm to reconstruct trace-level statistics from partial samples.

Overall, while head-based methods are efficient, they may omit critical edge cases. Tail-based and adaptive methods offer greater diagnostic value, but require careful tuning to control overhead. Balancing trace coverage, diversity, and cost remains an open challenge.

2.2.3 Instrumentation Strategies (Static, Dynamic, Adaptive)

Instrumentation is the process of inserting probes (i.e., tracepoints) into software code or binaries. These probes generate events when executed. Instrumentation can be categorized as static, dynamic, or adaptive, depending on how and when the tracepoints are inserted and adjusted.

Static Instrumentation: Static instrumentation involves inserting tracepoints into the source code before compilation. This is common in performance analysis tools like LTTng and is typically done by adding macro calls or annotations in code. Once compiled, the binary includes these probes permanently. Pythia [26] uses static instrumentation for tracepoint placement, based on prior analysis of problem areas.

Static instrumentation offers minimal runtime overhead and deterministic probe placement, but it lacks flexibility. Changes require recompilation, which is not practical in production systems or third-party components. Also, maintaining an optimal set of tracepoints, for all possible system states, is infeasible in large-scale dynamic environments.

Dynamic Instrumentation: Dynamic instrumentation injects tracepoints into the binary of a running application without modifying the source code. This allows tracing of arbitrary applications and supports runtime adjustments. Common tools for dynamic instrumentation include SystemTap, DTrace, and the dynamic mode of LTTng [27].

Dynamic techniques are especially useful in production, where source access is unavailable or downtime is unacceptable. However, they typically incur higher runtime overhead than static approaches and require careful configuration to avoid performance regressions.

Adaptive Instrumentation: Adaptive instrumentation techniques dynamically adjust the level or location of instrumentation based on runtime feedback. Zhang et al. [28] introduce an adaptive mechanism with three levels of tracing granularity: trace-level, server-level, and method-level. Their approach uses linear regression and density-based K-means to model response times and adjust instrumentation accordingly.

PIRA [29] and its enhancement [30] automate iterative refinement of compiler instrumentation using heuristics and runtime models. These tools leverage performance measurements to prioritize hot functions and reduce overhead. Sun et al. [31] propose a syntax-tree-based method to automatically insert instrumentation for MPI programs, later refining probe selection through random forest regression modeling.

Another notable approach is Castanheira et al.'s query-based instrumentation activation method [32]. Instead of inserting hard-coded tracepoints, they annotate code locations with conditional instrumentation markers. These can be toggled on or off at runtime via declarative queries, allowing DevOps teams to adjust instrumentation in production without code changes.

Static instrumentation is efficient but inflexible. Dynamic instrumentation offers flexibility but at a higher cost. Adaptive instrumentation seeks the best of both worlds by minimizing overhead and maximizing insight. However, adaptive strategies require performance modeling and control loops, which introduce complexity. Trade-offs between trace fidelity, system impact, and configuration effort remain a central concern in large-scale tracing.

In this work, we aim to integrate adaptive instrumentation and sampling with performance models to support low-cost, high-fidelity tracing aligned with current diagnostic goals.

2.3 Performance Understanding Approaches

In large-scale systems, tracing alone may not be sufficient to interpret the system's runtime behavior or adapt instrumentation effectively. Due to the large volume and complexity of the data collected, performance modeling, prediction, and measurement techniques are essential. These methods enable a more structured understanding of the system's dynamics and support runtime decisions regarding tracing and optimization.

Performance understanding methods fall into three categories: (1) modeling approaches that abstract the system's behavior using formal models or learning techniques, (2) prediction methods that estimate future behavior based on current observations, and (3) direct performance measurements using low-overhead instrumentation. Together, these approaches enable goal-driven tracing, where the tracing strategy adapts based on real-time feedback and analytical insight.

2.3.1 Performance Modeling (ML/DL, Queuing Models)

Performance modeling transforms raw execution traces into interpretable abstractions of system behavior. These models are especially useful in distributed and parallel systems, where performance is affected by interactions between multiple components, shared resources, and asynchronous events.

Tracing is one of the traditional approaches for collecting performance data. However, due to high data volume and storage limitations, full-scale tracing is impractical in live systems. Online modeling techniques aim to complement tracing by producing models that reflect system performance while minimizing tracing overhead.

Performance models serve various purposes such as anomaly detection, fault localization, bottleneck identification, and workload characterization. Common types of models include queuing networks, execution flow models, and component interaction graphs [33–36].

Models can be built from kernel-space traces, user-space traces, or a combination of both. For instance, Woodside et al. [37] discusses limitations and considerations when building kernel-level models. User-space trace–based modeling is also addressed in works like [28]. These models may differ based on the level of abstraction and targeted use cases.

In the context of distributed systems, several tasks have been addressed using modeling techniques:

- **Anomaly detection:** identifying execution patterns that deviate from normal behavior [36, 38, 39].
- **Root cause analysis:** pinpointing which part of the system is responsible for observed performance problems.
- **Performance variation detection:** identifying abnormal behavior across repeated requests [40, 41].
- **Workload characterization:** clustering and profiling workloads to understand usage patterns [42].

Fournier et al. [39] propose a method for detecting anomalies in web request traces by analyzing user-space and kernel-space trace features. The method extracts system call sequences, state transitions, and execution durations from the critical path of each request, then clusters them using DBSCAN and TF-IDF weighting. The result is a set of behavior-based outlier groups that offer insight into root causes.

Kohyarnejadfard et al. [43] propose vector-based modeling, where each feature represents the duration spent in a particular system state. This approach captures latency-sensitive deviations even when the control flow remains constant.

Zhang et al. [28] propose an adaptive tracing mechanism that uses regression-based modeling of performance metrics like response time and data size. They apply K-Means clustering to identify outliers and adjust instrumentation levels accordingly.

Nemati et al. [42] present a host-level workload characterization method using kernel-level traces and K-Means clustering in two stages (coarse and fine-grained) to classify virtual machine workloads.

In addition to traditional modeling, deep learning techniques are gaining attention. Dymshits et al. [44] use LSTM networks to model sequences of system call count vectors for process monitoring and anomaly detection. The model learns normal behavior and flags deviations due to faults or attacks.

Cortellessa and Traini [45] propose a genetic search–based method that clusters traces based on RPC timing patterns to detect latency degradations.

Bhattacharyya and Hoefler [46] automatically builds runtime models using compiler-inserted instrumentation and LASSO regression. This approach aims to reduce both tracing overhead and model size by selecting only essential metrics.

2.3.2 Performance Prediction (Analytic/Non-analytic)

While performance models provide a retrospective view, prediction methods offer foresight. They allow us to forecast system behavior and adjust tracing policies in advance.

Performance prediction methods are classified along multiple axes:

- **Application-dependent vs. application-independent:** whether the model is specific to a particular application or generalized across domains.
- **Platform-dependent vs. platform-independent:** whether the model is specific to a given hardware/software setup [47].
- **Analytic vs. non-analytic vs. characterization-based:** based on whether the prediction is derived from mathematical models, machine learning, or empirical profiling.

Analytic Methods

Analytic prediction methods use mathematical formulations to estimate performance. These include:

- **Statistical confidence intervals:** Doray and Dagenais [48] model performance across application versions to predict regression risks.
- **Linear regression:** Widely used in modeling throughput, latency, and resource consumption.

Analytic methods are easy to interpret but may struggle to model complex interactions in dynamic systems.

Non-Analytical Methods

Non-analytical methods rely on supervised and unsupervised learning techniques. Examples include:

- **Support Vector Machines (SVM):** Used in [43] to predict system call durations and classify anomalous traces.
- **Neural Networks (ANNs and LSTM):** Applied for nonlinear regression, sequence modeling, and latent anomaly detection [44].
- **Classification and tree-based models:** Techniques like k-NN, CART, Random Forest, and MARS have been employed in [28, 49].
- **Instance-based (lazy) learning:** These techniques store trace data and retrieve similar historical instances for prediction, useful when system behavior is irregular.

Characterization-Based Methods

Characterization methods construct profiles based on observed execution behavior. Categories include:

- **Application execution:** Methods that build performance models directly from runtime data.
- **Offline/online profiling:** Tools like Extra-P [50] and PIRA [29] use program characteristics to estimate future performance.

- **Source code analysis:** Static analysis of control flow, function size, or instruction count to infer performance properties.

Characterization is useful when modeling or predicting performance from high-level structure or code complexity. In this work, we primarily focus on execution-based profiling, as it aligns with tracing-based observation.

An example of a trace-based predictor is MEPFL [51], a learning model trained on fault-injected system logs. It uses features such as execution time, number of microservices invoked, and service interaction complexity to predict latent faults. MEPFL is application-independent and relies on classification models to predict trace failures based on historical trace structures.

2.3.3 Performance Measurement Techniques

In addition to modeling and prediction, direct measurement of key metrics provides practical insight with low overhead. These techniques allow runtime instrumentation to be guided by real-time thresholds and performance counters.

Some approaches measure system behavior directly:

- **QoS monitoring:** QDIME [52] dynamically adjusts instrumentation based on user-defined QoS thresholds.
- **Statistical measurements:** Lehr et al. [30] use Extra-P [50] to fit polynomial models to trace measurements.
- **Static analysis heuristics:** PIRA [29] selects functions for tracing based on function size or call count.
- **Dynamic analysis:** Code coverage and call frequency are used to adjust tracepoint priority.

[53] distinguish between internal and external anomalies using statistical outlier detection, then localize faults to system calls. esfossez et al. [10] selectively trace high-latency requests by discarding those below a predefined threshold.

Sambasivan et al. [40] compare performance across application versions using a median-based confidence interval method to detect regressions. This approach identifies meaningful shifts in execution metrics while ignoring noise.

These measurement-based methods are lightweight and well-suited for production systems where full-scale tracing is not feasible. They also provide input for adaptive instrumentation and selective sampling.

2.4 Performance Diagnosis and Localization

Tracing and modeling methods enable the detection of performance anomalies, but diagnosing their root cause remains a critical challenge. Performance diagnosis aims to identify which component, service, or condition is responsible for observed degradation. In large-scale systems, performance anomalies are often emergent behaviors caused by subtle interactions between distributed services.

Recent research has focused on three complementary directions for performance diagnosis: (1) workflow-centric tracing for capturing distributed request execution paths, (2) fault localization and visualization to support human analysts, and (3) knowledge-based and experimental techniques for identifying root causes.

2.4.1 Workflow-Centric Tracing (Dapper, Jaeger)

Workflow-centric tracing techniques capture the end-to-end execution path of a request across microservices. These techniques record causally related events, typically using spans and traces that represent service calls and execution contexts, respectively.

Pioneering work in this space includes Google's Dapper [8], which introduced trace-based diagnostics into production systems. Dapper records per-request metadata (e.g., RPC timing, service names, thread IDs) and uses Gantt charts for visualizing end-to-end latencies. Similar tools like Zipkin[1] and Jaeger[2] extend this model and are widely adopted in open-source environments.

Facebook's Canopy [9] builds on this by offering streaming trace processing and user-defined metric extraction. It supports multi-dimensional performance views and dynamic filtering over billions of traces. Pivot Tracing [54] adds runtime instrumentation control, enabling developers to define queries that cross service and machine boundaries.

These tools are crucial for identifying distributed bottlenecks, latency spikes, and unexpected dependencies. However, their visualizations, such as Gantt charts, are only practical when analyzing a small number of requests. In high-volume settings, analysts require summarization, anomaly highlighting, and automated diagnostics.

[1] `https://zipkin.io/`
[2] `https://www.jaegertracing.io/`

2.4.2 Fault Localization and Visualization

Once anomalies are detected, the next step is to localize their cause. Several studies support this process through visualization, machine learning, or statistical analysis of trace metrics.

RR Sambasivan et al. [55] evaluate three visualization approaches to present root cause analysis results. They compare raw traces, summary graphs, and ranked explanations to determine which best supports diagnosis. Their work highlights the importance of interface design in aiding analysts.

Malik et al. [56] propose constructing performance signatures from load test counters and using them to identify subsystems responsible for SLA violations. In [57], four ML-based approaches are evaluated to detect test-to-test deviations. These methods reduce the number of counters analysts need to examine, improving efficiency.

Han et al. [58] focuses on callstack analysis by mining execution traces to isolate the most relevant paths. It reduces the volume of callstack data presented to developers and aids in localizing faults with minimal manual effort.

Dean et al. [53] use statistical techniques to distinguish between internal and external anomalies and highlight the most impacted system calls. Their method helps identify root causes by computing the deviation of call timing distributions.

2.4.3 Knowledge-Based and Experimental Approaches

Beyond visualization and statistical summaries, knowledge-based models and controlled experiments can significantly improve fault localization. These approaches integrate system semantics, architectural knowledge, and performance taxonomies to guide diagnosis and tracing.

Systematic experimentation, as described in [59], involves fault injection or controlled workload scenarios to isolate specific root causes. For example, Surya and Rajam [60] use a second-order Markov model to detect contention and attribute it to specific resources.

Wert et al. [59] propose a performance problem taxonomy that connects symptoms (e.g., latency violations) to root causes (e.g., serialization, contention). This hierarchy enables targeted instrumentation and debugging by narrowing the search space. Similarly, Sánchez et al. [61] compile a catalog of real-world performance bugs, organized by cause, effect, and context. These structured knowledge bases help developers recognize recurring SPAs and trace them back to their architectural origin.

Other domain-specific taxonomies exist. For example, Smith [62] explores SPAs in cyber-

physical systems, which exhibit different failure modes due to timing and hardware-specific interactions.

In addition to these static models, several works explore adaptive tracing. Khan and Ezzati-Jivan [63] propose a runtime adaptive tracing technique that adjusts instrumentation based on execution trends. Their method enables or disables tracepoints using real-time anomaly scores. Likewise, Wert et al. [64] detects ramp and traffic jam patterns by analyzing metric trends, correlating queue buildup with resource contention. Nlane [65] targets variations of the *one-lane-bridge* problem by monitoring fine-grained metrics and synchronization bottlenecks.

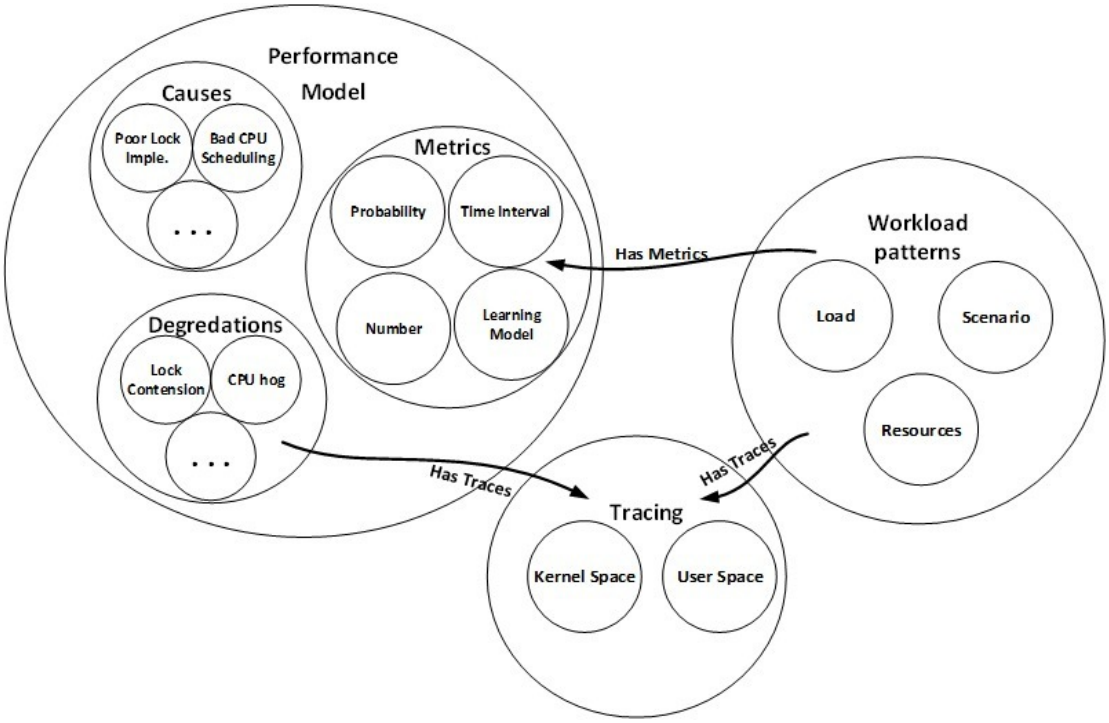

Figure 2.2 A knowledge model supporting adaptive tracing. Metric types, workload characterizations, and tracer configuration properties are formalized as knowledge elements to guide trace control decisions.

Figure 2.2 presents the knowledge model that we proposed for adaptive tracing in our research proposal. Inspired by prior taxonomies, this model formalizes the relationships between system behavior (e.g., metrics, workloads), trace configuration (e.g., sampling level, instrumentation granularity), and diagnostic goals (e.g., detection of specific SPAs). By capturing this domain knowledge explicitly, the model enables automated or semi-automated decision-making for tracing adaptation.

These approaches, especially when combined, enable more effective diagnosis in dynamic systems. While visualization supports analysts, knowledge-based and model-driven techniques facilitate automation, reuse, and generalization.

Nevertheless, many of these methods still face key limitations. Static tracing strategies often fail to adapt to changing runtime contexts. Instrumentation may be coarse-grained or misaligned with root-cause signals. Adaptive approaches, while promising, still suffer from tuning complexity and scalability challenges. Addressing these limitations through structured knowledge and goal-based control is a central aim of the framework proposed in this thesis.

2.5 Software Performance Anti-Patterns

SPAs are recurring design or implementation flaws that degrade system performance. Unlike transient anomalies caused by temporary resource contention or load spikes, SPAs are structural and persistent, stemming from architectural or logical inefficiencies. They may not violate correctness but often result in poor scalability, resource waste, or unresponsiveness.

SPAs typically emerge during software evolution as systems grow in complexity. Their detection and resolution are essential for maintaining performance, especially in distributed systems with numerous interacting components. This section introduces the taxonomy of SPAs, provides detailed examples relevant to this thesis, and reviews existing detection techniques across multiple paradigms.

2.5.1 Taxonomy

SPAs are recurring structural or behavioral flaws in software systems that degrade performance. These patterns have been categorized in various taxonomies, often based on their scope (e.g., internal vs. inter-service), resource usage (e.g., messaging overhead, payload size), or system-level manifestations (e.g., blocking, saturation, contention).

The concept of SPAs was first introduced by [66], who drew analogies to classical design anti-patterns. Subsequent work by [67, 68] cataloged patterns such as *God Object* and *Excessive Serialization* that negatively affect scalability and responsiveness.

A more structured SPA taxonomy was proposed by [59], organizing performance problems into a hierarchical model. In this model, high-level nodes represent observable symptoms (e.g., latency spikes or high variability), while deeper nodes represent execution-level or design-level causes (e.g., resource contention, redundant communication, or inefficient serialization). The taxonomy spans multiple abstraction layers:

- **User-level symptoms:** increased response time, high variability.
- **Execution-level issues:** blocking, queuing, garbage collection.
- **Design-level flaws:** excessive coupling, centralized coordination, lack of caching.

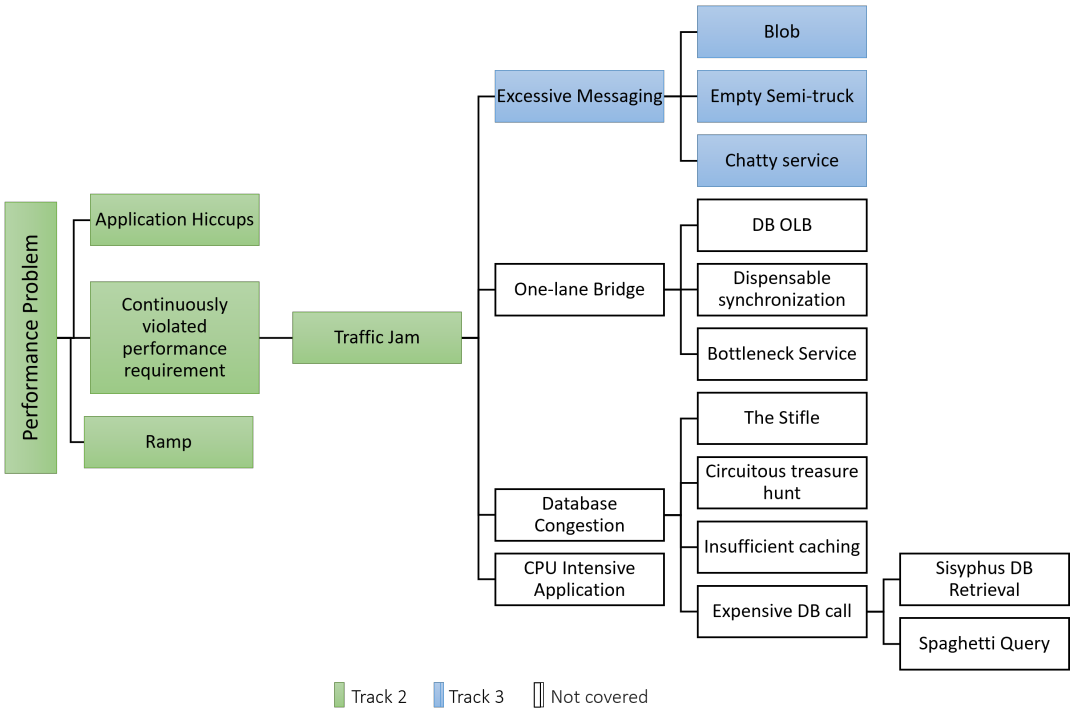

Figure 2.3 SPA taxonomy and hierarchy. Boxes marked in green indicate the SPAs addressed in this thesis; gray boxes denote patterns not covered.

Figure 2.3 shows the SPA hierarchy used in this which is based on [59]. We adopt this taxonomy as a diagnostic guide and restrict our focus to SPAs that can be detected via user-space instrumentation and inferred through trace data. Other SPAs, while relevant, fall outside the scope of this thesis due to limitations in trace granularity, system observability, or diagnostic priorities.

2.5.2 Detailed SPA Examples

Here we describe several SPAs that are commonly encountered in distributed systems and are central to our study.

Continuously Violated Performance Requirements. This SPA refers to sustained failure in meeting defined service-level objectives (SLOs), such as maximum response time or throughput. Causes include inefficient algorithms, under-provisioned resources, or sub-optimal architectural decisions. Detection involves comparing current behavior to baselines derived from historical traces [59].

Single User Problem. As defined by [69], this SPA describes poor performance under minimal load. It typically results from algorithmic inefficiencies, large initialization costs, or excessive queries. For instance, a single user triggering multiple synchronous database queries due to ORM misconfiguration leads to latency issues, regardless of system concurrency.

Hiccup. Hiccups manifest as brief, periodic spikes in latency caused by background tasks such as garbage collection, logging, or thread pool resizing [70]. They often require fine-grained trace resolution to detect, and may not be visible in coarse summaries.

Traffic Jam. This anti-pattern occurs when concurrent requests overwhelm a service, causing a backlog. Requests queue up, propagating delays throughout the system. Common in synchronous RPC chains, it amplifies delays downstream and creates performance cascades. Queueing models help simulate and detect this condition [71].

Ramp. The Ramp pattern describes gradual performance degradation over time, under constant load. Causes include memory leaks, unreleased resources, or inefficient caching [59]. Tracing trends in memory usage and response time can reveal this anomaly.

Blob. Described in [70], the Blob anti-pattern refers to centralized services that handle a disproportionate amount of logic or data, becoming communication bottlenecks. These nodes have high in-degree in the service graph and affect system scalability.

Empty Semi-Trucks. This SPA involves excessive fine-grained message passing between services. Instead of aggregating payloads, the system sends numerous small messages, increasing serialization overhead and I/O cost. [72] analyze span message sizes and frequencies to detect this condition.

Circuitous Treasure Hunt and Stifle. These patterns refer to inefficient access paths or excessive data queries, often manifested in long chains of dependent service calls [59]. Trace inspection of depth and call count is necessary for detection.

2.5.3 Detection Techniques (Statistical, Model-based, Machine Learning)

Numerous approaches have been proposed to detect SPAs, ranging from heuristic-based thresholds to advanced machine learning models. We categorize them into three main families: statistical methods, model-driven methods, and learning-based methods.

Statistical Detection Methods

Early work by [66] advocated statistical profiling to identify recurring indicators of SPAs. Metrics such as function execution time, memory usage, and message size are monitored, and thresholds or deviations are used to flag potential problems.

Stadelmaier [73] proposed clustering of function execution profiles to identify anomalies. Although effective for CPU-bound applications, this method may miss communication-based SPAs.

Wert et al. [70] analyzed span-level metrics such as hop count, latency per hop, and payload size. Avritzer et al. [71] combined statistical thresholds with queueing theory to identify performance bottlenecks such as Traffic Jam or Blob.

Graph-based methods, like [74], represent microservice interactions as graphs and use node-level metrics (e.g., degree centrality, betweenness) to detect architectural flaws such as Blob and Empty Semi-Trucks.

Model-Based Approaches

Model-driven SPA detection builds formal or empirical models of system behavior, then identifies deviations. Architecture Description Languages (ADLs) [75] annotate system components with performance metadata and compare observed behavior against these models.

Avritzer et al. [76] map traces to queuing models, measuring metrics such as service time variance to detect anomalies. Discrepancies between expected and actual metrics indicate potential anti-patterns. Model-based techniques offer explainability but depend on accurate annotations and system models.

Machine Learning Approaches

ML-based methods are data-driven and excel at uncovering complex or non-obvious patterns.

Lui et al. [77] used an SVM classifier to detect the "More is Less" anti-pattern in MySQL, where additional resources paradoxically reduce performance. The model used performance

counters such as disk I/O and memory usage.

Fournier et al. [39] and Kohyarnejadfard et al. [43] apply unsupervised learning to detect outliers in trace features. Techniques include DBSCAN clustering, TF-IDF scoring, and vector distance analysis. These methods detect anomalous executions without labeled training data. Despite their power, ML-based approaches face limitations:

- **Instrumentation Overhead:** Full-trace methods may not be feasible in production.
- **Lack of Adaptiveness:** Static models fail to capture runtime variation.
- **Sparse Labels:** Supervised learning requires labeled datasets, which are rarely available.

This thesis builds on these insights by proposing semi-supervised detection of SPAs using trace-driven features and adaptive instrumentation. The SPA taxonomy helps map observed anomalies to relevant traces, narrowing down the tracepoints that require closer examination. Our goal is to align SPA detection with runtime tracing, enabling real-time diagnosis with minimal overhead.

2.6 Summary and Gap Analysis

The preceding sections have surveyed the major approaches to tracing, performance modeling, anomaly diagnosis, and software performance anti-pattern (SPA) detection. Table 2.1 provides a high-level summary of the reviewed methods, categorized by strategy and evaluated by their primary focus and key trade-offs.

Despite significant progress in these areas, several important gaps remain in the current body of work:

- **Lack of runtime adaptiveness:** Most existing methods apply static tracing or sampling policies that do not respond to changing system behavior. This limits their effectiveness in dynamic environments.
- **Overhead–fidelity trade-off:** Techniques that reduce overhead often sacrifice trace fidelity, whereas those that preserve details (e.g., full tracing) introduce prohibitive costs. A better balance is required.
- **Sparse integration between SPA taxonomy and tracing:** While SPA taxonomies such as [59] structure performance problems well, few tracing strategies exploit this hierarchy to guide tracepoint selection or drill-down instrumentation.

Table 2.1 Summary of tracing and diagnosis techniques and their trade-offs

Technique	Focus	Strengths	Limitations
Static Instrumentation [26]	Predefined tracepoint placement	Low overhead, deterministic behavior	Inflexible, requires recompilation
Dynamic Instrumentation [27]	Binary-level runtime instrumentation	No source required, flexible	Higher runtime overhead
Adaptive Instrumentation [28, 29]	Runtime adjustment of trace granularity	Context-aware, reduces overhead	Requires runtime model, tuning
Head-based Sampling [8, 9]	Probabilistic request sampling	Lightweight, scalable	Random sampling may miss edge cases
Tail-based Sampling [10, 11]	Metric-based trace filtering	Captures anomalies, more informative	Costly, requires full instrumentation first
Performance Modeling [33, 36]	System abstraction using trace data	Enables diagnosis and prediction	Requires sufficient and relevant data
Performance Prediction [28, 43]	Forecasting future behavior	Supports proactive adaptation	Depends on trace quality, context-sensitive
Workflow-centric Diagnosis [8, 9]	End-to-end trace correlation	Service interaction visibility	Scalability and visualization limitations
SPA Detection (Statistical) [59, 70]	Metric-based outlier detection	Explainable, low overhead	Limited flexibility, symptom-oriented
SPA Detection (ML/Modeling) [39, 77]	Pattern discovery in trace data	Can detect complex anomalies	Needs training data or models, lacks interpretability

- **Limited use of semi-supervised learning:** Most ML-based detection methods rely on either fully supervised models (requiring labeled data) or unsupervised outlier detection. Few approaches leverage the power of weak supervision or semi-supervised learning to reduce labeling burden while improving accuracy.

- **Fragmentation across layers:** Many methods focus on either tracing, modeling, or diagnosis in isolation. Cross-layer feedback mechanisms that coordinate tracing granularity, sampling, and model-driven inference are rarely implemented in practice.

This research aims to address the above limitations by proposing a unified, adaptive tracing framework guided by performance goals and informed by a performance problem hierarchy. The key contributions of this approach are:

- Leveraging SPA taxonomy to prioritize and adapt tracepoint activation in real time.

- Combining performance measurements, prediction models, and runtime monitoring to balance tracing cost and detail.

- Applying semi-supervised learning to label SPA-related anomalies with minimal manual effort, using weak indicators extracted from traces.
- Integrating tracing control with trace analysis feedback, enabling context-aware adaptation of instrumentation and sampling policies.

By doing so, the framework supports low-cost, targeted trace collection aligned with system state and analysis goals, improving the timeliness and interpretability of performance diagnostics.

The following chapter provides an overview of the three research tracks presented in this thesis before delving into them in more detail in the subsequent three chapters.

CHAPTER 3 OVERVIEW

This dissertation brings together three research articles, each tackling a key challenge in performance diagnostics for modern distributed systems. These articles, presented in Chapters 4, 5, and 6, build progressively from detecting anomalies using execution traces to localizing performance issues and identifying SPAs in communication-heavy environments. The common thread across all three is the goal of making tracing smarter, lighter, and more focused on what actually matters.

The work starts with critical-path–based anomaly detection, moves toward goal-driven tracing for root cause analysis, and ends with selective tracing techniques to detect recurring inefficiencies in microservice architectures. Each article proposes concrete methods to reduce the volume and cost of trace data, while improving the clarity and usefulness of what is collected.

Chapter 4 presents the first article, titled *"TraceLens: Early Detection of Software Anomalies Using Critical Path Analysis"*, published in the *ICPE 2025 Conference*. This paper looks at how we can detect performance anomalies with less tracing overhead, by focusing only on the parts of the system that are likely to be performance bottlenecks. It introduces an adaptive tracing mechanism that uses critical path analysis to guide data collection, supported by LSTM models that learn patterns in system behavior. Instead of collecting everything, the system focuses on what is most relevant, resulting in fewer traces to analyze, but better insights. This method provides a solid base for the more targeted tracing techniques developed in the next stages of this research.

Chapter 5 presents the second article, *"Software Performance Problem Localization: A Goal-Based Adaptive Tracing Approach"*, currently under review at the journal of Concurrency and Computation: Practice and Experience. The core idea is that not all performance problems are the same, and tracing tools should adapt based on the specific issue being investigated. This article presents a flexible tracing framework that configures trace collection according to the analysis goal—such as identifying SPAs or pinpointing the source of latency. By leveraging user-level traces, this method can detect some of the most common SPAs. Because the work focuses on well-defined SPAs, statistical analysis of the collected data enables not only the detection of problems but also an understanding of their root causes—without being overwhelmed by excessive trace data.

Chapter 6 presents the third article, *"Efficient Detection of Communication-related SPAs*

in Microservices", submitted to the *Journal of Software: Evolution and Process*. This final piece looks at how SPAs like *Blob* and *EST*, which are common in microservice architectures, can be detected early and accurately, without full-scale tracing. It combines selective tracing (both in kernel and user space) with trace correlation across services. The method achieves high detection accuracy with very low overhead and is designed to plug directly into CI/CD pipelines. This means teams can catch communication inefficiencies during development or testing, long before they reach production. Another important focus of this work is to study the feasibility of using different types of machine learning techniques, namely supervised, unsupervised, and semi-supervised, to detect SPAs.

Across these three articles, the research presents a unified approach to smarter performance diagnostics: collect only what is necessary, analyze efficiently, and adapt tracing to suit the problem at hand. The result is a set of practical tools and techniques that help teams better understand and improve the performance of large, complex software systems, without paying the usual cost of full-scale tracing.

CHAPTER 4 ARTICLE 1 TRACELENS: EARLY DETECTION OF SOFTWARE ANOMALIES USING CRITICAL PATH ANALYSIS

Authors: Masoumeh Nourollahi, Amir Haghshenas, Michel Dagenais
Published in: Companion of the 16th ACM/SPEC International Conference on Performance Engineering. 2025-02-09.

abstract: Runtime smell detection in software systems, particularly through system call analysis, has garnered significant attention in recent years. Although various machine learning techniques have been employed to enhance detection accuracy and reduce false positives, limited focus has been given to their practical application in early real-time anomaly detection. To address this gap, we propose a deep learning-based approach, called TraceLens, designed for the early detection of performance-related issues in software systems. Unlike traditional methods that rely on system call data, our approach leverages critical path analysis, enabling more efficient and targeted anomaly detection. Experimental results demonstrate that this approach achieves detection performance comparable to methods that use system calls, while significantly improving data collection efficiency. In addition, the critical path dataset highlights software dependencies, both internal and external, providing deeper insight into the dynamic behavior of software systems.

Keywords: Early detection; Critical path analysis; Large scale tracing; Software performance; Anomaly detection

4.1 Introduction

The advancement of IoT, Edge, and Cloud systems has led to a rapid increase of software systems that operate continuously over extended periods. These systems exhibit dynamic behavior patterns that evolve over time. Execution tracing is a widely adopted technique to monitor the runtime behavior of such applications and detect potential performance issues [3, 78]. However, consistently understanding and reacting to the time-varying behavior of software systems remains a significant challenge [10, 26, 54, 79].

Early detection of software execution anomalies using trace data is critical for maintaining system reliability, preventing failures, and minimizing performance degradation. However, existing detection methods face several challenges. system call based approaches, while widely used, generate large volumes of trace data, making real-time analysis computationally expensive and impractical for large-scale systems [39, 80]. Additionally, these methods often do

not capture execution dependencies, making it difficult to correlate anomalies with their root causes. Many performance issues arise not from isolated system events but from complex interactions between software components, such as synchronization delays, cascading failures, or external resource bottlenecks. Addressing these challenges requires a more efficient and structured approach to anomaly detection that reduces data overhead while preserving key execution insights.

To address these limitations, we propose TraceLens, a deep learning-based early detection approach that analyzes critical path data extracted from execution graphs. Unlike system call based methods, TraceLens focuses on the most influential execution sequences, enabling highly accurate anomaly detection while significantly reducing the amount of data required. By capturing key execution dependencies, TraceLens provides a structured and scalable solution for real-time anomaly detection, allowing software engineers to identify and address potential failures before they impact system performance.

Our research investigates whether critical path analysis can achieve detection accuracy comparable to system call based methods while requiring significantly fewer data points. We also explore the application of deep learning models in detecting anomalies without relying on excessive trace collection. Through extensive experiments, we demonstrate that TraceLens can pinpoint execution anomalies with high accuracy and minimal overhead, making it a practical and scalable solution for real-time software monitoring in complex systems.

The lightweight nature of our early detection approach makes it well suited for adaptive tracing. By detecting anomalies in real time with minimal overhead, the system can dynamically adjust tracepoints, enabling or disabling specific data collection regions as needed. This adaptability allows future trace collection to focus on critical areas of execution, improving efficiency and enhancing root cause analysis without overwhelming the system with unnecessary trace data.

The remainder of this paper is organized as follows: Section 2 reviews related work, categorizes existing approaches and highlights their strengths and limitations. Section 3 introduces our proposed approach, beginning with a concise definition of critical paths and execution graphs. Section 4 evaluates the proposed method through experimental analysis and provides the results. Section 5 includes a discussion of our method and findings, and finally, Section 6 concludes the paper and outlines potential directions for future research.

4.2 Related work

Detecting performance anomalies in complex, continuously running systems is challenging, especially when tracing must dynamically adapt to behavioral changes. Traditional methods relied on static thresholds, such as Desfossez et al. [10], who introduced a kernel-level latency detection method using predefined thresholds. While effective for offline analysis, this approach lacked adaptability for real-time systems.

To overcome static limitations, machine learning-based anomaly detection methods have gained traction. Zhang et al. [80] used system call traces for runtime security anomaly detection, employing n-grams and TF-IDF representations to optimize classification. Similarly, Dymshits et al. [44] leveraged LSTM-based deep learning, analyzing system call count vectors to detect behavioral changes. While these methods improved detection accuracy, they generated large trace sizes, making them inefficient for real-time deployment.

Other approaches focused on outlier detection and clustering. Fournier et al. [39] used system call traces to cluster web request anomalies, identifying slow execution patterns. Kohyarnejadfard et al. [43] employed multi-class SVM to detect anomalies based on system call frequencies and execution times. However, these methods required comprehensive system call monitoring, leading to significant overhead.

A more recent alternative, proposed by Janecek et al. [81], introduces a critical path-based anomaly detection framework. Their method extracts execution critical paths and applies multiple sequence alignment to identify anomalies, reducing the amount of trace data needed. While effective in pinpointing performance issues, their approach is post-mortem (offline) rather than real-time, limiting its use in adaptive tracing scenarios.

Among these studies, only Zhang et al. work [80] incorporated dynamic adaptation, selecting classifiers based on runtime behavior. To further improve adaptability, we propose starting with minimal tracing [82] and dynamically adjusting tracing levels based on system behavior. Our approach significantly reduces trace overhead, enhances scalability, and makes real-time anomaly detection feasible for large-scale systems.

4.3 Early detection

In this paper, we introduce an early detection method for identifying performance issues in software systems. Our approach has two key objectives: (a) enabling real-time detection of performance anomalies and (b) identifying long-term issues like memory leaks. To apply our real-time detection, we used the publicly available data set published by Noferesti et al. [83]

to train our model and capturing normal and abnormal behavior patterns. Designed for practical deployment, our approach is efficient and lightweight, minimizing system overhead while ensuring usability at runtime.

This section begins with an overview of the execution graph and critical path analysis, highlighting their importance with an example. We then detail the early detection process, including preprocessing steps and analytical techniques used for anomaly detection.

4.3.1 Critical path dataset description

For our methodology, we used execution graphs and their critical paths to reveal dependencies within a software system. We present an example of critical path analysis, to better understand their role in detecting performance issues.

Giraldeau [84] presents an example of critical path tracing in the Advanced Packaging Tool (APT), a frontend for *dpkg*. The execution of *dpkg* involves multiple resource-intensive operations, including network activity, disk I/O, and CPU load, making the extraction of APT's critical path a non-trivial task. Figure 4.1, adapted from [84], illustrates the critical flow view of APT traces. The execution lasts approximately seven seconds, with 37% of the time spent in blocked states. Notably, label (3) in Figure 4.1 highlights a dense cluster of arrows corresponding to manual page installations, where 543 *mandb* processes are created. Further analysis of this section revealed an issue related to the internal behavior of *libpipeline*. This example demonstrates how critical path analysis provides deep insights into the execution phases of complex software like APT, helping to uncover performance bottlenecks and inefficiencies.
This motivates our choice of critical path-based approach over system call based approaches. Our methodology extracts two datasets derived from critical path execution traces: the first feature dataset captures the count vectors of states visited along the critical path, while the second dataset records the duration vectors of these states. These datasets provide valuable insights into the dynamic behavior and internal/external dependencies of a software system. They enable the detection of anomalies, such as latency issues, and facilitate root cause analysis by tracing performance bottlenecks.

4.3.2 Model training

As shown in Figure 4.2, our early detection method follows an iterative cycle. In the data collection phase, tracing agents deployed alongside the application capture lightweight event traces to extract the execution graph and critical path, which highlight waiting-for dependen-

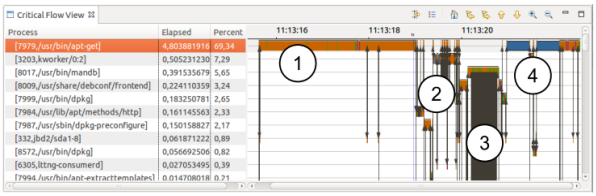

Figure 4.1 Overview of APT critical path.

cies to identify potential issues. Unlike traditional system call based methods, our approach analyzes a significantly smaller subset of trace data, reducing overhead. To extract critical path state sequences, we aggregated trace data within defined time windows.

For evaluation, we used the public dataset from Noferesti et al. [83], which contains 24,263,691 kernel-level events and system calls collected from an ELK stack running on Ubuntu 22.04.2 LTS. The dataset captures CPU, I/O, network, and memory noise under light-load and heavy-load conditions, providing a realistic testing environment for software performance engineering.

Since effective anomaly detection requires substantial data, we applied data augmentation, increasing the dataset sixfold within the first 1,200 seconds of execution. This segment primarily represents normal system behavior with minimal noise, allowing our model to better learn execution patterns and improve anomaly detection accuracy.

In the data preprocessing phase, we counted state occurrences and measured time spent in each state for every sequence, generating two feature vectors: state count and state duration. We then concatenated these vectors to create a unified dataset, which was split into training (90%) and testing (10%) sets for evaluation, which can be seen in detail in Table 4.1 Since real-world data lacks predefined scales, we normalized all values between 0 and 1 to ensure consistency during training.

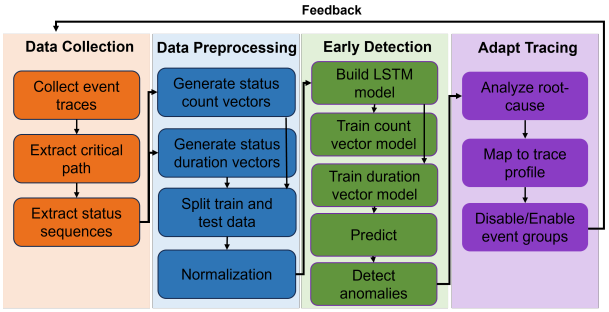

Figure 4.2 Early detection and adaptive tracing operational process.

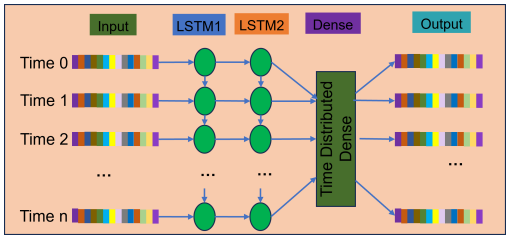

Figure 4.3 Time series LSTM model architecture.

Table 4.1 Dataset specifications for training and testing sets.

DataSet	***number of records***	***vector size***	***time window size***
Train	4,847	28-ary	100 seconds
Test	1,212	28-ary	100 seconds

To extract the feature vectors, we used EASE scripting within Trace Compass [85], an open-source trace analysis tool, converting traces into a Python-compatible format. For each time interval, we analyzed events, counted state occurrences, and measured time spent. The states included UNKNOWN, SYSTEMCALL, USERMODE, BLOCKEDIO, BLOCKED-CPU, BLOCKEDWAITKERNEL, BLOCKEDWAITPROCESS, BLOCKEDFORFUTEX, BLOCKED-CPUFUTEX, and TRASH.

For early anomaly detection, we implemented a Long Short-Term Memory (LSTM) model [38] designed to process sequential execution data. As shown in Figure 4.3, we adopted a many-to-many architecture, where a fixed-size window of test data vectors was fed into the model at each time interval. The network consisted of two hidden layers (16 and 4 units), followed by a time-distributed dense layer. We applied ten-fold validation to evaluate performance, reporting average accuracy, precision, and recall. After training, the model could be reused for continuous anomaly detection, with periodic retraining to adapt to evolving system behavior.

Anomalies are detected by analyzing prediction losses. Given the high precision of the model, deviations from expected results indicate anomalies with high confidence. To minimize noise, we set the reconstruction error of 95% or higher as our threshold to report anomalies, filtering out insignificant cases.

Table 4.2 Preliminary results and comparisons.

DataSet	Accuracy	Recall	Precision	Overhead	Data Size	Training Time
Critical path state count and duration	90.24%	92.30%	85.97%	5.59%	280.3 KB	35.92s
System call count and duration	91.46%	89.12%	84.33%	25.92%	576491.01 KB	3952.3s

To accurately locate anomalies within the execution trace, we employed a time-window approach to pinpoint the specific time interval in which a thread exhibited anomalous behavior. By segmenting the data into fixed time windows, we enhanced the precision of anomaly localization. After extensive experimentation with different time intervals, we determined that a 100-second window provided the most effective results. To enhance the reproducibility of our research, we have made the source code and data files publicly available on GitHub.[1].

4.4 Experiment results

As shown in Table 4.2, our model achieved high precision in detecting anomalies within the dataset. Compared to the system call based approach, our critical path-based method demonstrated a comparable detection accuracy while significantly reducing overhead to approximately 5%. This reduction in overhead is particularly beneficial for real-time anomaly detection, as it minimizes performance impact while maintaining detection reliability.

Figure 4.4 provides a more detailed analysis of anomaly detection, where events exceeding the threshold line are classified as anomalies. For example, at timestamp 1,917, our model flagged an anomaly. Upon further examination using the dataset, we observed that at this timestamp, the thread entered the INTERRUPTED state for an extended period, caused by 10 concurrent reporting processes running simultaneously. Considering the distribution of anomalies over time and the noise injection in the data from [83] used to train this model, it appears that, despite some false positives, the model is generally sensitive to the noise points, and its detections align well with them.

This result highlights the model's ability to accurately pinpoint the exact time anomalies occur. Such information can be utilized to dynamically adjust tracing focus, allowing a tracer to concentrate on specific areas of the code and identify the root cause of performance issues more effectively.

As shown in Figure 4.5, the t-SNE projection of our test data demonstrates the effectiveness

[1]https://anonymous.4open.science/r/ICPE2025-DataChallenge-240C/

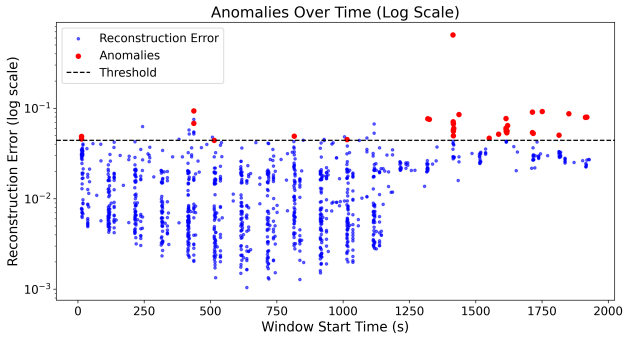

Figure 4.4 Early detection of abnormal behavior.

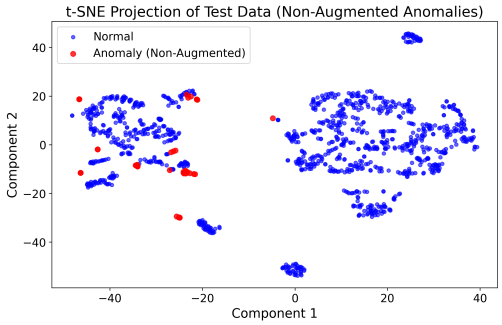

Figure 4.5 t-SNE projection of the test data.

of our model in distinguishing between normal and anomalous execution states. Normal data points (blue dots) form well-defined clusters, confirming that our model has successfully learned stable patterns in system behavior. Anomalous points (red dots) are largely isolated or positioned on the outskirts of normal clusters, indicating that our model can effectively detect outliers. However, a few anomalies appear within normal clusters, suggesting the presence of borderline cases that may lead to false positives. Although our experiments show that increasing the reconstruction threshold to 97% significantly reduces these false positives, we decided to prioritize having more false positives over missing true positives. The clear separation between clusters demonstrates that our model recognizes distinct operational modes, though certain states are more prone to anomalies.

4.5 Discussion

In this study, we performed anomaly detection using both state count vectors and duration vectors. To ensure a comprehensive comparison, we also collected system call traces under identical load conditions, as used in [79]. Table 4.2 summarizes the results, comparing

anomaly detection based on critical path traces and system call traces.

Our findings show that the data size generated by our method is significantly smaller than using system call traces, making it more suitable for real-time anomaly detection. Additionally, while system call vectors have a size of 628-ary, our critical path vectors are only 28-ary, drastically reducing the data volume while maintaining comparable detection accuracy to system call based methods.

The primary goal of our approach is to enable real-time early detection while supporting large-scale tracing. Experimental results indicate that critical path event traces significantly reduce trace analysis time and system resource consumption, addressing two key challenges in large-scale real-time tracing: (a) Minimizing tracing overhead (b) Enabling efficient online tracing

Given these advantages, critical path event traces offer a scalable and efficient alternative to system calls for real-time anomaly detection and adaptive tracing.

4.6 Conclusions and Future Work

In this paper, we proposed a deep learning-based method for early detection of runtime software issues. Our approach achieves accuracy comparable to existing methods while introducing two key innovations: (1) focusing on software dependencies instead of system calls for better long-term issue detection, and (2) requiring fewer events to collect critical path data, reducing trace collection time and system overhead.

For future work, we plan to explore execution graph data with other ML methods used in system call analysis to identify the most efficient approach for early anomaly detection. Additionally, leveraging state sequences and dependency graphs for anomaly detection could further improve accuracy. We also aim to test our method in industrial settings to evaluate its adaptability to code changes, workload variations, and performance analysis.

A key objective is to develop an end-to-end adaptive tracing approach, integrating our detection method to enhance large-scale system tracing. We are also researching dynamic tracing profiles to allow the tracing controller to adjust horizontally across nodes and processes and vertically within execution layers.

Finally, we plan to extend our work to resource-constrained systems, such as IoT devices, where efficient performance management is critical. Our lightweight approach can significantly enhance runtime efficiency and reliability in these environments.

CHAPTER 5 ARTICLE 2 SOFTWARE PERFORMANCE PROBLEM LOCALIZATION: A GOAL-BASED ADAPTIVE TRACING APPROACH

Authors: Masoumeh Nourollahi, Naser Ezzati-Jivan, Adel Belkhiri, Fatemeh Faraji Daneshgar, Michel Dagenais

Submitted to: Concurrency and Computation : Practice and Experience. 2024-08-29.

abstract: As modern applications continue to increase in size and complexity, execution tracing has become essential for diagnosing performance-related issues. However, the high computational costs associated with trace collection and analysis often hinder its widespread adoption, especially in dynamic environments where applications evolve rapidly. Traditional tracing methods, while useful, struggle to balance efficiency and thoroughness, making regular performance diagnosis a costly endeavor.

In this paper, we propose a novel adaptive tracing approach that leverages the hierarchical structure of software performance anti-patterns (SPAs) to optimize tracing efforts. By initially focusing on high-level performance issues, our method significantly reduces the size and overhead associated with whole-system tracing. Once a problem area is identified, the tracing is adjusted to diagnose the issue with greater precision. We validate our approach using the DeathStarBench social network application, addressing four distinct performance challenges. Our results demonstrate a substantial reduction in trace sizes while maintaining or improving the effectiveness of performance problem detection in most scenarios.

Keywords: software analysis, performance evaluation, adaptive execution tracing, performance anti-patterns.

5.1 Introduction

The complexity and size of modern applications have made diagnosing performance problems extremely difficult. Logging and tracing are two widely used methods for identifying software issues [8, 9]. Logging provides a record of events that occur during the execution of an application, which can be useful for diagnosing problems but may not capture all the necessary details [69, 86, 87]. Tracing, on the other hand, tracks and records the execution flow and performance metrics of an application in much finer detail, making it a powerful tool for diagnosing performance issues [88]. However, tracing overhead is a significant challenge for large applications. To mitigate this, selective instrumentation and trace sampling methods were introduced, which control the volume of traces but at the cost of reducing the diagnostic

capability. Therefore, a method that provides a balance between collecting enough traces to effectively diagnose performance problems and minimizing the overhead associated with tracing [54, 89, 90].

To address this challenge, many adaptive approaches were proposed [90–93]. These approaches adjust tracing points to align with the ever-changing environment (e.g., workload changes, code changes, updates, deployment changes, resources changes) and conform to the new requirements [90, 93–96]. Performance problems often arise in a system when there is a change in workload patterns, application code, deployment strategies, or resource constraints [97–99]. Adaptive methods focus on these changes to locate and diagnose issues. Some of the adaptive approaches work on reducing tracing overhead, while others focus on capturing more effective traces.

Methods aimed at reducing trace overhead can be classified into two categories. Some focus on selective instrumentation [92, 94, 100, 101], while others leverage sampling to achieve the same goal. For example, the Dapper [8] and Facebook Canopy [9] tracing frameworks are based on head-based sampling methods. The sampling decision in these methods is taken mostly randomly, which has a high risk of missing valuable edge-case traces. On the other hand, there are frameworks [10, 11, 23, 24] that use tail-based sampling methods, which impose a large tracing overhead.

Furthermore, another strategy in adaptive tracing focuses on collecting more effective traces for performance diagnosis, with the aim of keeping a balance between the quality of the recorded traces and tracing costs. Effective traces in software performance tracing refer to the collected data that accurately and efficiently capture the behavior of software applications during their execution. These traces provide critical insights into the performance characteristics and interactions within the system, enabling developers to diagnose issues, optimize performance, and ensure reliability [9, 102]. Measurement of some metrics within the gathered traces can provide valuable insight into system performance. Indeed, calculating the appropriate metrics that accurately reflect the states of the observed system is crucial for making informed sampling and instrumentation decisions [29, 52]. Furthermore, to move towards performance diagnosis, some approaches have developed workflow-centric tracing techniques [9], [103], [8], [104] to diagnose performance problems.

Performance regression testing is a method used to ensure that recent code changes have not negatively impacted the performance of a software application. This type of testing aims to identify any deterioration in performance metrics such as response times, throughput, and resource usage following updates or modifications to the codebase. According to Hasnain et al. [105], performance regression testing involves using models such as recurrent neural networks to forecast and ensure that web services performance remains stable despite code

changes. This testing helps maintain service quality by detecting potential performance issues early in the development cycle. Elbaum, Malishevsky and Rothermel [106] emphasize the importance of prioritizing test cases in regression testing, which includes performance regression tests. They highlight how prioritizing test cases based on their impact can help in identifying performance regressions more efficiently. The taxonomy of SPAs can be used as a performance test priotization method, by starting from high-level SPAs and going down to more specific problems. This can help in having a starting point after software changes, by priotorizing highest level of the SPA taxonomy in regression tests.

Tracing system performance immediately after a change, be it at the software or hardware level, can help ensure a seamless transition, by locating newly introduced performance issues. However, in some cases, tracing the whole system might be impractical due to the significant overhead this operation incurs. This limitation, coupled with adaptive tracing approaches, has led us to propose a framework that involves an adaptive strategy focused on gathering more effective traces. Since performance issues are usually expected after environmental evolution, this approach utilizes a hierarchical search to uncover these problems.

This framework provides an initial tracing setup to start monitoring efforts of the changed application without adversely affecting the performance of the system. Based on the performance problems taxonomy presented in [59], our framework systematically looks for possible high-level software performance anti-patterns (SPA). It uses a combination of workload characterization and selective tracepoint activation to identify signs of performance anti-pattern. The idea here is to understand what each performance problem is and how it may be displayed in the system. In this way, we can focus our tracing efforts on identifying these signs. This framework reduces the cost of tracing by initially gathering higher-level traces to confirm or deny the existence of performance issues through specific symptoms. It then provides feedback to the tracing agent, identifying potential sources of the observed issue. This feedback is used to adapt the tracing to locate the source of the problem. It does so by going further down a specific branch of the performance problems taxonomy and gathering more traces that are focused on a smaller scope related to the identified issue, to observe symptoms of finer-grained problems that are linked to known causes. This method ensures gathering traces that are enough to diagnose the problem, and at the same time not gathering every trace possible. Tracing overhead is reduced by selectively activating tracepoints and increasing efficiency by focusing on specific problem symptoms. In summary, the following are the main contributions of this work:

- Development of a goal-based adaptive approach: We introduce a novel adaptive tracing method centered on predefined performance anti-pattern detection goals. This goal-driven approach enhances the efficiency and accuracy of tracing by focusing on critical

performance issues, thereby minimizing unnecessary data collection and reducing the associated overhead. The strategy aligns tracing efforts with the most relevant aspects of system performance, ensuring a more targeted and effective diagnostic process.

- Leveraging hierarchical search for SPAs: Our method employs a structured hierarchical search to detect potential SPAs, particularly following software changes. This hierarchical approach allows tracing to be conducted in a layered manner, starting with broader, high-level SPAs and progressively narrowing down to more specific issues. This method helps in precise and targeted tracing while also reducing the overall cost of tracing activities by confining the search to relevant levels of the hierarchy.

The rest of the paper is structured as follows. Furthermore, section 5.2 discusses related work in three categories, including performance problems hierarchy, adaptive tracing strategies, and performance problem localizing methods. Next, the proposed goal-based adaptive approach is presented in section 5.3. Section 5.4 evaluates and completes our method through experiments for each SPA, and then discusses the results. Finally, in section 5.5, we conclude the study, discuss threats to the validity of our method, and present our ideas for future research directions.

5.2 Related Work

This section presents previous works in three main categories, including performance problem hierarchy and details on each performance problem, strategies for adaptive tracing and performance problem localization.
Modern software applications are perpetually increasing in complexity, rendering the diagnosis of performance issues more challenging. Tracing serves as a technique for tracking and diagnosing these problems within software systems [66]. However, managing the overhead of tracing itself poses a significant challenge, imposing substantial system overhead during collection and intricate computational complexity during analysis. To tackle this challenge, selective instrumentation and trace sampling methods have emerged, aiming to alleviate overhead by not capturing every possible trace. Nevertheless, these approaches may lack the necessary information for effective problem diagnosis [54,90]. In response to these challenges, adaptive approaches have emerged, addressing the dynamic nature of software environments to gather more efficient traces. These methods adapt to the changing system conditions induced by workload variations [104], code updates, or resource alterations [95, 96]. Performance issues often arise following changes in the environment [97–99, 104], with adaptive tracing methods specifically targeting these changes to identify emerging problems.

5.2.1 Performance problems hierarchy

Several SPAs are defined in the literature [66, 68, 107]. Wert et al. [59] categorize these SPAs into a taxonomy. The major advantage of using this taxonomy is that it provides a chain of causes and symptoms of SPAs, which can be traced by moving from the root node to a leaf node. As a result, by moving from the top levels of the taxonomy to the lower levels, finer-grained SPAs can be detected. We utilized this taxonomy as our reference for adaptive tracing. Our focus in this work is on SPAs identifiable through user-space tracing, excluding those requiring kernel tracing, which will be addressed in future studies. Thus, the SPAs taxonomy depicted in Figure 5.1 serves as our reference. Therefore, we have provided a brief description of each SPA below.

Continuously violated performance requirements. Instances of performance require-

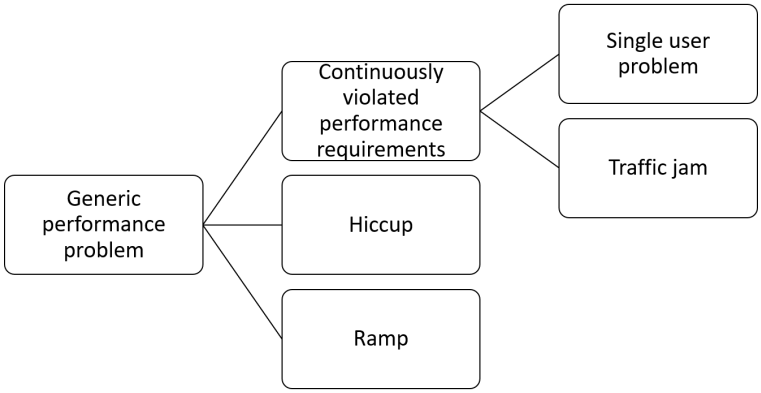

Figure 5.1 Software Performance anti-patterns taxonomy.

ments violations signify underlying performance issues. If response time serves as an indicator of performance requirements, the Continuously violated performance requirements heuristic seeks deviations from the response time criteria.

Single user problem. In Smith [69] definition the single user problem SPA refers to a situation where a system or application exhibits performance issues even when only a single user is interacting with it. This type of problem typically indicates fundamental inefficiencies or bottlenecks in the system architecture, code, or configuration that prevent it from performing adequately under minimal load conditions.

Hiccup. The application *hiccup* anti-pattern refers to a behavior where the application ex-

periences periodic, temporary spikes in response time. Typically, this anomaly arises due to recurring tasks like log writes, garbage collection, periodic deprecation tasks, and similar activities [108]. Detecting the existence of this anti-pattern within the application involves monitoring fluctuations in the application response times.

Traffic jam. In software performance engineering, a *traffic jam* anti-pattern refers to a situation where excessive concurrent requests overwhelm a system resource, causing significant delays and degraded performance.

Ramp. The *ramp* anti-pattern occurs when the processing demand to fulfill a request gradually escalates over time. This situation leads to a deterioration in performance over time, despite the load remaining relatively stable.

5.2.2 Adaptive tracing strategies

Adaptive tracing strategies can be categorized into three main groups: methods aimed at reducing tracing overhead, approaches concentrating on gathering more effective traces, and studies focused on localizing performance issues. These strategies are further elaborated upon in the subsequent discussion.

Tracing overhead reduction

Efforts aimed at minimizing tracing overhead primarily fall into two categories: selective instrumentation and trace sampling. Selective instrumentation involves activating tracepoints aligned with specific tracing objectives [92, 94, 100, 101]. Yu et al. proposed an efficient adaptive instrumentation method in [109], facilitating the detection of data race conditions. In another work by Lehr et al. [29], a framework was introduced to automate an iterative process to refine compiler instrumentation using user-defined heuristics for C/C++ with the Clang compiler. PIRA [50] empirically models and predicts the performance of parallel programs, particularly MPI programs.

On the other hand, trace sampling techniques, including head-based and tail-based sampling methods, aim to reduce overhead by selectively capturing traces. Dapper [8] and Facebook Canopy [9] provide head-based sampling methods, which often involve random sampling decisions and carry a risk of missing valuable edge-case traces. Other methods such as those presented in [10], [23], [24], and [11] offer tail-based sampling, making sampling decisions based on metrics like latency thresholds or page faults. While these approaches generate detailed traces, they tend to incur substantial tracing overhead and address trace storage

challenges by recording only the sampled traces. Adaptive instrumentation and sampling methods generally aim to reduce both trace collection and analysis overhead. However, they may sometimes lead to the collection of low-quality traces that offer limited insight into performance degradation.

The Balance between quality and cost

Balancing the quality of recorded traces to gather more effective data has been a key focus in previous studies [29, 52]. Performance measurement techniques, which involve defining metrics to gauge system performance, play a pivotal role in making decisions regarding trace sampling and instrumentation. For instance, workflow-centric methodologies, like those found in [9, 103], track changes in workflows to aid in diagnosing performance issues. QDIME [52] focuses on measuring Quality of Service (QoS) parameters, while others rely on static code analysis, as described in [29]. In a study by Sambavisan et al. [55], three widely recognized visualization approaches are compared to automatically conduct root cause analysis. Malik et al. [56] utilize performance counter data from a load test to create performance signatures, pinpointing the subsystems responsible for performance violations. Furthermore, in [57], four machine learning approaches are introduced and evaluated to aid performance analysts in more effectively comparing load tests to detect performance deviations. StackMine [58] also contributes to enhancing performance debugging by narrowing down the scope of call stack trace analysis. Khan, and Ezzati [63] introduce an adaptive tracing approach that adjusts the tracing level by selectively instrumenting tracepoints at runtime. To achieve this, they initially conduct stress testing to obtain expected execution times for methods and system calls in the test application. During runtime, the application is monitored to detect changes in the execution time trends. If the change exceeds a certain threshold, an anomaly score is updated, indicating whether tracing for that specific component should be enabled or disabled.

5.2.3 Performance problem localization

To effectively diagnose performance issues, a comprehensive understanding of potential problems and their associated symptoms is crucial. This knowledge, when coupled with adaptive tracing techniques, significantly aids in performance diagnosis. Smith et al. [66] concentrate on recognizing common flaws in software design that often result in performance issues. This forms a fundamental basis for determining what aspects should be traced and mea-

sured. Surya et al. [60] utilize the second-order Markov model to look for signs of resource contention to identify this performance issue and its source. Conversely, systematic experimentation studies, such as those outlined in [59], establish controlled experiments to isolate and comprehend the root causes of performance decline. Moreover, this research introduces a taxonomy of performance issues, assisting in directing efforts toward identifying these problems. This hierarchical structure offers a systematic approach to diagnosing SPAs, enabling a comprehension of the interrelations among various problems and aiding in prioritizing them based on their impact on system performance. Furthermore, [65] offers a method to detect a variation of the *one-lane-bridge* performance problem. In addition to identifying latencies caused by incorrect use of multi-threading, it also discovers latencies introduced by overuse of active resources. To achieve this, they monitor metrics extracted from system events.

By pinpointing issues across different levels of detail, system analysts can concentrate their efforts on specific areas or components, making the troubleshooting process more efficient and targeted. This method is used in [64] for detecting *ramp* and *traffic jam* performance anti-patterns. In addition, [61] uses a systematic review of related literature to present a taxonomy of real-world performance bugs, which includes their effects, causes, and contexts. They also provide a collection of real-world performance bugs. Furthermore, other works like [62] study performance anti-patterns on specific software systems.

While the aforementioned works contribute significantly to the field of performance problem diagnosis, they have several limitations. Traditional tracing methods often fail to balance trace quality with overhead, leading to either excessive data or insufficient diagnostic information. Selective instrumentation and trace sampling methods attempt to mitigate overhead but may overlook critical traces necessary for accurate diagnosis. Furthermore, adaptive tracing strategies, although promising, often require complex configurations and can still impose substantial overhead under certain conditions.

Our proposed approach aims to address these limitations by integrating adaptive tracing with a hierarchical problem detection approach. This integration allows for dynamic adjustment of tracing levels based on the application behavior, improving the balance between trace quality and overhead. By focusing on specific performance problems and adapting to changes in the application environment, our method enhances the efficiency and effectiveness of performance diagnosis. This transition highlights the need for a more refined and adaptable approach to tracing, which our proposed method seeks to fulfill.

5.3 Adaptive Goal-based Approach for Localizing Performance Problem

Adaptive tracing is instrumental in diagnosing performance issues by dynamically adjusting the extent and depth of tracing in response to the current system state. Its primary objective is to minimize the overhead of tracing while efficiently identifying performance problems. The initial phase of adaptive tracing involves determining the starting point for tracing efforts and guiding subsequent adjustments in tracing levels to diagnose software performance issues. As discussed above, commencing tracing within complex systems poses a significant challenge. Our objective is to propose a method for establishing the initial tracing setup, aiding in the identification of essential traces required to swiftly identify performance problems.
Figure 5.2 provides an overview of the adaptive goal-based tracing process. First, a tracing goal is determined, and subsequently an initial tracing setup is established to achieve that goal. Next, relevant traces are then collected and subjected to analysis. This analysis leads to measurement of metrics associated with the tracing goal. Finally, the results are cross-checked with the tracing goals to determine whether the information collected is sufficient to locate the problem. If further tracing is necessary, feedback is conveyed to the tracing agent, allowing adjustments in the tracing setup, including its scope and collection method, to systematically pinpoint the source of the regression. Considering the adaptive goal-based

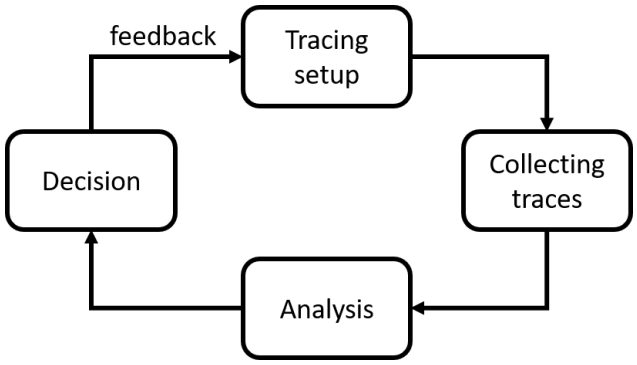

Figure 5.2 Overview of the adaptive goal-based tracing process.

tracing approach, we propose an adaptive goal-based tracing method, which utilizes SPA detection as its tracing goal, and uses their definitions to determine trace collecting method, metric definition, and decision making on adapting the tracing setup.

5.3.1 Goal-based tracing approach

Detecting performance regression sources in micro-service applications is complicated. Focusing on detecting software performance anti-patterns facilitates this task by providing

specifications about the regression we are looking for. Each SPA definition comes with its symptoms and best practices to resolve it. Therefore, setting the goal of tracing to detect performance anti-patterns instead of simply looking for performance regressions has the following potential benefits:

- It facilitates the definition of the regression metric, because the symptom of each SPA points to the metric that should be measured to detect it.
- The solution to resolve SPA is part of its definition and is extensively discussed in the literature. By detecting SPAs developers are also guided through their resolution.
- SPAs are defined hierarchically. Using them gives us the benefit of breaking the performance regression detection problem into finer-grained problems, by looking into higher level sPAs at first, and then traversing through the taxonomy of SPAs to the lower level, more specific problems. performance Goal-based tracing facilitates this task

Adaptive goal-based tracing approach influences the tracing process in two key ways. Firstly, it involves adaptations to tracing setup, required to pinpoint the targeted SPA within the application. A tracing setup is defined for each SPA. This setup includes the tracing scope and tracing frequency. Tracing scope includes selecting instrumentation points or tracing levels, whether in user space or kernel space, to identify the intended SPA. Tracing frequency refers to the interval between each trace sample.
Secondly, it considers the application workload in time of trace collection, by characterizing the execution load to efficiently detect the issue, recognizing that different problems may surface under varying system loads.
Figure 5.3 provides a more detailed overview of our methodology. To pinpoint the source of observed performance regression symptoms, the hierarchy of SPAs is leveraged as a decision tree. Traversing it in multiple tracing iterations will guide us to reach the source of observed symptoms. For each SPA, two key factors aid in their identification: workloads capable of manifesting the problem and the detection heuristic that investigates the traces for signs of the existence of the SPA.
Consider a SPA called *traffic jam*, as an example. The *traffic jam* SPA occurs when excessive requests overwhelm a system resource, leading to significant delays and reduced overall system performance. This happens when multiple processes or threads compete for limited resources, such as CPU, memory, or I/O bandwidth, creating a bottleneck [110]. The primary symptom of a *traffic jam* is an excessive increase in response times with increasing system load. To detect this problem, the traces are gathered under a gradually increasing workload, at various intervals, to compare response times changes by increasing load, and determine if

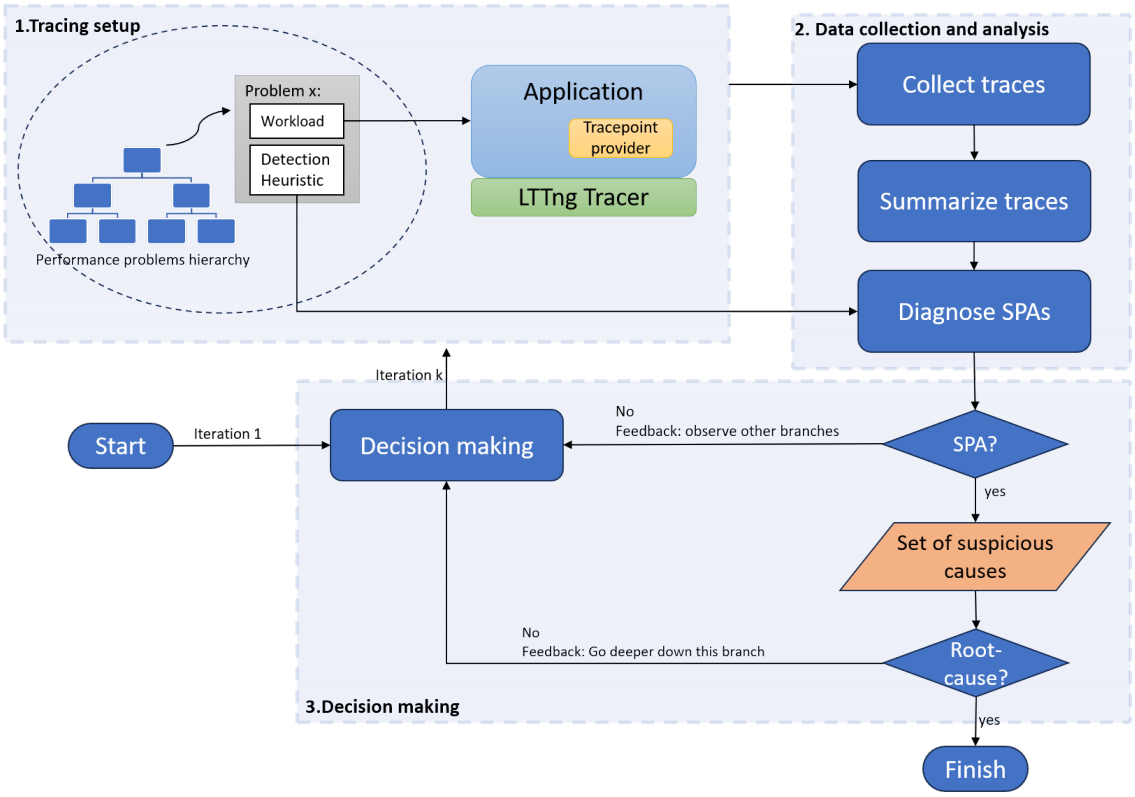

Figure 5.3 Performance problems localization via adaptive goal-based tracing method.

there are signs of a traffic jam.

Using appropriate workloads provides the conditions that will result in surfacing SPAs in the application, if they exist. Returning to the method, we use this knowledge to adapt the tracing setup for each SPA. We define a workload plan for each SPA. This facilitates the collection of valuable selective traces. In the first tracing iteration, starting from the root of the SPA hierarchy, the goal is to determine whether there is a continuously violated performance requirements in the application. Workload recommendations are employed to detect this SPA on the application under test. Simultaneously, the application is instrumented with LTTng tracepoints to enable user-space trace collection using liblttng-ust. LTTng (Linux Trace Toolkit Next Generation) is chosen for its high performance and low overhead, making it ideal for detailed and efficient tracing of complex applications [111]. The recommended workloads are applied to the application, allowing for trace collection. These traces are then summarized into an application call stack, and response time summaries are calculated for each call.

The next step involves diagnosing the SPA. Detection heuristics specific to each SPA are utilized to determine whether the traces exhibit signs of the intended SPA. If no SPA is de-

tected, the investigation moves to an adjacent branch in the taxonomy to restart the process. If a SPA is identified, the method provides a set of tracepoints with the highest likelihood of being sources of the SPA. These suspicious causes are investigated, determining whether the cause of the SPA is among them. If yes, the search for the SPA concludes. If not, feedback is provided for the next iteration of tracing to further explore the problem hierarchy in the same branch, potentially pinpointing the cause. These heuristics and identification of suspicious causes are discussed in detail in 5.3.3 section.
In the *traffic jam* SPA example, the next iteration involves identifying whether the problem is a hiccup, a *ramp*, or a continuously violated performance requirement. To detect the *traffic jam* SPA, we required three iterations of tracing.
The rest of this section provides further details about each step of our proposed method with an example of a *traffic jam* SPA.

5.3.2 Tracing setup

In this section, we use *traffic jam* SPA to explain the proposed method. *traffic jam* SPA can be caused by multiple factors, including competition for software and hardware resources (e.g., synchronization points, locks, CPU). Our focus in this work is on software bottlenecks. Figure 5.4 [59] illustrates bottleneck impact on application response time and CPU utilization. As the figure shows, whether there is a resource or software bottleneck, with increasing load on the application response time increases. However, in case of software bottleneck resource utilization does not change significantly. Identifying and mitigating this performance issue is crucial to ensure the reliability and efficiency of software applications.

As discussed, the workload used to test the application in the time of trace collection sig-

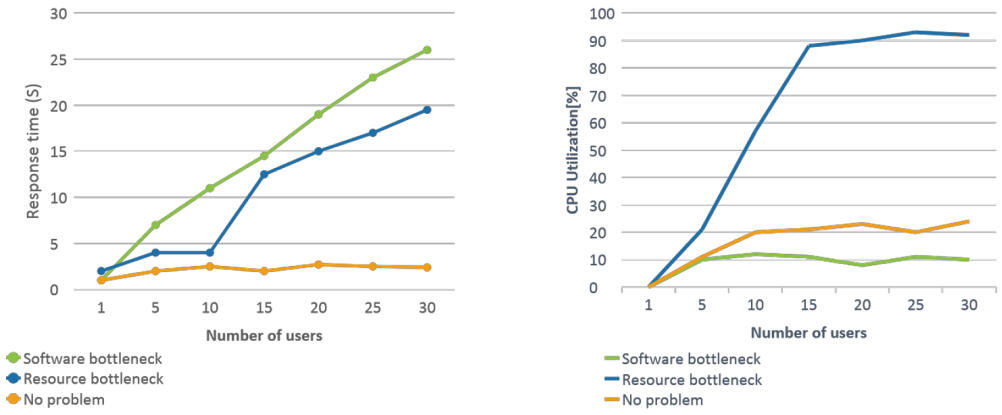

Figure 5.4 Impact of a bottleneck on a web application response time and resource utilization.

nificantly influences the resulting traces. Often, *traffic jam* SPA surfaces in heavy loads. To

detect it we need to have several stages of putting load on the application and collecting traces. Figure 5.5 depicts tracing setup required for collecting the required traces.

Trace collection can occur at either the kernel or user-space level, necessitating a decision

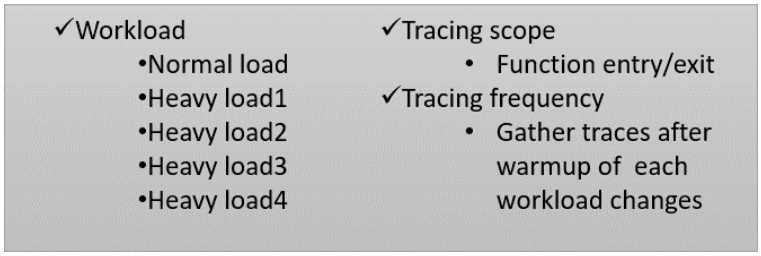

Figure 5.5 Tracing setup for *traffic jam* SPA.

based on the specific information needed to identify the target performance anti-pattern. This paper exclusively addresses patterns that can be identified by user-space tracing, with an intent to investigate kernel and user-space trace integrations in future research. We collected function call stacks, which is a common practice, providing ample detail to identify the issue. Nevertheless, collecting all call stacks in larger contemporary applications is usually unfeasible. To overcome this challenge preprocessing tracing points before trace collection is essential. Functions like abstract definitions often contribute little to valuable trace information. Thus, it is prudent to first discard less critical tracepoints and then focus on tracing the more significant functions to avoid major system performance degradation. Earlier research, as referenced in [112], provides methodologies for this preprocessing, though it falls outside the scope of this paper. In situations where these practices are not enough to overcome tracing overhead, we recommend starting at broader levels, such as the request level, and executing multiple tracing cycles to more accurately isolate problem areas.

Nowadays most microservice applications already come with distributed tracing instrumentation, like OpenTelemetry [113] and OpenTracing [114]. These tools can be used to trace the application at the request level. Gelle et al. [115] present a method to combine OpenTracing with LTTng traces. Tracing levels can be adjusted based on specific needs, aligning with the current detection goals. For precise problem localization, developers may opt to trace at lower levels of detail, like the basic block level or each loop in the code.

Regarding the choice of instrumentation methods, we opted for static instrumentation, due to its greater simplicity, compared to dynamic instrumentation, as it avoids the complexities associated with modifying code at runtime. It is important to note that our method is independent of the chosen instrumentation approach. Therefore, if developers opt for dynamic instrumentation methods and tools, such as uprobe [116] or dtrace [117], our methodology remains applicable. GCC offers the *-finstrument-functions* tool [118], which is a compiler

option in GCC, to automatically instrument function entry and exit points. Then function entry and exit traces are collected by LTTng. If there is no direct access to source code, scripts can be used that patch the source code to insert instrumentation.

5.3.3 Data collection and analysis

The collected trace data is summarized by extracting the happens-before relations among the function calls in the call stack. Eclipse Tracecompass [85], which is a tool for reading and analyzing traces and logs of a system, is used to build a call stack of functions. Two main outputs are extracted from the call stack graph: first, a set of response times for each edge of the call stack; second, repeating sequences of the edges. The trace summarization steps are depicted in Figure 5.6. The function happens-before relationship graph shows the association between two functions, namely the caller and the callee.

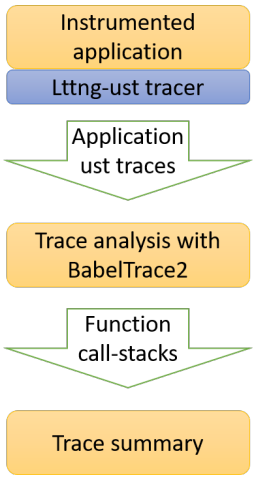

Figure 5.6 Trace summarization steps.

Each event instance 'i' is defined as follows:

$$event_i = (funcId, eventType, timeStamp, ThreadId, ProcessId) \tag{5.1}$$

The relationships among functions are extracted from the call stack. As a result, each function instance within our data model has the following description:

$$func_i = (funcId, callerId, entryTs, exitTs, duration, threadId) \tag{5.2}$$

Moreover, we establish edges within the happens-before relation graph based on function data. Within graph 'G', every function instance represents a node, while the association

between two functions denotes an edge. In this graph, an instance of an edge 'i' is defined as follows:

$$edge_i = (edgeId, sourceId, targetId, entryTs, exitTs, duration) \tag{5.3}$$

A happens-before graph is denoted as a directed acyclic graph G=(V,E,R), where V represents a set of function calls, E signifies directed edges linking two functions, and R signifies the time duration between the entry of the first function and the exit of the second function. Additionally, we conducted further analysis on the resulting happens-before graph, extracting repeating sequences of function calls from this graph. These sequences provide valuable insights for investigating suspicious functions. As function calls occurring before the execution of a particular function can contribute to observed problems, tracking these sequences help complement the set of suspicious functions.

We leveraged the SPA taxonomy and detection heuristics discussed in previous works [59, 64, 66, 67, 119] to identify SPAs. However, to facilitate goal-based adaptive tracing through this method, we adapted their suggested methods. The algorithms heuristic related to each SPA detection is presented in the following subsections. Detecting deviations in software performance relies on response time requirements. As a result of previous phases, we now have access to function calls data, which is used in this phase to analyze the traces and detect SPAs.

Before delving into details on detection methods, we need a common representation of the data extracted from the traces. The following formalizes the data that is used in our detection algorithms:

Traces refer to the collected call stack traces, which include function-entry/function-exit traces.

Caller-Callee Relationship CCR_i represents the relationship between caller and callee functions. This indicates which functions are calling which other functions during the execution.

Mean Duration of Edge is the average response time for a particular function call, calculated by:

$$\mu_i = \text{mean}(RT_i), \text{ where} RT_i \text{is the response time for edge } i. \tag{5.4}$$

Threshold is the performance threshold for edge i using factor k. It is calculated by:

$$T_{i,k} = T_{mid_i} \times \text{Threshold Factor}_k \tag{5.5}$$

The **Threshold Midpoint** is the mean duration used as a baseline for setting performance thresholds. It is calculated by:

$$T_{mid_i} = \mu_i \tag{5.6}$$

To create a range of performance thresholds the following multiplicative factors are used. The **threshold factors** used are: $\{10^{-10}, 10^{-9}, 10^{-8}, 10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10^{1}, 10^{2}, 10^{3}, 10^{4}, 10^{5}, 10^{6}, 10^{7}, 10^{8}, 10^{9}, 10^{10}\}$.

Occurrence Difference is the time difference between consecutive entries for edge i. It is calculated by:

$$\Delta t_{i,k} = t_{entry}(e_{i,k+1}) - t_{entry}(e_{i,k}) \tag{5.7}$$

The **Mean Occurrence Difference** is the average time difference between consecutive entries for the edge i. It is calculated by:

$$\mu_{\Delta t_i} = \text{mean}(\Delta t_i) \tag{5.8}$$

The **Moving Percentile Size** is the size of the moving window to calculate the percentiles, where n_{occ} is the number of occurrences. It is calculated by:

$$MP_i = \mu_{\Delta t_i} \times n_{occ} \tag{5.9}$$

The **Cumulative Count** is the count of problem flags for edge i. It is calculated by:

$$C_i = \text{size}(\text{has-problem-flag}_i == \text{True} \tag{5.10}$$

The **Cumulative Ratio** is the ratio of the occurrences of problems to the total size of the edge i. It is calculated by:

$$CR_i = \frac{C_i}{\text{size}_i} \tag{5.11}$$

$RT_{i,j}$ denotes the **Response time** for function i during experiment j.This represents the time taken for a specific function to respond during different load tests. The Sample mean is the average response time of the observations in the set, calculated by:

$$\bar{S}_j = \frac{1}{n_j} \sum_{k=1}^{n_j} RT_{i,j,k} \tag{5.12}$$

where n_j is the number of observations in S_j. Furthermore, **Sample standard deviation** measures the amount of variation or dispersion in the response times. It is calculated by:

$$s_j = \sqrt{\frac{1}{n_j - 1} \sum_{k=1}^{n_j} (RT_{i,j,k} - \bar{S}_j)^2} \tag{5.13}$$

T-statistic is used to compare the means of two sets of response times to determine if they are significantly different. It is calculated by:

$$t = \frac{\bar{S}_j - \bar{S}_{j+1}}{\sqrt{\frac{s_j^2}{n_j} + \frac{s_{j+1}^2}{n_{j+1}}}} \tag{5.14}$$

In addition, **degrees of freedom** is used in statistical tests to determine the critical value of the t-statistic. It is calculated by:

$$df = \min(n_j, n_{j+1}) - 1 \tag{5.15}$$

Moreover, **critical Value** $t_{\alpha/2,df}$is the threshold beyond which the difference between the means is considered statistically significant. The critical value from the t-distribution, for a given significance level α and degrees of freedom df.
The following subsections discusses each anti-pattern detection algorithm in two main categorise of threshold base and load test results comparison methods.

Threshold based performance antipattern detection

Violations of performance requirements indicate a performance antipattern. Accordingly, if the response time is an indicator of performance requirements, then the threshold based performance antipattern detection method looks for cases that violate the response time requirements. Previous work like [59, 64] assume the response time requirements are provided by the users, which is not possible in most cases. Algorithm 1 presents the method used for threshold based detection of performance antipatterns. This algorithm is used for *Continuously violated performance requirment* and *hiccup* SPAs detection method. The following steps are taken to calculate the maximum acceptable response time threshold for each function, to eliminate the need for user input.

- Record call stacks traces for all functions involved in the intended requests by putting

single operation and mixed operation normal load on the application

- Repeat the above step several times to get the usual response times. In our experiments we repeated this step 100,000 times.
- Calculate the average response time across these experiments for each function call. This calculated value is utilized to establish the anticipated response time average requirements for each function call in standard conditions. This data is used to calculate response time thresholds for each function call. This threshold assists in determining if an execution response time surpasses the expected value.
- For each function caller and callee pair use the *threshold factors* defined in the equations above to calculate the threshold for each function. we refined our analysis by incorporating a spectrum of thresholds established through the formula $X*10^n; -10 =< n <= 10$. We calculated the test results using these thresholds. Then, we compared the SPA diagnoses across each threshold to select the most appropriate one for each SPA case.
- The threshold based performance problem detection algorithm 1 is applied for each threshold, to find the problematic function calls. By use of cumulative ratio of repetition of each observed problem in this algorithm, suspicious function calls are ranked. Finally, in order to determine which threshold was the best choice for this problem, the results of each threshold are compared against the expected results.

Algorithm 1 Threshold Base Performance Antipattern Detection Algorithm

1: Run load on the intended service
2: $Traces \leftarrow$ Collect call stack traces
3: $CCR \leftarrow$ Traces
4: Calculate performance requirement threshold CCR_i
5: Calculate $\mu_i \leftarrow \text{mean}(RT_i)$ $edge_i$
6: $T_{mid_i} \leftarrow \mu_i$
7: $T_{factors} \leftarrow \{10^{-10}, 10^{-9}, 10^{-8}, 10^{-7}, 10^{-6}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10^1, 10^2, 10^3, 10^4, 10^5,$
8: $10^{-5}, 10^6, 10^7, 10^8, 10^9, 10^{10}\}$
T_{factor_k}
9: $T_{i,k} \leftarrow T_{mid_i} \times T_{factor_k}$
10: **Software performance antipattern detection**
11: $C_i \leftarrow$ size(has-problem-flag$_i$ == True)
12: $CR_i \leftarrow \frac{C_i}{\text{size}_i}$
13: $CR_{set} \leftarrow CR_i$
14: Performance-problem-ranking $\leftarrow$ Sort(CR_{set})

Continuously violated performance requirement. To detect this SPA, we used algorithm 1. Algorithm 2 provides details of the "Software performance antipattern detection"

step in line 16 of algorithm 1. In this method the response time of each function call is checked against the expected response time threshold to determine whether that function is suspicious of *continuously violated performance requirement* SPA. Then in step 3, the cumulative SPA ranking method is used to determine if this is a continuous problem.

Algorithm 2 Continuously Violated Performance Requirement Detection

Step 1: Calculate performance requirement threshold $edge_i$ T_{ik} $\{RT_{i,j} < T_{ik}$
$has - problem - flag_{ik} \leftarrow False$
$has - problem - flag_{ik} \leftarrow True$
Step 2: *Continuously violated performance problem* detection
Mark this function i as experiencing Hiccup performance anti-pattern
Step 3: Performance antipattern ranking

Hiccup. *Hiccup* SPA causes periodic response time spikes. To detect the presence of this SPA we should track application response time changes. [59] provides an algorithm to detect application hiccups. We modified this algorithm by adding steps represented in 1, first to calculate response time thresholds, and finally to rank the detected suspicious functions. The core of the main heuristic is provided in alroigthm 3. In step 2 (lines 3-9) of this algorithm, we used the fixed time-based width moving percentile method. To calculate the moving percentile size, we need to calculate the mean time between the occurrence of each edge, and also determine the number of occurrences that we want to consider in each percentile. Therefore, for each edge in the call graph, first, we calculate the duration between occurrences of the edge and then calculate the mean duration of occurrences for that edge. Then we determine the moving percentile size for each edge individually, by considering the mean occurrence time of that edge multiplied by the number of occurrences for each percentile. Then, in step 3 (lines 10-16) this moving percentile is used to detect suspicious functions for *hiccup* SPA.

Algorithm 3 Hiccup Performance Problem Detection

Step 1: Calculate performance requirement threshold
Step 2: Calculate moving percentile size $edge_i$ $edge_{i,k}$
$\Delta t_{i,k} \leftarrow t_{entry}(edge_{i,k+1}) - t_{entry}(edge_{i,k})$
$\Delta t_i \leftarrow \Delta t_{i,k}$
$\mu_{\Delta t_i} \leftarrow \text{mean}(\Delta t_i)$
$MP_i \leftarrow \mu_{\Delta t_i} \times n_{occ}$ T_k MP_i
Step 3: *Hiccup* detection
Mark this function i as experiencing *Hiccup* performance anti-pattern
Step 4: Performance antipattern ranking

Load test comparison based performance antipattern detection

Traffic jam and *ramp* SPAs are detected by load test comparison based SPA detection method. Unlike the threshold based detection method, in load test comparison based method, instead of ranking SPAs by comparing response times with a threshold, the response times of functions are seen as a set, and compared with the results of other load tests to detect SPAs. Looking at the response times as a set facilitates capturing propagation of SPAs in other function calls. Algorithms 4 and 5 that are respectively used for *Traffic jam* and *ramp* SPAs detection both follow three main steps. However, they differ in the response time datasets that are used in each step. The following details these algorithms.

Traffic Jam. We used a method called the time window strategy [59] to detect *traffic jam* SPA. This method aims to detect the SPA by analyzing response time series from multiple load tests. Algorithm 4 details the huristic to detect *traffic jam* SPA. The input for this algorithm is the response time series for each function from multiple load tests. The following steps are executed: Step 1, perform pairwise comparisons of the response time series from neighboring experiments. Step 2, for each pair of neighboring experiments, gather response time sets of each experiment. Bootstrap the response time sets to ensure that they are normally distributed. Compute sample means and standard deviations. Calculate the t-statistic to compare the means. Determine degrees of freedom. Compare the t-statistic with the critical value from the t-distribution. Step 3, if the comparison shows a substantial increase in response time across all sets, mark the function call as experiencing the *traffic jam* SPA. The output is a set of function calls suspicious of exhibiting the *traffic jam* SPA.

Ramp. Ramp usually happens after long times of application execution. Considering this, designing appropriate long running load tests is a key in *ramp* SPA detection. Algorithm 5 details steps followed to detect this SPA. The main steps of this algorithm is the same as algorithm 4 for *traffic jam* detection. Here again we use the time window strategy to separate sets of response times for comparison proposes. However, the main difference between these two algorithms lies in which set of response times are compared with each other. In *ramp* detection response time should show an all time growing pattern. So, to detect *ramp*, several experiments are conducted over time. As mentioned in step 1, in each experiment beginning and end response times sets are compared together.

This section described the proposed goal-based tracing method, serving as an initial stage for tracing endeavors aimed at identifying indications of performance problems. This methodol-

Algorithm 4 Traffic Jam Performance Problem Detection - Time Window Strategy

Input: Response time series $\{RT_{i,j}\}$ for each function i from multiple load tests j
Output: Detection of *traffic jam* performance anti-pattern each function i
Step 1: Pairwise comparison
Conduct pairwise comparison of the response time series $\{RT_{i,j}\}$ of neighbouring experiments j and $j+1$ each pair of neighbouring experiments $(j, j+1)$
Step 2: Bootstrap and t-test
Gather response time sets $S_j = \{RT_{i,j}\}$ and $S_{j+1} = \{RT_{i,j+1}\}$
Bootstrap S_j and S_{j+1} to ensure normal distribution
Compute sample means $\bar{S}_j$ and $\bar{S}_{j+1}$
Compute sample standard deviations s_j and s_{j+1}
Compute t-statistic:

$$t = \frac{\bar{S}_j - \bar{S}_{j+1}}{\sqrt{\frac{s_j^2}{n_j} + \frac{s_{j+1}^2}{n_{j+1}}}}$$

Determine degrees of freedom:

$$df = \min(n_j, n_{j+1}) - 1$$

Compare t with critical value $t_{\alpha/2,df}$ where df is the degrees of freedom $|t| > t_{\alpha/2,df}$
Step 3: *traffic jam* detection
Mark this function i as experiencing *traffic jam* performance anti-pattern

ogy assists in concretely defining the extent of the initial tracing process by concentrating on the objective of recognizing signs related to a predetermined performance anti-pattern. In the subsequent section, we have included sample experiments and the outcomes yielded by these experiments. Through this phase, we identify a set of suspicious functions that might be contributing to the performance problem.

5.4 Results and Discussion

This section delves into the experiments conducted to assess the proposed method, offering specific instances to illustrate its practical application. Initially, we detail the experimental setup and outline our testbed. Subsequently, we adapt the heuristics outlined in section 5.3.3 for four performance anti-patterns to identify problem indicators that are aligned with our objectives. Following this, we apply the method to the designated testbed, presenting the obtained results, including ranking of suspicious function calls.

Algorithm 5 Ramp Performance Problem Detection

Input: Response time series $\{RT_{i,t}\}$ for each function call i over time t during load tests
Output: Detection of *ramp* performance anti-pattern each function call i
Step 1: Pairwise comparison
Conduct pairwise comparison of the response time series $\{RT_{i,t}\}$ from the beginning and end of the load test each pair of neighboring time intervals (t_{start}, t_{end})
Step 2: Bootstrap and t-test
Gather response time sets $S_{start} = \{RT_{i,t_{start}}\}$ and $S_{end} = \{RT_{i,t_{end}}\}$
Bootstrap S_{start} and S_{end} to ensure normal distribution
Compute sample means $\bar{S}_{start}$ and $\bar{S}_{end}$
Compute sample standard deviations s_{start} and s_{end}
Compute t-statistic:

$$t = \frac{\bar{S}_{start} - \bar{S}_{end}}{\sqrt{\frac{s^2_{start}}{n_{start}} + \frac{s^2_{end}}{n_{end}}}}$$

Calculate degrees of freedom:

$$df = \min(n_{start}, n_{end}) - 1$$

Compare t with critical value $t_{\alpha/2,df}$ where df is the degrees of freedom $|t| > t_{\alpha/2,df}$
Step 3: *Ramp* detection
Mark this function call i as experiencing *Ramp* performance anti-pattern

5.4.1 Experimental Setup

We evaluate the proposed method using DeathStarBench [120], an open-source benchmark suite for microservices released in 2019. This application is widely used and cited in research regarding microservice applications. It is an active project, that is recent and still supported by its development team. Moreover, it is complicated enough to emulate real-world cases, by using different programming languages, and state of the art technology. We deployed the application containers on an Intel Core i7-6700K machine equipped with 64GB DDR4 RAM, running Ubuntu 18.04.6 LTS Bionic.

For load generation in our experiments, we employed the Wrk2 load generator [121], an HTTP benchmarking tool known for its ability to create substantial load on web servers. Furthermore, this tool offers the flexibility to specify various parameters such as throughput (requests per second), number of threads, and others, which are essential for tailoring our load tests to our specific needs. Furthermore, an LTTng [12] agent was deployed within each application container, allowing individual tracing without the need to correlate traces with their respective containers. Several tools like eBPF [122], DTrace [117], Perf [123], and LTTng [12] support userspace tracing. Since we are already familiar with LTTng tracer, and it

offered us the flexibility of tracing on microservice applications that use different programming languages, we used LTTng to collect traces.

Our focus for tracing goals is on SPAs detectable solely through user-space traces. Future works will cover SPAs that might require kernel-level traces or a combination thereof. Nevertheless, the methodology remains universal, and applicable to all traces regardless of their source.

Fault injection is a pivotal consideration throughout our experiments. We intentionally injected faults into the application to evaluate our test results concerning the successful detection of the injected SPAs. While the application might inherently possess some performance anti-patterns, without the injected faults, evaluating performance of the algorithm becomes challenging. Table 6.6 describes the performance bugs we injected into the application to evaluate our approach. The table showcases our implementation of five problems including continuously violated performance requirement problem, application hiccups, *traffic jam*, single user problem, and *ramp* SPAs. To detect these SPAs, as discussed in the previous section, we define the tracing setups required to collect traces that would facilitate the surfacing of the intended SPA. This includes the development of specific trace collection scripts aligning with the load test and tracing objectives for each ASP. The column "Injected fault" in Table 6.6 provides insight into the injected faults.

Table 5.1 Description of the performance bugs injected into the application under test.

Performance anti-pattern	**Impacted service**	**Injected fault**
Generic performance problem	ComposePost	Delay in 20% of cases on “WriteUserTimeline" function
Continously violated performance requirement	ComposePost	Changed code to not release a lock mutex
Single user problem	ComposePost	Delayed code by changing "UploadUserMentions" function execution time to 5 seconds
Hiccup	UserTimeline	Delay in 20% of cases on “WriteUserTimeline" function
Traffic jam	ComposePost	Changed code to not release a lock mutex
Ramp	ComposePost	No changes

Furthermore, our experiments demonstrate proficiency of the method in diagnosing SPAs. Subsequent subsection detail our findings regarding each SPA studied and their corresponding evaluation outcomes.

5.4.2 Experiments and Results

Multiple experiments were conducted on the DeathStarBench social network application, under various load profiles, capturing user-space level traces for each container. Information regarding our test setup, test data, and code is available in our github[1] page. Subsequently, a more intricate tracing strategy was devised, based on the identified SPAs. The ensuing section provides comprehensive insights into the experiments carried out.

Table 5.2 lists the results of our experiments. In total, 21 test cases were executed, where each one exhibiting zero, one, or multiple SPAs. This table uses true positive, true negative, false positive, and false negative metrics to indicate whether the proposed approach successfully detected the SPA in the test case.

Following is the list of our test cases: TC1: long time executing function-base load; TC2: long time executing function-normal load; TC3: long time executing function-heavy load; TC4: long time executing function-stress load TC5: periodically long time executing function-normal load; TC6: periodically long time executing function-heavy load; TC7: periodically long time executing function-stress load; TC8: no fault-base load, TC9: no fault-normal load; TC10: no fault-heavy load, TC11: no fault-stress load; trace gathering in base load; TC12:lock mutex not released fault- base increasing load; TC13:lock mutex not released fault- base increasing load TC14: start of large log file generation- normal load; TC15: start of large log file generation- heavy load; TC16: 2 hour after large log file generation- normal load; TC16: 1 hour after large log file generation- normal load; TC17: 2 hour after large log file generation- heavy load; TC18: 10 hour after large log file generation- normal load; TC19: 10 hour after large log file generation- heavy load; TC20: 20 hour after large log file generation- normal load; TC21: 20 hour after large log file generation- heavy load.

In addition to each test result, this table presents accuracy and precision for each of the studied SPAs. Accuracy is the proportion of correctly classified instances (both true positives and true negatives) among the total instances. It is calculated by the following formula:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{5.16}$$

Precision is the proportion of true positives among the instances classified as positive. It is calculated by the following formula:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.17}$$

[1] https://github.com/mnourollahi/UST$_a$*daptiveTracing*/

Sample fault injection and SPA detection experiment

In this section, we provide details on one of our experiments. Same as section 5.3, to avoid repetition, here we only present details of our experiments for the *traffic jam* SPA.

To validate our approach, we performed fault injection to simulate a *one-lane-bridge traffic jam* SPA. This SPA occurs when a system or application requires all processing or data to pass through a single constrained resource. This leads to a bottleneck where the resource becomes a point of congestion, causing significant delays and reduced throughput [124]. This involved intentionally locking a read mutex within the *compose post* service, resulting in clients being blocked behind this barrier. The release and reacquisition of the read mutex are part of the "waitForWork()" function, indicated by the code snippet "sync_.waitForWork(seqid)".
Identifying this issue required three tracing iterations. Initially, we pinpointed a continuously violated SPA using the aforementioned method. Subsequently, in the second phase, we conducted two additional load tests – base and heavy – to determine whether the problem was a *traffic jam* or was isolated to a single user. Our findings suggested a *traffic jam*, yet the exact source remained unclear at this stage. The ranking of *traffic jam* issues identified during this phase is outlined in Table 5.3. This ranking presents the suspicious function calls that may be the source of the observed SPA. These function calls are sorted from the most probable culprit to the least probable one.

To further narrow down our search, we excluded unbalanced processing as a cause for the performance regression, given that the pattern of response time changes did not align with it. The next potential causes the performance regression that we considered were CPU or database congestion, which was beyond the scope of our study, and the one-lane bridge problem. A one-lane bridge typically stems from a software bottleneck rather than a blockage of hardware resources. Such a bottleneck might be due to a database lock, synchronization resources, or a service bottleneck. Based on our *traffic jam* analysis results, we employed the happens-before event graph derived from the call stack, during the trace summarization phase, and the identified *traffic jam* candidates for further examination. This observation revealed that instances of the *dispatchCall* function, followed by read or write requests, occurred in 11 distinct function call sequences. The sequences of calls identified at this stage are listed in Table 5.4, and are our main suspects for the one-lane bridge SPA. We then traced the synchronization and database accesses in these functions to pinpoint the root cause, which turned out to be the *process_UploadUniqueId* function.

Table 5.2 Results of SPA detection with goal-based adaptive tracing approach- TP indicates true positive, TN indicates True negative, FP indicates false positive, FN indicates false negative.

Test case	**Generic performance problem**	**Continously violated performance requirement**	**Single user problem**	**Hiccup**	**Traffic jam**	***ramp***
TC1	TP	TP	TP	TN	TN	TN
TC2	TP	TP	TP	TN	TN	TN
TC3	TP	TP	TP	TN	TN	TN
TC4	TP	TP	TP	TN	TN	TN
TC5	TP	TN	TN	TP	TN	TN
TC6	TP	TN	TN	TP	TN	TN
TC7	TP	FP	TN	TP	TN	FP
TC8	TN	TN	TN	TN	TN	TN
TC9	TN	TN	TN	TN	TN	TN
TC10	TN	TN	TN	TN	TN	TN
TC11	TN	TN	TN	TN	TN	TN
TC12	TN	TN	TN	TN	TN	TN
TC13	TP	TP	TN	TN	TP	TN
TC14	TN	TN	TN	TN	TN	FN
TC15	TN	TN	TN	TN	TN	TN
T16	TN	TN	TN	TN	TN	TN
TC17	TN	TN	TN	TN	TN	TN
TC18	TP	TN	TN	TN	TN	FN
TC19	TP	TN	TN	TN	TN	TP
TC20	TP	TN	TN	TN	TN	TP
TC21	TP	TN	TN	TN	TN	TP
Accuracy	1	0.95	1	1	1	0.81
Percision	1	0.89	1	1	1	0.75

Table 5.3 Traffic jam performance problem candidates ranking.

Test case	**(Service, Function)**	
Traffic jam	(HomeTimelineServiceProcessor, dispatchCall) (MediaServiceProcessor, dispatchCall) (SocialGraphServiceProcessor, dispatchCall) (GetFollowers_result, write) (Getfollowers_result, read)	height

Table 5.4 Potential candidates of the one-lane bridge issue event sequences.

Test case	Function event sequences	
One-lane bridge	'dispatchCall', 'process_UploadCreator', 'read' or 'write' 'dispatchCall', 'process_UploadMedia', 'read' or 'write' 'dispatchCall', 'process_UploadUserMentions', 'read' or 'write' 'dispatchCall', 'process_UploadText', 'read' or 'write' 'dispatchCall', 'process_UploadUniqueId', 'read' or 'write' 'dispatchCall', 'process_ReadPosts', 'read' or 'write' 'dispatchCall', 'process_GetFollowers', 'read' or 'write' 'dispatchCall','Process_UploadCreatorWithUserId', 'read' or 'write' 'dispatchCall', 'process_WriteUserTimeline', 'read' or 'write' 'dispatchCall', 'ReadPosts', 'read' or 'write' 'dispatchCall', 'process_UploadText', 'read' or 'write' 'dispatchCall', 'process_UploadUrls', 'read' or 'write' 'dispatchCall', 'process_UploadUserMentions', 'read' or 'write'	height

5.4.3 Discussion

Our approach focuses on tracing software performance problems detectable through user-space traces. Fault injection played a key role in our experiments, as we intentionally injected faults into the application to evaluate our test results concerning the successful detection of the injected SPAs. Our method demonstrated effective detection of various SPAs at the user-space level, including continuously violated performance requirement problems, application hiccups, traffic jams, single-user problems, and ramp SPAs. The results show that our approach can categorize and identify performance issues, guiding engineers to detect and address these problems more effectively.

An important component of our method is the use of goal-based adaptive tracing, a dynamic approach that adjusts the level of tracing detail based on the observed system behavior while focusing efforts in a specific direction. Execution tracing, goal-based adaptive tracing, and SPA (Software Performance Anti-patterns) detection are interconnected. Goal-based adaptive tracing helps in detecting anti-patterns by adjusting the tracing detail to capture relevant performance metrics when specific conditions are met, based on predefined goals. For example, if a latency spike is detected, goal-based adaptive tracing can increase the granularity of traces to identify the root cause, whether it be a traffic jam, hiccup, or resource contention. This focus allows for more efficient and effective detection of performance issues by directing tracing efforts where they are most needed.

To evaluate the effectiveness of our method, we experimented with four SPAs across three levels of the performance problem hierarchy, using a well-established microservice-based bench-

mark testsuite. The confusion matrix in Figure 5.7 illustrates the detection accuracy of our method, with high true positive rates for the injected SPAs. This demonstrates the robustness of our approach in identifying performance issues. However, for the Ramp SPA, the detection accuracy does not perform as well as for other anti-patterns. Nevertheless, it still shows promising accuracy compared to other works, such as [59].

Additionally, we compared the tracing overhead between our method and full Line of Code

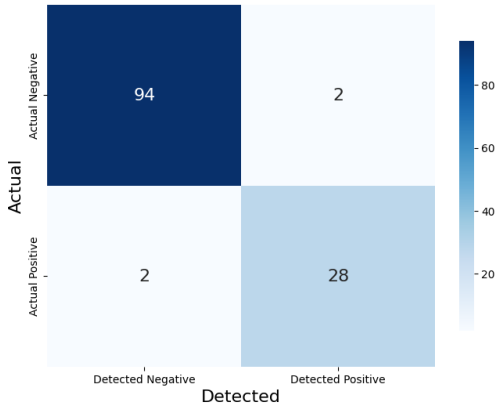

Figure 5.7 Confusion matrix indicating detailed breakdown of the performance of the adaptive goal-based tracing approach.

(LOC) tracing. As shown in Figure 5.8, our method significantly reduces the tracing overhead, making it more efficient and less intrusive. This is important in practical scenarios where minimizing overhead is essential to maintain system performance.

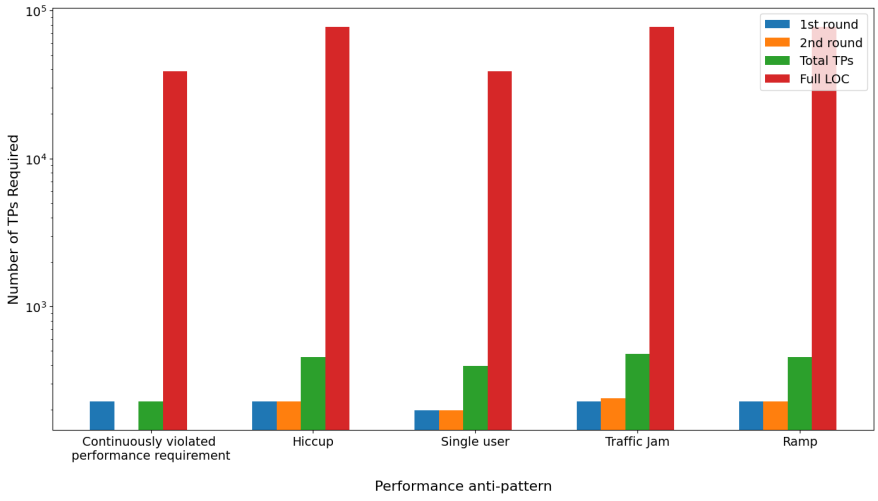

Figure 5.8 Comparison of tracing overhead between our method and full Line of Code (LOC) tracing to identify the studied problems.

Furthermore, Table 5.5 presents the number of iterations of goal-based adaptive tracing required to detect each SPA. The results indicate that our method requires fewer iterations to detect SPAs compared to traditional methods, thereby reducing the diagnostic effort and speeding up the problem resolution process.

Table 5.5 Number of iterations of goal-based adaptive tracing required to detect each SPA.

Performance anti-pattern	**Number of tracing iterations**
Generic performance problem	1
Continuously violated performance requirement	2
Single user problem	2
Hiccup	2
Traffic jam	3
Ramp	2

Potential Threats to Validity: Considering potential threats to the validity of this method, it is possible that collecting function entry/exit traces, or call stacks, may be too expensive in certain systems. To address this concern, our method remains adaptable, enabling initial tracing at preferred levels (be it on request, component, or service level, and advancing towards targeted tracing at finer-grained levels) for issue diagnosis. Another potential validity concern is the completeness of the performance problems hierarchy employed in our method. This can be addressed by enhancing the hierarchy with additional problems, without affecting the validity of the original method.

However, our method has several other limitations. It does not include kernel tracing, which restricts its ability to detect SPAs involving kernel threads or requiring kernel-level information. For instance, detecting a "CPU Intensive Application," a subset of traffic jam problems, requires kernel tracing to provide information on CPU utilization. Without kernel-level data, our method cannot detect such SPAs. Additionally, the method relies on predefined heuristics for SPA detection, making it somewhat ad-hoc and potentially missing systematic capture of all types of performance issues.

Despite these limitations, our method is particularly useful when the performance problem resides in a function implementation. Kernel traces do not capture events occurring solely within a function, but our method does. It can detect latency and categorize high-level SPAs causing this latency, such as hiccups or traffic jams, making it a valuable tool for performance analysis, especially for problems that occur at the application level.

5.5 Conclusions and Future Directions

This paper introduces an approach designed for diagnosing performance issues during system changes, with a primary focus on addressing the problem of tracing overhead. Traditional approaches to performance problem detection often result in excessive data collection and analysis, making them inefficient and time-consuming. Our method leverages a hierarchical approach to performance problem detection, optimizing trace gathering by focusing on tracing setup, including tracepoints and application load test workloads specifically associated with distinct performance problems. By doing so, we minimize unnecessary tracing overhead and improve diagnostic efficiency.

We evaluated our approach by experimenting with four SPAs across three levels of the performance problem hierarchy, using a well-established microservice-based benchmark testsuite. This approach enables performance engineers to employ a goal-based tracing strategy for efficient problem detection in applications, resulting in reduced effort. Furthermore, it allows for rapid high-level validation of changed applications without the need for extensive trace collection and analysis. Simultaneously, in diagnosing detected issues, this method facilitates targeted tracing efforts, guiding engineers to detect performance issues more effectively.

While this paper concentrates on performance problems detectable by user-space traces, our future work aims to extend the scope to include more SPAs by integrating kernel-space tracing or distributed tracing solutions like OpenTracing. Additionally, our approach can be further improved by constructing a knowledge base of performance problems, updated with each newly diagnosed problem, serving as an expert guide in the problem detection procedure. Moreover, machine learning and deep learning methods like LLMs (large language models) can be used in combination with this knowledge base to more effectively diagnose the application.

By addressing these limitations and expanding our approach, we hope to provide a more comprehensive tool for performance analysis, capable of diagnosing a wider range of SPAs in microservice applications. This will further enhance the ability of engineers to detect, categorize, and address performance issues effectively.

5.6 Acknowledgements

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Ciena, Ericsson, and EffciOS for funding this research project.

5.7 Declaration of Competing Interest

The author declares that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

5.8 Declaration of Generative AI and AI-assisted Technologies in the Writing Process

The authors used ChatGPT-4O to assist with paraphrasing and refining the grammar of this manuscript. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the published article.

CHAPTER 6 ARTICLE 3 EFFICIENT DETECTION OF COMMUNICATION-RELATED PERFORMANCE ANTI-PATTERNS IN MICROSERVICES

Authors: Masoumeh Nourollahi, Naser Ezzati-Jivan, Adel Belkhiri, Michel Dagenais
Submitted to: Journal of Software : Evolution and Process. 2025-08-15.

abstract: Modern microservice-based applications are inherently complex due to their distributed nature and intricate service interactions, making performance diagnosis challenging. A significant source of degradation in such systems is Software Performance Anti-patterns, which can arise from inefficient coding practices, poor architectural design, or suboptimal deployment strategies. Existing Anti-pattern detection methods often rely on costly, intrusive data collection, limiting their practicality in production environments.

We propose a low-cost, non-intrusive approach for detecting communication-related Anti-patterns by combining selectively captured networking-related system call traces with distributed observability traces. This hybrid tracing enables detailed correlation between low-level communication metrics and high-level service interactions, supporting accurate detection of Anti-patterns such as Blob and Empty-semi-trucks. To minimize overhead, only essential system call events are collected, while distributed tracing data is used for root cause analysis at the service and operation level.

We design a machine learning pipeline, supporting supervised, semi-supervised, and unsupervised detection, achieving up to 91% accuracy with just 2.74% data collection overhead. Offline model training further supports seamless integration into Continuous Integration/Continuous Deployment (CI/CD) workflows for early performance regression detection.

Our approach is validated on the DeathStarBench microservice benchmark across 14 scenarios under both clean and noisy conditions. Results show high detection accuracy, effective root cause identification (over 80% match to manual analysis), and minimal runtime impact. This work offers a practical, scalable, and accurate solution for SPA detection in complex microservice environments.

Keywords: software performance anti-patterns, tracing, system calls, kernel tracing, distributed tracing, semi-supervised learning.

6.1 Introduction

Microservice-based applications have revolutionized software design, allowing applications to scale by decoupling functionalities into modular services. While this architecture improves flexibility and scalability, it introduces considerable complexity in performance management. In distributed microservice environments, services interact in intricate ways, which can lead to communication bottlenecks and performance degradation that are difficult to diagnose and resolve. These issues often stem from Software Performance Anti-patterns (SPAs) [1]; common design and implementation flaws that impede performance due to inefficient coding, flawed architectural design, or poor deployment practices [56, 72]. Addressing these SPAs, particularly under high-load scenarios, is critical for maintaining efficient and reliable systems.

In practice, detecting SPAs is a challenging task due to the high data collection costs and the intrusiveness of traditional monitoring methods. Conventional techniques such as profiling, deep tracing, and network dumps impose significant overhead, making them less suitable for production environments, especially those requiring real-time monitoring. Furthermore, these methods often necessitate extensive manual effort or access to source code, limiting their practicality in fast-paced industrial settings. Consequently, many existing approaches to SPA detection are infeasible for deployment in industries where applications need to maintain high performance with minimal operational disruption. Our industrial partners in the telecommunications sector exemplify these requirements: telecommunications systems need to maintain peak performance with reliable diagnostic tools that can identify issues quickly and fit seamlessly into CI/CD workflows. These industry-driven constraints call for solutions that are both low-overhead and easily integrated into continuous deployment cycles.

In response, this research introduces a low-cost, non-intrusive approach to detect communication-related SPAs in microservices. Our method addresses several key gaps in existing SPA detection approaches. First, to mitigate the high cost of data collection, we employ a selective tracing approach that minimizes the amount of traced data, targeting only necessary system-call events related to networking alongside distributed tracing data. This approach reduces overhead and aligns with the need for cost-efficient solutions in production environments. Second, to overcome the limitations of purely statistical methods, we utilize machine learning techniques to classify system behavior and identify SPA patterns. Specifically, our supervised learning model enables accurate differentiation between normal and SPA-related behaviors, achieving up to 91% accuracy. This classification, combined with low data collection overhead, makes the approach viable for real-time applications and suitable for deployment in CI/CD pipelines. Third, we study the feasibility of using supervised, semi-supervised, vs

[1] In this paper, the abbreviation "SPA" is used for Software Performance Anti-pattern.

unsupervised learning to detect SPAs. Fourth, we extended the work on microservices SPA detection by going one step further, using a call graph analysis-based method to detect the root cause of the observed SPAs. Finally, we concluded our work by proposing a workflow that can be used in software changes to compare the new software version to the previous one and make sure no SPAs are added to the new version.

To validate the proposed method, we conducted experiments on the DeathStarBench microservice testbed, a widely recognized benchmark that simulates realistic microservice scenarios and SPA conditions. These experiments demonstrate that the minimal data collection of our method is sufficient to detect SPAs, while maintaining low system overhead. Moreover, our approach leverages existing OpenTracing instrumentation for root-cause analysis, allowing us to identify not only the presence of SPAs but also the specific services and operations responsible. This functionality is essential in industrial contexts, where rapid root cause analysis can enable swift remediation of performance bottlenecks. By training our model offline and operating it in an online detection mode, we create a solution that is efficient and adaptable to continuous deployment pipelines. The pre-trained baseline of the model enables proactive detection of new SPAs introduced through software updates, making it practical for CI/CD workflows and continuous performance monitoring.

In summary, this paper presents a low-cost, non-intrusive approach for detecting communication-related SPAs in microservices that addresses critical challenges in performance diagnosis within production environments. Our main contributions are as follows: (1) a selective, low-overhead tracing approach that effectively combines system-call and distributed tracing data for SPA detection; (2) a machine learning-based classification model for high-accuracy SPA identification and root-cause localization; (3) a feasibility study of using supervised, vs semi-supervised, vs unsupervised learning; and (4) a solution optimized for integration into CI/CD workflows, enabling early SPA detection in continuous deployment settings. This work contributes to the field of performance engineering by offering a practical, scalable, and effective SPA detection solution tailored to meet the demands of complex, high-performance microservice applications.

The remainder of this paper is organized as follows: Section 6.2 covers background on SPAs and tracing techniques. Section 6.3 reviews SPA detection methods, including statistical and machine learning approaches. Section 6.4 outlines our methodology, while Section 6.5 presents the experimental setup and results. Section 6.6 addresses practical implications and limitations, emphasizing industrial applicability. Finally, Section 6.7 summarizes our contributions and future research directions.

6.2 Background Information

SPAs describe commonly occurring poor coding and design practices, along with solutions to resolve them [125]. This section first discusses background information related to communication SPAs. Then software tracing concepts are explained.

6.2.1 Communication-related SPAs

SPAs document common software performance problems and their solutions [67]. SPAs are one of the sources of performance degradation in applications. Unlike variables such as workload changes and resource bottlenecks that impact application performance, SPAs often result from fundamental application problems like poor coding practices, suboptimal architecture, or improper deployment configurations. Although SPAs are always present in the system, they may impact application performance only under high-load scenarios.

Smith et al [66] introduce several SPAs, such as the God Class and Excessive Dynamic Allocation. In that work, he describes the impact of these SPAs on software systems and provides solutions to resolve them. This research was extended by introducing additional SPAs, including Concurrent Processing Systems, Pipe and Filter Architecture, and Falling Dominos [67, 68]. Later, more SPAs were introduced and categorized in other works, such as [64,70]. Both [126] and [59] present a hierarchical classification of SPAs. Furthermore, [127] provides a classification of SPAs based on the place in which the performance problem manifests in code. They provide four main categories, including local implementation problems, micro-architecture problems, macro-architecture problems, and deployment problems. This classification is further used to provide SPA fault injection methods.

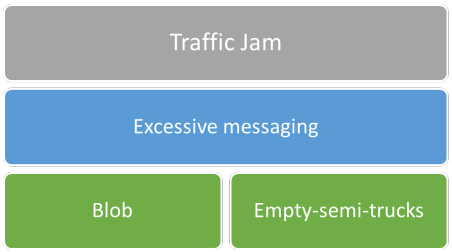

Figure 6.1 Communication SPAs within SPAs hierarchy.

In this work, we focus on one specific category of SPAs, namely communication SPAs that result in excessive messaging. The Blob SPA refers to a central service that either contains a large portion of the system logic, acting as a controller by processing data and sending instructions to other services, or holds most of the data required by other services, serving as

a data repository with little logic of its own. In both cases, this results in excessive messaging involving that service. The Empty-semi-trucks SPA occurs when a service transmits too many small messages instead of aggregating them. In this case, the ratio between message overhead and payload is too high, resulting in poor performance [70]. Figure 6.1 provides a view on the placement of communication SPAs within the SPA hierarchy proposed by [59].

Wert et al. [70] provide a method to detect SPAs resulting in high response times, focusing on inter-component communication SPAs in two categories: SPAs that result in excessive messaging, namely Blob and Empty-semi-trucks; and SPAs resulting in high database overhead, including Stifle and Circuitous Treasure Hunt. Trubiani et al. [72] utilize software model testing to identify and resolve SPAs related to communication inefficiencies like Blob. Additionally, Avritzer et al. [71] provide a queue-based method to detect several SPAs, including Blob and Empty-semi-trucks. Although all the aforementioned works detect communication SPAs, they do not leverage the specific properties of these SPAs to reduce tracing and data collection costs.

6.2.2 Software Tracing

[21] Tracing approaches vary significantly in focus and capability. Kernel tracing techniques are highly effective for diagnosing a wide range of bugs, as the kernel typically has insight into an application's critical activities through system-calls or traps. However, kernel tracing alone can be insufficient, especially in microservice applications, where numerous requests are processed across multiple services, making it challenging to trace application execution solely from the kernel's perspective. To address this, user-space tracing was introduced to track traditional applications more effectively [21]. For distributed microservices, additional effort is required to correlate traces across all nodes. Distributed tracing captures the complete journey of a request as it passes through each service within a distributed system [128], with projects like OpenTelemetry and OpenTracing [113, 114] supporting this approach. Hybrid tracing [115] combines distributed tracing with kernel-level tracing to provide both extensive traceability and low-level details.

In summary, communication-related SPAs significantly impact microservice applications, particularly under high load. Tracing and analyzing system-calls can offer valuable insights into these SPAs, although current methods often involve intrusive data collection. This discussion sets the stage for related work, which investigates existing SPA detection methods and highlights their limitations in minimizing data collection and overhead.

6.3 Related Work

In this section, SPA detection is explored in three main categories: statistical methods for detecting SPAs, performance modeling practices for identifying SPAs, and the use of machine learning methods. Previous works on detecting SPAs can be categorized into three main approaches: statistical methods, performance modeling, and machine learning techniques.

6.3.1 Statistical Methods

[62] proposes a statistical approach to detect SPAs in cyber-physical systems. This is one of the earliest works on detecting SPAs; however, it lacks automation and cannot be used in live environments. Wert et al. [59, 129] propose a SPA diagnostics approach based on

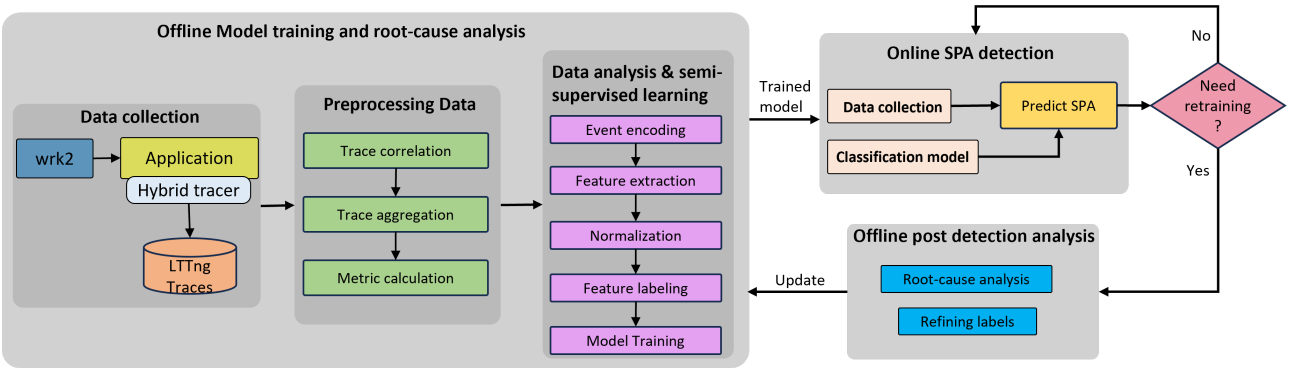

Figure 6.2 High-level architecture of the proposed approach.

systematic experimentation for enterprise software systems, along with a taxonomy of performance problems. Their method detects SPAs with good accuracy but relies on various software traces and logs, such as user-level traces or network dumps, which can be intrusive and costly to collect. These methods are intrusive, expensive, and require access to source code.

Trubiani [72] introduces an approach that leverages load testing and profiling results to identify SPAs and refactor software to resolve them. By using characteristics of collected operational profiles to produce representative workloads and employing profiler tools to gather performance data, statistical methods are used to match these measurements with SPA specifications.

VanDonge et al. [65] propose a method to detect the N-Lane Bridge SPA using system-level traces. They also use statistical analysis in their detection method; however, the main difference between their method and others is that they benefit from system-level traces, which

are less intrusive than previous works and do not require access to source code. Stadelmaier et al. [73] propose an automatic profiling approach to detect SPAs, which also suggests refactoring solutions to resolve them.

6.3.2 Performance Modeling

Cortellessa et al. [119] propose an automatic model-driven approach using Architecture Description Languages (ADLs) to enable SPA detection based on software architecture. However, their approach lacks detailed information about software components, limiting its generalizability for SPA detection. Sanctis et al. [75] detect and resolve SPAs by building architectural models, providing architectural alternatives to fix identified issues. The authors in [130] propose a tool that detects SPAs in design language models of application architectures. Avritzer et al. [71, 76] use a multivariate analysis approach to characterize SPAs and employ a mapping function to detect them, utilizing queue-based models to calculate response times.

In studies such as [71, 76, 131], specific counters are chosen to monitor the presence of each SPA. These works use queuing performance modeling techniques to calculate response times for each request, measuring performance through network interface metrics. Statistical metrics, including mean, variance, standard deviation, and the second moment of service load values, are calculated to establish response time baselines. In addition, Chalawadi et al. [132] use basic runtime data to detect SPAs. Their approach is automatic, reducing the complexity of SPA detection for developers.

6.3.3 Machine Learning Methods

Liu [77] trains a support vector machine and employs dynamic analysis to detect the More is Less SPA in MySQL databases using runtime data. System-level metrics, database metrics, and MySQL status variables are used to build the detection dataset. Peiris et al. [133] use non-intrusive techniques, such as performance metrics, in combination with classification models to detect the One Lane Bridge SPA. They also propose a method to detect the Excessive Dynamic Memory Allocation SPA by using dynamic analysis to build the application's call graph and then coupling it with k-means clustering to detect the SPA by identifying short-lived, high-frequency allocations from the execution trace.

In comparison to the proposed approach, [76] lacks root-cause analysis. The graph-based method proposed in [74] does not consider runtime data, and relies just on static analysis, which makes the detection of deployment SPAs impossible. Many previous approaches rely

on extensive data collection, resulting in high overhead, or use traditional methods that require complete recalculation in each detection cycle. Our methodology addresses these gaps by employing low-cost, selective tracing, enabling efficient SPA detection in real-time microservice applications. Additionally, our learning model can be trained offline and reused, allowing for rapid identification of SPAs.

6.4 The Proposed Approach

This research proposes a machine learning-based approach to detect SPAs in microservices, with an emphasis on reducing data collection costs. Figure 6.2 illustrates the high-level architecture of the proposed approach, to gather required data and train the model, to detect SPAs. It consists of two main components: 1. Offline training mode, and 2. Online detection mode. This architecture is designed to facilitate version comparisons over time and detect the emergence of SPAs. In this setup, training is conducted offline, while detection is enabled when a new version is deployed. This approach can be integrated seamlessly into CI/CD pipelines. Additionally, when significant changes are finalized and approved -after completing the SPA checks and other performance and functional evaluations; the model must be retrained to establish the new version of the model as the baseline.

The offline training component follows three key steps: First, Data Collection Setup: This includes setting up distributed tracing, and configuring system-call data collection; Second, preprocessing: This involves correlating and aggregating the collected trace data; Third, model Training: The processed data is used to label the features by statistical methods, and then train a multi-class classifier that assigns probabilities to the likelihood of a SPA being present in service and operation level.

In the online detection component, the primary function is to use the trained model to detect SPAs. At this phase, new performance data are collected and tested against the built model to detect services with a high probability of exhibiting SPAs.

To guide our study, we define the following research questions, which are addressed through the methodology in Sections 4.1–4.5 and answered based on the experimental results in Section 6.5 and the discussion in Section 6.6:

- RQ1: Is it possible to detect SPAs in microservices by combining system-call data with observability traces?
- RQ2: Can machine learning methods be used to detect the introduction of SPAs following software or configuration changes (e.g., updates, deployment modifications)?

- RQ3: Can unsupervised learning methods detect SPAs with sufficient accuracy to be practical?
- RQ4: How do supervised, unsupervised, and semi-supervised methods compare in terms of accuracy, precision, and recall for SPA detection?

To answer these questions, first, in section 6.4.1 we present our hybrid tracing approach, to collect data used to build a SPA detection model. Second, in section 6.4.2, the steps required to preprocess the data and extract metrics required for the learning models are presented. Then, in section 6.4.4, we study different supervised, unsupervised, and semi-supervised machine learning methods to detect SPAs. These two sections provide the required setup for testing our hypothesis on the possibility of using an unsupervised machine learning method to detect SPAs across software changes. Altogether, these sections enable us to answer the above research questions.

6.4.1 Hybrid Tracing

Considering that our study focuses on microservices, we need to have a view of the request behavior across distributed nodes. We benefited from the [113] observability framework, which provides distributed tracing. On the other hand, it is important to study the internal behavior of the application to detect specific cases of SPAs. To this aim, we integrated kernel-level tracing within a distributed tracing framework. We used this setup to gather system-call traces. Figure 6.3 presents the data collection process in detail. As shown, the data used in our approach comprises two types:

- User-level distributed traces: One significant challenge in tracking performance issues within microservices is their distributed nature, which complicates the tracing of user requests. Distributed tracing tools like OpenTelemetry [113] and OpenTracing [114] are commonly employed in these applications to follow user requests across services by propagating a tracing context through nested sub-requests. Since this practice is widespread in microservices and often implemented in production environments, we leveraged the information from these traces to aid in root-cause analysis of service performance anomalies SPAs. To accomplish this, we used liblttng-ust, the LTTng user-space tracer, to instrument the Jaeger client [134], which serves as a tracing client for OpenTracing [114]. This setup allows us to intercept span-start and span-end events, marking the initiation and conclusion of OpenTracing spans, and gather essential data for our analysis.

- system-calls from kernel-level execution tracing: While user-level tracing is beneficial for identifying root causes, it lacks details on system-level metrics such as CPU usage, the internal call count required between operations, or the amount of data transferred in each operation. To address this, we also collected system-call traces. We used the LTTng modules tracer [12] to capture system-call traces at the kernel-level. Since our focus in this work is on communication-related SPAs, we collected only networking-related system-calls, specifically *recvfrom, recvmsg, recvmmsg, sendto, sendmsg,* and *sendmmsg.*

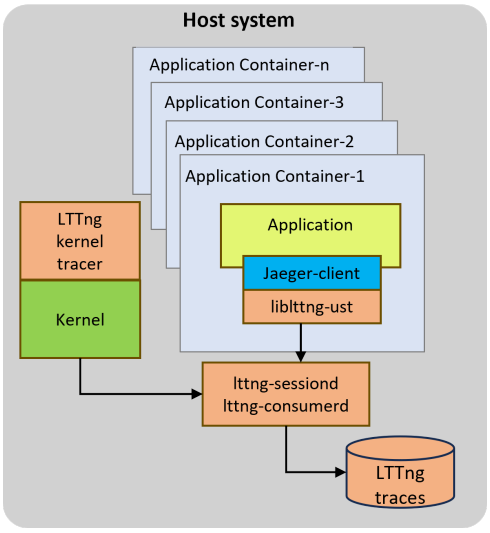

Figure 6.3 Data collection setup- Jaeger-client instrumentation by liblttng-ust and kernel tracing at the same time.

As previously mentioned, we used the LTTng tracer [12] to collect system-call traces, as it provides low-overhead tracing capabilities. The traces collected by LTTng can be viewed using the Trace Compass trace viewer [85], which aided in our method design by offering valuable statistics and trace analysis. Additionally, the Babeltrace Python library [135] is used to directly read the traces for further analysis. Gelle et al. [115] present a method for correlating distributed tracing with LTTng execution traces. Following this approach, we instrumented the OpenTracing distributed tracing [114] with LTTng tracepoints. This ensures that each time a distributed trace is generated, a corresponding LTTng trace is created, containing request information. Figure 6.4 shows the instrumentation code added to OpenTracing. As depicted, parameters such as *traceID, spanID, parentID, operationName, serviceName, and startTimeSystemInt* are passed to the tracer. To avoid direct code manipulation, we applied a patch to the source code for instrumentation, allowing us to aggregate and correlate all collected data for subsequent analysis.

```
tracepoint(jaeger_ust, start_span, ctx.traceID().high(),
 ctx.traceID().low(), ctx.spanID(), ctx.parentID(), operationNameCStr,
serviceName().c_str(), startTimeSystemInt);

 tracepoint(jaeger_ust, end_span, _context.traceID().high(),
     _context.traceID().low(), _context.spanID(), duration);
```

Figure 6.4 Opentracing Start-Span and End-Span instrumentation with LTTng UST.

6.4.2 Preprocessing and Trace Correlation

We aim to correlate each request with its related operation call and system-calls. To achieve this, the system-calls and user-space events data are merged into a unified dataset, capturing essential features for SPA detection. The following details outline the steps to correlate the data at the request level:

Extract system-call events and user-space events: A trace T is defined as a sequence of events with the same `trace_id`:

$$T = \{e_1, e_2, \ldots, e_n\}$$

where each e_i represents an event in the trace. The trace's start and end times are defined as:

$$\text{start\_ts} = \min(\{\text{cur\_ts}(e_1), \text{cur\_ts}(e_2), \ldots\}),$$

$$\text{end\_ts} = \max(\{\text{cur\_ts}(e_1), \text{cur\_ts}(e_n)\})$$

First, the Kernel call stack is built by identifying system-calls with both entry and exit events (e.g., `syscall_entry_sendto` and `syscall_exit_sendto`). Then the same process is repeated to build each request call stack. The response time for each event is computed as the difference between the entry and exit timestamps.

For each system-call, the response_time is computed as:

$$\text{response\_time} = \text{cur\_ts}_{\text{exit}} - \text{cur\_ts}_{\text{entry}}$$

Data set correlation: We merged the system-call and user-space events to build the complete request execution path. The events are merged based on timestamps and thread IDs (`vtid`). A system-call event T_s occurring at timestamp cur_ts_s is part of a user-space trace T_u if the timestamp of the kernel event falls within the start and end times of the user-space

trace:

$$\text{start\_ts}_u \leq \text{cur\_ts}_s \leq \text{end\_ts}_u$$

Additionally, the thread ID of the kernel event must match the thread ID of a user-space event in the trace:

$$\text{vtid}_s = \text{vtid}_u$$

If both conditions are satisfied, the system-call event inherits the `trace_id` and `service_name` from the user-space trace. Table 6.1 shows a sample aggregated trace.

Handling Missing Data in Kernel Events: Initially, kernel events do not have the `trace_id` and `service_name` fields, which are required in the final dataset. During the merging process, kernel events that match a user-space trace by timestamp and thread ID are assigned the corresponding `trace_id` and `service_name`. Kernel events that do not match any user-space trace are filtered out. The parameter *ret* is one of the parameters passed in system-call events. If the value of this parameter is positive, it indicates the message size; otherwise, it shows there is an error in the network system-call. We considered this value 0 for the user-space traces, which do not specifically indicate network activity. In addition, the parameter *op_name* indicates the name of the operation at the user level or the system-call name at the kernel-level.

Dataset aggregation: The final dataset is created by concatenating the processed system-call and user-space event data:

$$D = D_u \cup D_s$$

where D_u is the user-space dataset, D_s is the kernel dataset, and each record in D contains the required fields: `name`, `cur_ts`, `ret`, `response_time`, `span_id`, `parent_span_id`, `op_name`, `service_name`, and `trace_id`.

The output resulting from this step is used to perform analysis and identify SPAs.

Event encoding and padding: The event names are encoded into numerical representations using `LabelEncoder`. The sequences are padded to ensure all traces have the same length.

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ be the set of trace sequences, where each S_i is a sequence of operations $\{o_1, o_2, \ldots, o_{L_i}\}$.

Table 6.1 Sample of aggregated trace combining user-space and system call events.

name	current ts	return value	span id	parent span id	response time	operation name	service name	trace id
jaeger_ust:start_span	1726464134589066276	0	1709015	4706249999899046019	17642707676433399526	ReadUserTimeline	user timeline service	9616759882794086189
jaeger_ust:start_span	1726464134589081297	0	24775	17102146431855677904	4706249999899046019	RedisFind	user timeline service	9616759882794086189
jaeger_ust:start_span	1726464134589469928	0	16601	1624531274915946197	4706249999899046019	MongoFindUserTimeline	user timeline service	9616759882794086189
syscall_sendmsg	1726464134589559979	195	54444	Null	Null	syscall_sendmsg	user timeline service	9616759882794086189
syscall_recvfrom	1726464134589633523	0	15908	Null	Null	syscall_recvfrom	user timeline service	9616759882794086189
syscall_recvfrom	1726464134590166434	116	19280	Null	Null	syscall_recvfrom	user timeline service	9616759882794086189
syscall_recvfrom	1726464134590390202	4	33007	Null	Null	syscall_recvfrom	user timeline service	9616759882794086189
syscall_recvfrom	1726464134590456198	123	27597	Null	Null	syscall_recvfrom	user timeline service	9616759882794086189
jaeger_ust:start_span	1726464134590526296	0	19078	14929287647722053771	4706249999899046019	ReadPosts	post storage service	9616759882794086189

Table 6.2 Description of the unified request-based data extracted from the original trace data.

#	Data	Description
1	Time stamp	The starting time stamp of that event
2	Trace ID	Each request has a trace ID, which is the same across all events related to that request
3	Service Name	Name of the service in the microservice application
4	Operation Name	Name of the operation executed within that service in the microservice application
5	Span duration (ms)	A span in OpenTelemetry represents a single unit of work or operation within a trace. This field shows the time spent to execute the intended span.
6	Received Data (bytes)	Data received within the intended span, calculated by considering the amount of received bytes recorded by recv() and recvfrom() system-calls.
7	Transmitted Data (bytes)	Data sent within the intended span, calculated by considering the amount of sent bytes recorded by send() and sendto() system-calls.
8	Receive syscall Count	Data received within the intended span, calculated by considering the number of recv() and recvfrom() system-calls within that span.
9	Send syscall Count	Data sent within the intended span, calculated by considering the number of send() and sendto() system-calls within that span.
10	Total Operations	Total number of system-calls executed within the intended span.
11	Avg. Size Recv Msg	Average data received within the intended span, calculated by considering the average size of the received in bytes recorded by recv() and recvfrom() system-calls.
12	Avg. Size Sent Msg	Average data sent within the intended span, calculated by considering the average size of the sent in bytes recorded by send() and sendto() system-calls.
13	Ratio Recv time	Measures the throughput of receiving data in that span by dividing the Receive syscall Count by Span duration.
14	Ratio Sent time	Measures the throughput of sending data in that span by dividing the Send syscall Count by Span duration.
15	Service Contribution (%)	Measures time spent in this service in the intended request. So, if there are n services in the request, service contribution is the sum of the Span duration for that service, divided by n.
16	Error Count	Measures the number of errors recorded for the intended span. The metric amount is retrieved from "ret" parameter value in system-calls.

Table 6.3 Metrics extracted from each trace for SPA detection, aligned with typical SPA patterns

Metric	**Definition**	**SPA Type**	**Description**
Span Duration ($d_i)	$t_end - t_start$	Empty-semi-trucks, Blob	Time spent to execute the span.
Total Operations (TO)	N	Blob	Number of syscalls in a trace.
Total Response Time (TRT)	$\sum d_i$	Empty-semi-trucks, Blob	Sum of response times per trace.
Max Response Time (MaxRT)	$\max d_i$	Empty-semi-trucks	Longest operation duration.
Mean Response Time (MeanRT)	$\frac{1}{N}\sum d_i$	Empty-semi-trucks	Average duration per operation.
Received Data (b_{recv})	$\sum r_i \cdot \mathbb{I}_{\text{recv}}(o_i)$	Blob	Total data received.
Sent Data (b_{send})	$\sum r_i \cdot \mathbb{I}_{\text{send}}(o_i)$	Blob	Total data sent.
Avg. Size Recv Msg (a_{recv})	$\frac{b_{\text{recv}}}{c_{\text{recv}}+\epsilon}$	Blob	Mean recv message size.
Avg. Size Send Msg (a_{send})	$\frac{b_{\text{send}}}{c_{\text{send}}+\epsilon}$	Blob	Mean send message size.
Ratio Recv Time	$\frac{\sum d_i \cdot \mathbb{I}_{\text{recv}}(o_i)}{\sum d_i+\epsilon}$	Empty-semi-trucks	Ratio of recv time in span.
Ratio Send Time	$\frac{\sum d_i \cdot \mathbb{I}_{\text{send}}(o_i)}{\sum d_i+\epsilon}$	Empty-semi-trucks	Ratio of send time in span.
Transfer Ratio (r_{data})	$\frac{b_{\text{recv}}}{b_{\text{send}}+\epsilon}$	Blob	Data receive/send ratio.
Total Return Values (TRV)	$\sum r_i$	Blob	Sum of syscall return sizes.
Max Return Value (MaxRV)	$\max r_i$	Blob	Largest syscall return value.
Mean Return Value (MeanRV)	$\frac{1}{N}\sum r_i$	Blob	Mean syscall return value.
Error Count	Count of errors	All	Errors recorded in a span.
Service Contribution	Percent of total span	Blob,Empty-semi-trucks	Service share in total span.
Betweenness Centrality	Shortest-path freq.	Blob	Bottleneck indicator.
Closeness Centrality	Inverse mean path len.	Empty-semi-trucks	Delay propagation measure.
Operation Count (c_o)	$\sum \mathbb{I}(o_i = o)$	All	Count of each unique op.

1. **Encoding:** Define a mapping function f : Operations $\rightarrow \mathbb{N}$ that assigns each unique operation to a unique integer label. Each sequence S_i is transformed into an encoded sequence $E_i = \{f(o_1), f(o_2), \ldots, f(o_{L_i})\}$.

2. **Padding:** Pad sequences to a maximum length L (default is 100). For sequences shorter than L, zeros are appended:

$$P_i = \begin{cases} [E_i, 0, 0, \ldots, 0] & \text{if } L_i < L \\ E_i^{(1:L)} & \text{if } L_i \geq L \end{cases}$$

6.4.3 Feature Extraction and Metric Formulation

We used the unified data to extract metrics important for SPA detection. Table 6.2 describes the processed data extracted from the execution traces. Then, considering the calculated metrics, we extract features that best represent symptoms of each SPA and use them to train a machine learning model. These trained models are used in the detection phase to uncover SPAs in new trace data. Table 6.3 summarizes the extracted features, with corresponding mathematical definitions, their alignment to SPA detection, and a brief description.

These metrics provide data on service and operation level statistics, and also system-calls executed within each operation.

6.4.4 Machine Learning for SPA Detection

Research questions RQ2, RQ3, and RQ4 are all about what kind of machine learning approaches can be used in detecting SPAs. RQ3 checks whether SPAs (SPAs) can be accurately detected using unsupervised machine learning. While unsupervised methods are attractive due to their independence from labeled data, production real-world systems often exhibit SPAs even in early deployment stages, complicating the establishment of a reliable ground truth.

To address this challenge, we introduce a **hybrid detection pipeline** that begins with weak labeling and progresses through semi-supervised refinement, with optional supervised training. This approach leverages the scalability of unsupervised learning while improving precision through iterative label refinement. Semi-supervised models strike a practical balance, minimizing annotation effort while achieving high accuracy by integrating newly detected SPAs into subsequent training cycles.

We applied clustering algorithms such as **DBSCAN** and **KMeans** on targeted feature sub-

sets specific to SPA types:

- **Blob detection** relied on service-level features (e.g., service contribution, total span duration).
- **EST detection** utilized operation-level metrics, syscall intensities, and data transfer ratios.

Weak Labeling and Hybrid Refinement

Weak supervision has emerged as a scalable alternative to manual annotation [136–138]. Unlike prior work that relied heavily on expert labels or rigid metric thresholds, our method combines heuristic labeling with unsupervised clustering and rule-based refinement.

Initially, weak labels were generated using percentile-based statistical thresholds (e.g., 75th/90th percentiles) applied to key metrics. These approximations provided a foundation for clustering and refinement. Principal Component Analysis (PCA) supported visual inspection of clusters.

Next, we refined the labels using a hybrid ensemble of KMeans clustering (for identifying latent patterns) and decision trees (for rule-based correction). Clusters were mapped to SPA categories via majority voting. This iterative refinement helped resolve inconsistencies and improve label quality without requiring manual annotations. Visual tools such as box plots and histograms assisted in threshold tuning and outlier detection.

The refined labels were then used to train classifiers. This method is both flexible and adaptive, capable of detecting previously unseen SPAs while maintaining high labeling accuracy. The complete pipeline—from raw trace extraction to final classification—is shown in Figure 6.5.

Comparative Evaluation of Learning Paradigms

To validate the effectiveness of our hybrid pipeline, we implemented and compared **unsupervised**, **semi-supervised**, and **supervised** learning approaches on a benchmark dataset with known SPA labels. Table 6.4 summarizes the techniques used, along with their strengths and limitations.

- **Supervised learning:** Using refined labels, we trained classifiers such as *Random Forest* and *XGBoost*, which achieved high accuracy in distinguishing among Blob and Empty-semi-trucks SPAs.

- **Semi-supervised learning:** We employed *Label Propagation* on partially labeled datasets derived from weak labels. This improved accuracy on initially unlabeled instances, offering a good compromise between labeling effort and performance.
- **Unsupervised learning:** *KMeans* and *DBSCAN* formed the baseline. SPA labels were inferred post hoc based on cluster proximity and feature distributions. While less accurate, this approach required no prior labeling.

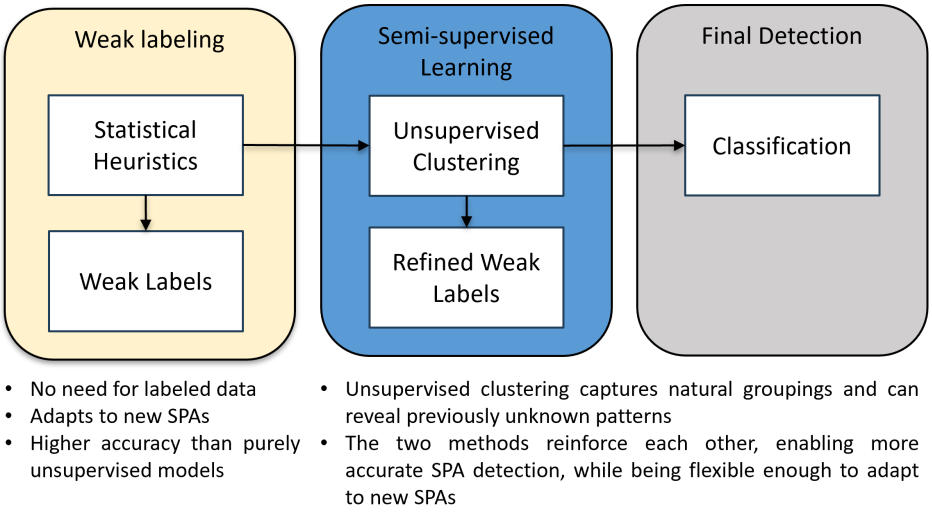

Figure 6.5 SPA detection pipeline: combining weak labeling, clustering, refinement, and classification.

6.4.5 Root Cause Analysis for SPA-Affected Traces

Causal analysis in distributed systems has received increasing attention due to the opaque and asynchronous nature of microservice communications. Building on prior work such as Gellen et al. [115], we correlate kernel- and user-space traces and extend this by incorporating Granger causality and PageRank-based centrality scoring to enhance root cause localization. This combination of statistical inference, graph theory, and machine learning–based labeling represents a novel contribution to the SPA detection literature.

Following SPA identification, our root cause analysis (RCA) framework applies causal reasoning and structural centrality to rank the most influential services contributing to trace anomalies. The RCA process is restricted to SPA-affected traces to limit computational overhead and ensure focused diagnostics.

To capture directional influence between services, we apply the Granger causality test to service-level time series aggregated by trace buckets. Let X_t^u and X_t^v denote the time series

Table 6.4 Comparison of Learning Methods for SPA Detection

Learning method	**Supervised**	**Semi-Supervised**	**Unsupervised**
Techniques	Random Forest, XGBoost	Label Propagation, Tree-based Refinement	KMeans, DBSCAN
Limitations	Requires labeled data from known SPAs or injections	Sensitive to label noise, depends on initial weak labels	Hard to evaluate, labels must be inferred post hoc
Benefits	High accuracy, interpretable, strong generalization	Improves with limited labeled data, balances flexibility and accuracy	No need for labels, good for discovering new patterns

of span durations for services u and v. The Granger test checks whether past values of X^u improve the prediction of X^v:

$$X_t^v = \sum_{i=1}^{p} a_i X_{t-i}^v + \sum_{j=1}^{p} b_j X_{t-j}^u + \epsilon_t$$

Here, the null hypothesis $H_0 : b_j = 0$ is tested using an F-test. A causal link is retained if the p-value < 0.05, and the strength of causality is computed as $1 - p$-value. In our implementation, dynamic time buckets are constructed to ensure statistical stability, and pairwise Granger tests are performed across all service pairs.

To contextualize causality, we construct per-trace dependency graphs and compute graph-theoretic centralities. Specifically, we use betweenness centrality to identify intermediate services acting as bottlenecks, degree centrality to detect communication-heavy services, and closeness centrality to identify services with poor network connectivity. These metrics are computed at the trace level to capture context-specific anomalies rather than relying on static service graphs.

We then combine the Granger causality results with graph centralities to construct a weighted directed graph $G = (V, E)$, where V represents services and E the causal influences between them. The weight of each edge is computed as:

$$\text{Weight}(u \to v) = \alpha \cdot \text{Granger}_{u,v} + \beta \cdot \text{Betweenness}(u)$$

with $\alpha = 0.7$ and $\beta = 0.3$ favoring statistical causality. PageRank is then used to compute a root cause score $R(v)$ for each node:

$$R(v) = \sum_{u \in \text{Pre}(v)} \frac{R(u)}{\text{OutDegree}(u)} \cdot \text{Weight}(u \to v)$$

Services with the highest $R(v)$ are prioritized as root cause candidates. Scores are normalized and visualized using a color-coded causal graph.

To illustrate this approach, consider a trace involving five services: *A, B, C, D, E*. The observed anomaly affects Service E, but its origin is unclear. After performing pairwise Granger causality tests, we obtain the following statistically significant scores:

- $\text{Granger}_{C \to E} = 0.93$
- $\text{Granger}_{B \to C} = 0.85$
- $\text{Granger}_{A \to B} = 0.10$ (not statistically significant; $p > 0.05$)

From previously computed centrality values:

- $\text{Betweenness}(B) = 0.35$
- $\text{Betweenness}(C) = 0.45$

We compute the edge weights as follows:

$$\text{Weight}(C \to E) = 0.7 \cdot 0.93 + 0.3 \cdot 0.45 = 0.786$$

$$\text{Weight}(B \to C) = 0.7 \cdot 0.85 + 0.3 \cdot 0.35 = 0.700$$

These weights are then used in PageRank calculations. Despite the anomaly appearing in Service E, the algorithm identifies Service B as the most influential root cause due to its strong upstream effect. Manual log inspection confirms that a misconfiguration in B led to cascading delays.

Figure 6.6 illustrates the output of the RCA system. Nodes represent services, colored by root cause score. Directed edges indicate Granger-based influence, annotated with corresponding weights. This figure illustrates RCA on our baseline traces. As shown, even without SPA injection, the Compose-Post-Serive seems to be a strong candidate for Blob.

The RCA module is integrated into the SPA detection pipeline and is executed only after a trace is labeled as anomalous. Each SPA is linked to its originating service and operation

via the `service_name` and `op_name` fields. RCA then ranks and filters these using root cause scores to highlight the most critical contributors.

Each trace is first labeled as *Normal*, *Blob*, or *Empty-semi-trucks* using a hybrid pipeline combining weak heuristics with clustering and supervised models. RCA is applied only to anomalous traces, ensuring computational efficiency.

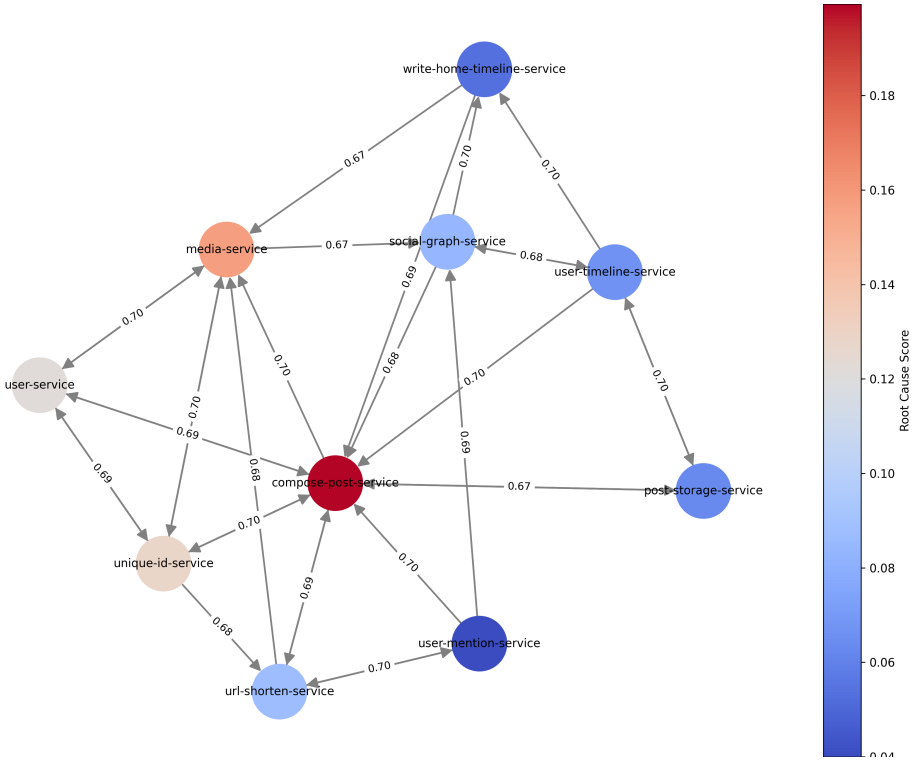

Figure 6.6 Color-coded causal graph displaying root cause scores and directional service influence.

By decoupling model training from detection and RCA, our method supports scalable and continual monitoring using pre-trained classifiers and efficient trace-level graph computations. In our evaluation, over 80% of top-ranked RCA results matched manually confirmed bottlenecks, demonstrating the effectiveness and robustness of our approach in real-world microservice environments.

Our approach contributes to the state of the art by automatically deriving baseline performance requirements in a baseline run and assessing pass/fail criteria for the load tests, using a baseline computation of these requirements. By separating the data analysis and model training phase from the detection phase, we establish a clear workflow where the model is first trained and validated using known data and then used to detect SPAs in new, unseen trace

data. This separation ensures that the performance of the model is evaluated independently of the detection results, promoting robustness and generalizability.

To detect and analyze SPAs, we designed a comprehensive machine learning pipeline by studying the benefits of using unsupervised, semi-supervised, and supervised models against each other. Each approach is designed to address one or more of our research questions (RQ2–RQ4), focusing on their capacity to generalize across service changes and detect SPAs in real-world deployments. To study how performance evolves across software updates and configuration changes, we use time-bucketed feature extraction and causality-based centrality measures. We compute Granger causality between service-level performance metrics across time to capture evolving dependencies. These dynamic causality graphs are then used to compute root cause scores and detect newly emerging SPAs. This approach is evaluated by comparing detected SPA types before and after code/configuration changes using historical trace datasets.

6.5 Experiments and Results

This section presents the experiments conducted to assess the proposed approach, demonstrating the practicability of the method by providing specific instances. Initially, the experimental setup, including testbed details, SPA injection, and tracing setup, is provided. Then, different scenarios are presented to test and evaluate the method.

6.5.1 Experiment Setup

We used DeathStarBench [120], an open-source benchmark suite for microservices, as our test application for these evaluations. DeathStarBench is widely recognized and frequently cited in research on microservice applications, offering a sufficiently complex structure to emulate real-world scenarios. We deployed the application containers on an Intel Core i7-6700K machine with 64GB of DDR4 RAM, running Ubuntu 18.04.6 LTS Bionic. Figure 6.7, presented in [120], shows the architecture of the DeathStarBench application, which will be referenced throughout our experiments when referring to different services within the application.

In our experiments, we utilized the Wrk2 load generator [121] for imposing a load on the application. Wrk2 is an HTTP benchmarking tool known for its ability to create a substantial load on web servers. Furthermore, this tool offers flexibility in specifying various parameters, such as throughput (requests per second), the number of threads, and more, which are essential for tailoring load tests to our specific requirements. Moreover, we used the LTTng

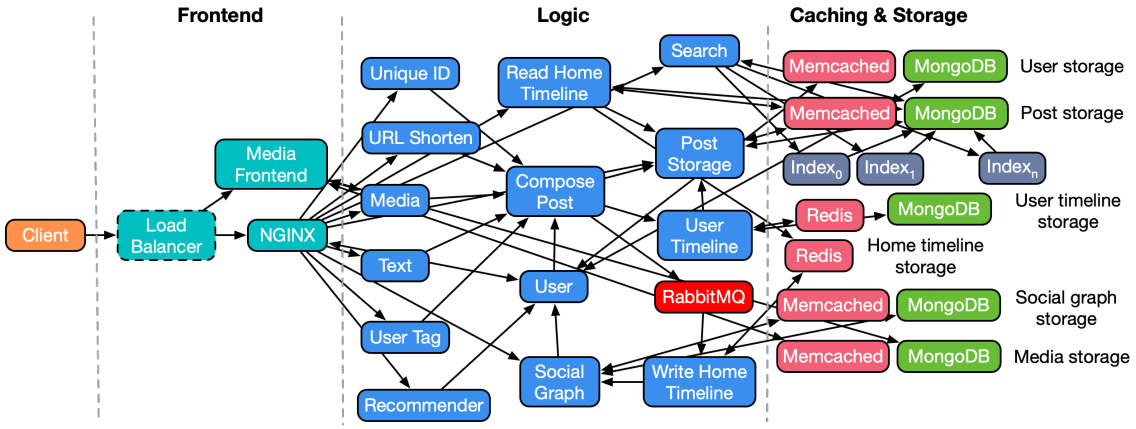

Figure 6.7 The DeathStarBench social network application architecture. This architecture features unidirectional relationships implemented with microservices that communicate with each other via Thrift RPCs.

tracer [12] to collect user-level and kernel-level traces. Figure 6.8 shows our trace collection script, which includes both user-level and kernel-level tracing at the same time. In addition, sample trace data generated from UST tracing and system-call tracing is provided.

SPA Injection

One important limitation in SPA detection studies is the difficulty of finding real-world SPA use cases. One popular approach to overcome this is SPA injection into the test application. To be able to accurately evaluate the performance of a SPA detection model, there needs to be clearly labeled data. We deliberately inject faults into the application to assess the effectiveness of our method. Although the application may naturally exhibit some SPAs, evaluating the effectiveness of the method becomes difficult without certainty about these SPAs in the absence of injected faults.

As mentioned before, we are aiming to detect Blob and Empty-semi-trucks communication SPAs as our case study. As per [127] classification of SPA faults, Empty-semi-trucks is a local implementation problem. A local SPA only affects a single method. It is usually the result of a bad programming decision. Many methods may be impacted by this bad practice; however, the problem can be fixed by modifying just the single faulty method. Considering this, it is quite easy to inject Empty-semi-trucks within the application code by simply changing the implementation of one method. However, Blob is a macro-architecture problem. This kind of problems affects the overall system structure. They manifest in the design phase. It is impossible to inject them within their current application implementation because the responsible modules for them do not exist in a correctly designed system. Considering this, we

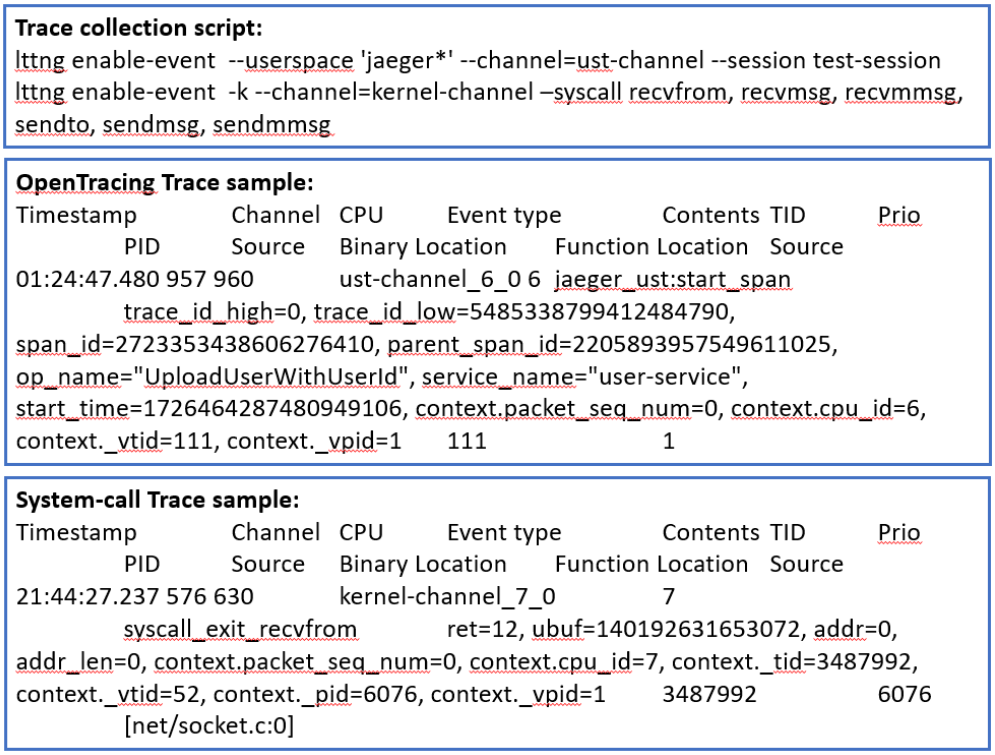

Figure 6.8 Sample user-space and system-call traces.

had to perform extensive changes to the application design to inject Blob, by moving all the logic from one service to another one, and thus unnecessarily enhancing the communication load on that changed service. Table 6.5 describes the SPAs we injected into the application.

There are various ways to implement each SPA. The Blob, for instance, can arise from overloading a microservice with too much logic or from its role as a data distributor. We focused on the former. In the DeathStarBench application, the UserTimeline microservice supports two primary operations: read and write. For the Blob-1 SPA injection, we moved the write operation logic to the PostStorage microservice. Thus, when the write operation is invoked, the UserTimeline service delegates the task to PostStorage. In the Blob-2 fault injection, we transferred both read and write logic to PostStorage. In Blob-2, in addition to read and write, we moved the logic of compostPost to the PostStorage service. In the Empty-semi-trucks-1 scenario, we modified the ComposeAndUpload commit, each call separately, to simulate overhead. For the Empty-semi-trucks-2 fault injection, instead of retrieving data in batches from MongoDB, we retrieved items individually, resulting in an excessive data load.

Our approach pinpoints SPAs by leveraging the collected tracing data to train a detection model for each SPA class. Therefore, the data collection steps need to be repeated for each

Table 6.5 Description of the SPAs injected into the test application.

SPA	Impacted services	Details
Blob-1	UserTimeline, PostStorage	Moved whole logic of writeUserTimeline operation to PostStorage service
Blob-2	UserTimeline, PostStorage	Moved whole logic of readUserTimeline operation to PostStorage service
Blob-3	UserTimeline, PostStorage	Moved whole logic of readUserTimeline and writeUserTimeline operation to PostStorage service
Blob-4	hometimeline, PostStorage, ComposePost	Moved whole logic of writeUserTimeline, and composePost operations to PostStorage service
Blob-5	hometimeline, PostStorage	Moved whole logic of hometimeline operations to PostStorage service
Blob-6	UserTimeline, PostStorage, hometimeline	Moved whole logic of readUserTimeline, writeUserTimeline, and hometimeline operations to PostStorage service
Empty-semi-trucks-1	ComposePost	In ComposeAndUpload commit each call separately to simulate overhead
Empty-semi-trucks-2	UserTimeline	In readUserTimeline instead of fetching all posts at once, the operation uses a for loop where each post is queried from MongoDB individually

SPA to have relevant data for further analysis. Hence, any step described here is performed for the baseline application to have a baseline for comparison, and also for each SPA, which are Blob and Empty-semi-trucks in this work.

System-calls vs. Kernel events

At the start of our experiments, we evaluated whether system-call traces or kernel-level events would be more suitable for SPA detection. Our goal was to select a source that (a) could be correlated with user-level traces, (b) provided accurate measures for data size, response time, and throughput, and (c) minimized tracing overhead.

We examined networking-related kernel events (e.g., net-if-receive-skb, net-dev-queue, sched-switch) and compared them to communication-related System-calls (recvfrom, sendto, sendmsg,

etc.) in extreme SPA-injected cases. In both data sources, SPA symptoms were visible ; for example, Blob cases showed significantly longer durations and higher message counts than baseline. However, we found three practical issues with using kernel events:

- Noise filtering complexity – kernel events contained more unrelated activity, increasing preprocessing effort.
- Trace size overhead – kernel events generated significantly larger traces (up to 315% more) because each system call could result in multiple kernel events.
- Missing size metrics – extracting accurate send/receive byte counts from kernel events was less direct than from System-calls.

Despite our choice, we observed some interesting patterns, like changes in sched-switch when there is a Blob, which we believe is worth studying in more detail later on.

6.5.2 Experiments and Test Cases

The evaluation environment setup is discussed in the previous section. In this section, we present the various scenarios tested to simulate SPAs, followed by a detailed analysis of the results from the collected data and the multi-classification model developed.

During the data collection phase, we implemented multiple scenarios to ensure sufficient data for building the learning model. Table 6.6 lists the experiments conducted during this phase. In addition to scenarios that generated data specific to each injected SPA, we included two key baseline scenarios: one capturing traces from the unchanged application and another with added system fluctuations as noise. We used stress-ng [139] to generate CPU, heap memory, interrupt, and page fault noise, several times during the data collection phase. The same noise profile was applied in all "with noise" scenarios to ensure experimental consistency. Data from the first scenario establishes baseline expectations for the application in our calculations. In contrast, the second scenario simulates conditions such as resource overload, allowing us to capture traces from the unchanged application in a more realistic setting. This approach supports validation of the ability of the method to detect SPAs, despite common application degradation. In addition, we applied the same strategy of noise generation for each of the other scenarios while studying the impact of the injected SPAs on the application. There, for each of the injected SPAs, we have one test case that does not include any noise, and another one that is captured while additional resource noise was applied to the application. Table 6.6 provides the list of the experiments conducted, and the SPA related to each test case.

Additionally, we conducted load testing by gradually increasing the load and maintaining it at each level for one-hour intervals, from 1 user up to 500 users, which is the maximum capacity of the application. This comprehensive testing ensures that the proposed method remains robust and effective under varying load conditions.

Table 6.6 List of experiments conducted to collect data related to all SPAs under study.

#	Scenario	Load
TC1	Baseline	no noise, 3600 seconds, increasing users gradually from 1 to 512, request starting from 100/sec to 1500/sec
TC2	Baseline resource noise	3600 seconds, resource noise, increasing users gradually from 1 to 512, request starting from 100/sec to 1500/sec
TC3	Blob-1 no noise	3600 seconds,1800 seconds warm-up with base load, fluctuating users between 200 to 512, requests 800/sec to 1500/sec
TC4	Blob-2 no noise	3600 seconds, 1800 seconds warm-up with base load, fluctuating users between 200 to 512, requests 800/sec to 1500/sec
TC5	Blob-3 no noise	3600 seconds, 1800 seconds warm-up with base load, fluctuating users between 200 to 512, requests 800/sec to 1500/sec
TC6	Blob-4 no noise	3600 seconds, 1800 seconds warm-up with base load, fluctuating users between 200 to 512, requests 800/sec to 1500/sec
TC7	Blob-5 no noise	3600 seconds, 1800 seconds warm-up with base load, fluctuating users between 200 to 512, requests 800/sec to 1500/sec
TC8	Blob-6 no noise	3600 seconds, 1800 seconds warm-up with base load, fluctuating users between 200 to 512, requests 800/sec to 1500/sec
T9	Blob-1 resource noise	3600 seconds, 1800 seconds warm-up with base load, fluctuating users between 200 to 512, requests 800/sec to 1500/sec
TC10	Blob-2 resource noise	3600 seconds, 1800 seconds warm-up with base load, fluctuating users between 200 to 512, requests 800/sec to 1500/sec
TC11	Blob-3 resource noise	3600 seconds, 1800 seconds warm-up with base load, fluctuating users between 200 to 512, requests 800/sec to 1500/sec
T12	Blob-4 resource noise	3600 seconds, 1800 seconds warm-up with base load, fluctuating users between 200 to 512, requests 800/sec to 1500/sec
TC13	Blob-5 resource noise	3600 seconds, 1800 seconds warm-up with base load, fluctuating users between 200 to 512, requests 800/sec to 1500/sec
TC14	Blob-6 resource noise	3600 seconds, 1800 seconds warm-up with base load, fluctuating users between 200 to 512, requests 800/sec to 1500/sec
TC15	Empty-semi-trucks-1	600 seconds, increasing users gradually from 1 to 10, request starting from 1/sec to 500/sec
TC16	Empty-semi-trucks-2	600 seconds, increasing users gradually from 1 to 10, request starting from 1/sec to 500/sec
TC17	Empty-semi-trucks-1 resource noise	600 seconds, increasing users gradually from 1 to 10, request starting from 1/sec to 500/sec
TC18	Empty-semi-trucks-2 resource noise	600 seconds, increasing users gradually from 1 to 10, request starting from 1/sec to 500/sec

Details on the experiments

To facilitate understanding the steps taken in our analysis, and provide more context on the experiments, details on the Blob-5 scenario are discussed in this section. First, as discussed previously, user-level traces, generated by the integration of OpenTelemetry with LTTng, are gathered. Figure 6.9 shows a sample trace from open-telemetry. This is a sample of Blob, where most of the logic in the hometimeline-service is moved into the post-storage-service. As shown in the figure, the execution time of this request is 4.99s, which is unusually longer than the average execution time of the application that typically takes under 1s.

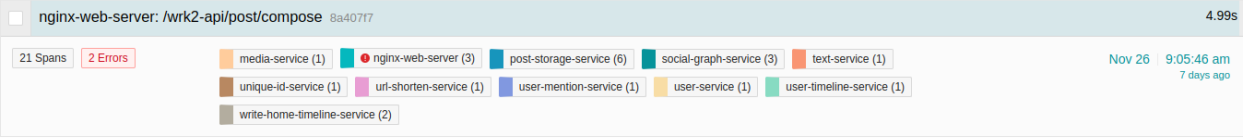

Figure 6.9 Blob-5 scenario user-level trace, viewed with Jaeger. In this scenario, read functionality from the hometimeline service is moved to the post-storage service.

Furthermore, the collected traces on both kernel and user levels are preprocessed to detect SPAs. A blob is caused by bad design or deployment decisions at the microservice or node level. Regarding this, we chose service level features like syscall-Count, Request-Intensity, Avg-Data-Sent, Avg-Data-Received, and Error-Rate to study this SPA. Using the normal baseline traces, we are able to compare the impact of the SPA on these features. Figure 6.10 provides the base features distribution plot for this scenario. In addition, Figure 6.11 provides the service calls dependency graph, and the number of service calls in a sample execution, captured by our baseline scenario tracing setup.

Finally, Figure 6.6 presents the results of the RCA analysis.

6.5.3 Results

The collected data in baseline scenarios are preprocessed and used to train the classification model, as detailed in the methodology section. This confirms RQ1 by demonstrating that combining system-call and distributed tracing data enables accurate SPA detection with minimal performance impact, as later quantified in our overhead analysis. Then, the data from a test case is utilized to validate the capability of the trained model to detect the intended SPA.

We encountered several challenges during the experiments. Initially, the detection accuracy appeared very high (99.99%), raising concerns about overfitting, which occurs when a model

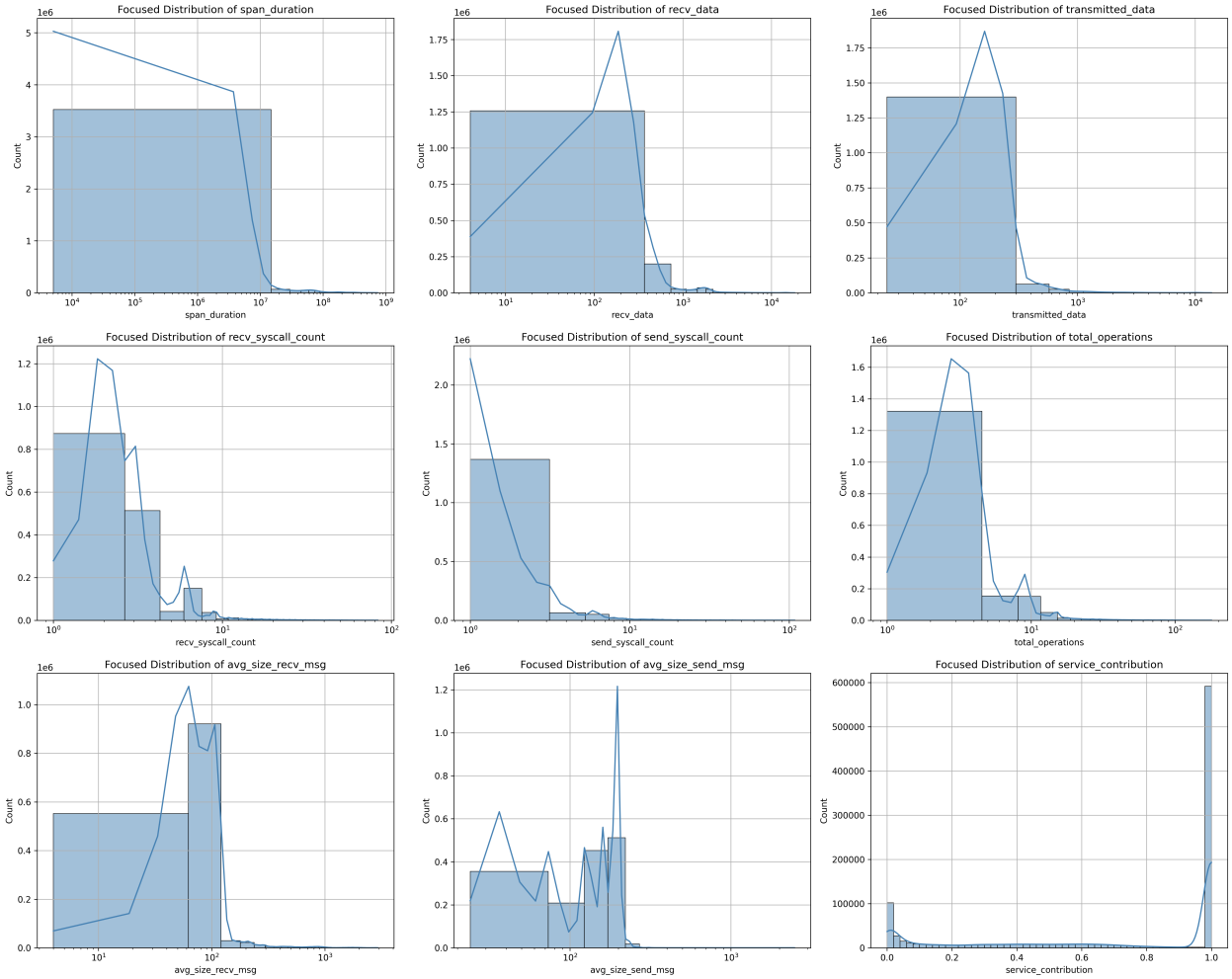

Figure 6.10 This figure illustrates the distribution of features selected to study the impact of the Blob in TC13.

becomes too closely fitted to the training data, resulting in poor performance on new data. To validate this, we employed several common practices, including altering the dataset, performing k-fold cross-validation, calculating information gain for all features, extracting the decision tree to better understand the decision-making process, and visualizing the most important features for deeper insight.

Through these techniques, we discovered that our initial SPA implementations during fault injection were too extreme, leading to significant differences in response times for each case. This oversimplified the SPA detection problem by allowing classification based solely on response times. To address this, we took three corrective measures. First, we included stress on various resources, while collecting baseline data to simulate a more realistic environment and make SPA detection more challenging. Second, we adjusted the SPA implementations

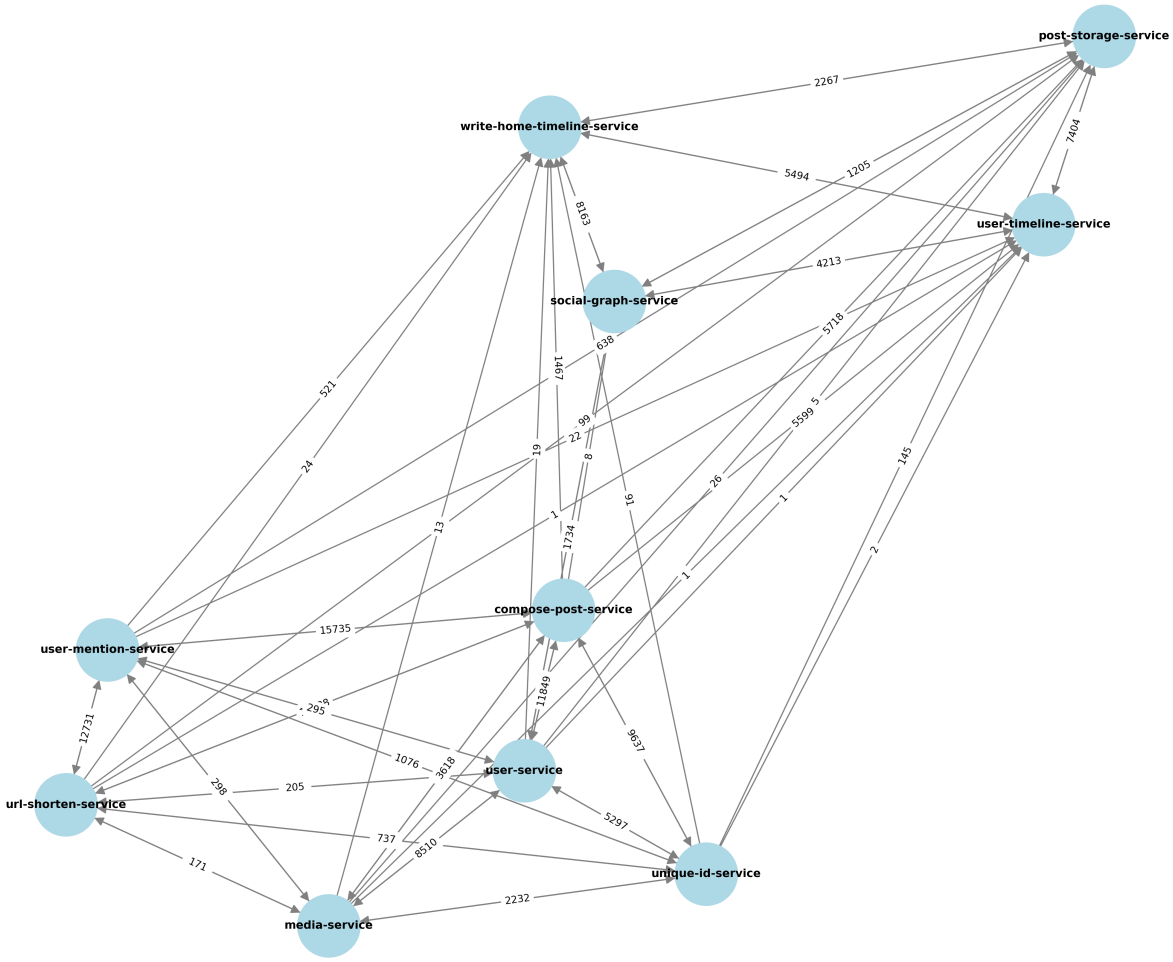

Figure 6.11 Service call network updated

to represent less extreme cases. Finally, we used multiple implementations for each SPA type, enabling the model to account for a broader range of impact on training features. This confirms RQ2 by showing that our offline–online workflow can detect newly introduced SPAs by comparing updated software versions to a baseline model, even when additional resource noise is present.

Performance Comparison

We tested different classifiers to identify the most effective one for our scenario. Table 6.7 provides comparisons of classifier performance for detecting Blob and Empty-semi-trucks SPAs. The table reports the Precision, Recall, and Accuracy of SPA detection. Precision

measures the accuracy of true and false positive predictions. Recall measures how well the model identifies all actual positive cases by considering the true positives and false negatives. Accuracy measures the overall correctness of the model by evaluating the proportion of correctly classified instances (both positive and negative) out of the total instances. High Recall indicates that a low false negative rate is crucial to avoid missing SPAs and reducing detection accuracy. High Precision indicates a low false positive rate. These comparative results directly answer RQ4, showing that supervised methods (e.g., XGBoost) achieve the highest accuracy, followed by semi-supervised approaches, with unsupervised methods performing the lowest across all metrics. Low Precision will result in the detection of false SPAs, leading to unnecessary efforts by the development team to address non-existent issues. As shown in the tables, our method achieves a high Accuracy. However, Precision and Recall can be improved. This addresses RQ3, indicating that while unsupervised approaches can detect SPA patterns without labeled data, their 63% accuracy is insufficient for reliable production use, making them better suited as supplementary rather than primary detection methods.

Table 6.7 Model Performance Comparison (Average over 5-fold Cross Validation)

Method	Accuracy	Precision	Recall
Unsupervised (minibatch-KMeans)	0.61	0.55	0.57
Unsupervised (KNN)	0.63	0.60	0.57
Unsupervised (gaussian-mixure)	0.63	0.60	0.59
Unsupervised (hierarchical clustering)	0.61	0.60	0.56
Semi-Supervised (KMeans)	0.78	0.76	0.74
Supervised (Random Forest)	0.89	0.88	0.86
Supervised (XGBoost)	0.91	0.90	0.89

To ensure fairness in comparison, all models were trained and evaluated on the same datasets. Metrics such as accuracy, precision, recall, and F1-score were computed using 5-fold cross-validation on a manually verified subset of traces. Additionally, PCA and decision tree visualizations were generated to interpret model behavior and highlight feature importance for each SPA class.

6.6 Discussion

Our primary focus in this work is on communication SPAs. While this method is generalizable to detect additional SPAs, we have limited our scope to Blob and Empty-semi-trucks SPAs, which are identifiable by similar system-calls. Notably, by selecting relevant system-calls, our method can be adapted to detect other SPAs as well. Additionally, our previous work addresses SPAs detectable exclusively through user-space traces [140].

The semi-supervised approach also allows us to strengthen the detection accuracy as the model is used more in context. Our best detection Accuracy for supervised learning is 91%. Furthermore, our detection Precision is 0.9% and Recall is 0.89%, which are good, but have room for improvement.

In summary, our proposed method is less intrusive than others, as it primarily relies on system-call traces. By leveraging OpenTracing instrumentation, already present in the application, our approach remains low-effort and practical. Access to user-space traces also enables us to identify the specific service and operation causing the SPA, providing developers with targeted insights for resolution. Additionally, by collecting baseline traces, we derive performance requirements tailored to the test application, creating domain-specific performance benchmarks that are more realistic than the common practice of using static thresholds.

To insure that the classification method used is appropriate for the intended problem, as mentioned in the experiments section, we classified the data with several classifiers and selected the one that has the best accuracy, precision, and specificity overall. In addition, we considered using methods like LSTM; however, their computation cost is much higher than methods like the XGBoost classification method, while not offering significantly better results.

One important challenge in this study is selecting the right kernel-level events for our analysis. We experimented with different types of data. Initially, we considered kernel-level events such as *net-if-receive-skb*, *Net-dev-queue*, and *sched-switch*, but ultimately chose communication-related system-calls. By studying kernel events related to networking and communication, we were able to observe distinct changes in event trends, coinciding with each SPA. These changes can be considered symptoms of SPAs. However, this data also contained a lot of noise, which is difficult to filter. In addition, since each system-call may translate into several kernel events, considering them generates more tracing and analysis overhead. On the other hand, system-call events provide more detailed information about the communication, such as the result and duration of each call. As a result, we decided to use system-calls instead of kernel events. Appendix I includes some screenshots of the trend changes for each SPA when kernel events are considered.

To assess the overhead introduced by our SPA detection method, we measured the application throughput across three scenarios: 1. Without LTTng tracing; 2. With LTTng tracing for OpenTracing span-start and span-end tracepoints and system-call traces; and 3. With LTTng tracing for OpenTracing span-start and span-end tracepoints along with full kernel tracing. All tests were conducted under a mixed load scenario with 500 threads, 500 connections,

and 2000 requests per second. To obtain realistic results, measurements were taken after a 10-minute warmup period. Figure 6.12 presents the outcomes of this experiment. The application uses a default OpenTracing setup, which we maintained throughout these tests. As shown, the overhead from our trace collection only adds approximately 2.74%, a minimal impact, particularly when compared to most existing methods. This low data collection overhead makes our method highly suitable for real-world environments.

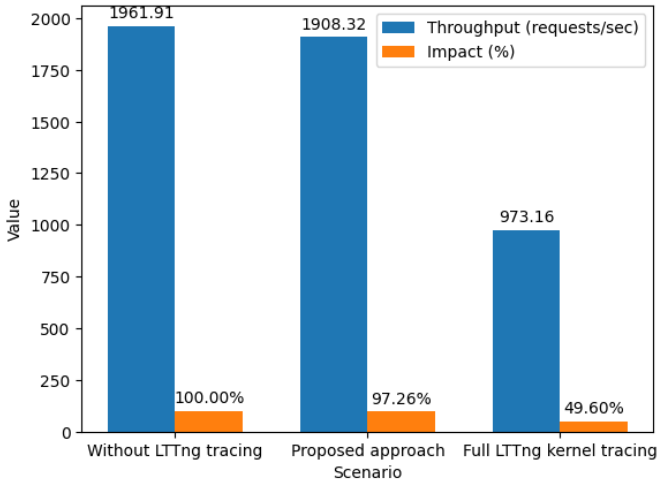

Figure 6.12 Data collection overhead analysis.

6.6.1 Limitations and Threats to Validity

An important limitation in testing our approach is the lack of access to real-world application data that contains reported SPAs. Most prior studies use test applications to evaluate their methods. Although it may be possible to identify SPAs by performing an in-depth search in GitHub commits, these cases are typically not labeled with specific SPA characteristics, making this task complex and a project in itself. Furthermore, when using real-world production data, there is no prior knowledge available to evaluate the detection accuracy.

Our approach is based on leveraging distributed tracing frameworks, such as OpenTracing and Open Telemetry [113], commonly used to instrument modern microservice applications. However, if these tracing tools are unavailable, implementing our method becomes challenging. While our experiments suggest that system-calls alone can sometimes suffice for SPA detection, they do not provide enough details for root-cause analysis. To address this challenge, tools like Envoy [141] could be used, as they report tracing information on inter-service communication within a service mesh. However, this approach is limited to communication SPAs and reduces the generalizability of our method. Our approach considers low and high

level traces, which makes coordinating the root-cause analysis in micro-services possible.

Further concerns regarding generalizability arise with SPAs that primarily affect the internal workings of a single method, such as certain implementations of the hiccup pattern, which may cause delays in specific methods. In these cases, it is more effective to rely on user-space tracing methods, as discussed in our previous work [140]. Additionally, replicating certain SPAs through problem injection may be nearly impossible. Studying the detection of these SPAs requires real-world cases with confirmed reports of their occurrence.

Finally, using system-calls related to communication limits our results to specific types of SPAs. To study other types, e.g., IO-related SPAs, we need to add system-calls that cover interactions related to IO. This shows both our approaches are generic, and also a limitation in terms of not being able to use the current implementation, without adding anything, to uncover other SPAs.

6.7 Conclusion and Future Work

SPAs are common practices that lead to scalability issues under high loads. Focusing on identifying SPAs, rather than general performance problems, offers the advantage that these issues are very common and well-documented in the literature, with clearly defined solutions. Given that microservices rely heavily on service communication, identifying communication-related SPAs is crucial.

As discussed in previous sections, most existing SPA detection methods require extensive data collection, which is impractical in industrial settings due to the emergence of SPAs under high loads. Our methodology combines advanced data processing techniques with machine learning algorithms, to detect SPAs in trace data efficiently. By selectively tracing system-calls, our approach significantly reduces data collection requirements for SPA detection. Additionally, building a learning-based model, from the current application version, allows for adaptability to future updates, enabling rapid SPA detection using the pre-trained model. Our evaluation shows that this approach is low-overhead (2.74%), highly accurate (91%), and can be automated for use within continuous development pipelines.

In this study, we validated our approach using the widely recognized DeathStarBench microservice testbed, which provides a realistic and reproducible environment for SPA detection experiments. This setup allowed us to establish the core approach in a controlled and well-understood setting. While public testbeds like DeathStarBench are valuable for initial validation, we acknowledge that industry-specific data and operational environments may present unique challenges and nuances. Due to practical constraints, we did not directly evaluate our

approach on industrial proprietary systems; however, future work will focus on adapting and testing this approach in collaboration with our industrial partners, using real-world data to further assess its robustness and applicability in high-performance telecommunications environments. This partnership will allow us to refine the method to address specific demands and performance conditions relevant to the industry.

A promising direction for future work is extending this method to include more SPAs from different categories. For instance, the Chatty I/O SPA [142] arises from inefficient input/output handling, and detecting it could be achieved by tracing system-calls related to input/output operations. Additionally, exploring advanced techniques such as topic modeling [143] and generative AI [144] could further enhance SPA detection. For example, our ongoing work uses topic modeling to streamline preprocessing steps significantly.

This study addressed four research questions on the feasibility and effectiveness of SPA detection in microservices. We showed that (RQ1) hybrid tracing of system-calls and distributed traces can detect SPAs with minimal overhead, (RQ2) our offline–online ML workflow can accurately identify newly introduced SPAs after software or configuration changes, (RQ3) unsupervised methods can detect SPA patterns without labels but with lower accuracy (63%), and (RQ4) supervised methods outperform semi- and unsupervised ones, with semi-supervised approaches offering a balanced alternative when labeled data is scarce. These findings confirm the practicality of our method for CI/CD integration and continuous performance monitoring in complex microservice environments.

CHAPTER 7 GENERAL DISCUSSION

This chapter revisits the main findings of the thesis, reflecting on their significance and implications in the context of performance analysis and tracing in distributed systems. It also outlines the key limitations encountered during the research and discusses their impact on the generalizability and applicability of the results. Finally, it highlights how the methods developed in this work can be generalized and extended beyond the initial use case of software performance anti-pattern (SPA) detection.

7.1 Key Contributions and Findings

The central objective of this thesis is to improve the observability of complex software systems through **adaptive, goal-driven tracing**. Across three research tracks, we proposed methods that enable tracing strategies to dynamically align with performance analysis goals, with a particular focus on the detection of SPAs.

In the **first track**, we demonstrated how unsupervised learning models could detect anomalies in traces using features derived from critical paths. The selection of critical path data instead of full or system call tracing strategy allowed for reduced tracing overhead while preserving diagnostic effectiveness. We validated the approach using both synthetic and publicly available datasets, showing that goal-aware trace collection improves both efficiency and detection accuracy.

In the **second track**, we tackled the challenge of SPA detection by creating a structured testbed where multiple variations of SPAs were injected and distributed across services. Our method modeled SPAs in terms of causes, symptoms, and metrics, forming a foundation for structured and reproducible evaluations. The controlled testbed enabled consistent benchmarking and paved the way for future comparative studies. To detect SPAs, several statistical detection algorithms, that only use user-space traces, are presented in this work.

In the **third track**, we explored tracing at the kernel level and found that system call–level instrumentation provides the right balance between detail and signal-to-noise ratio. By switching from low-level kernel events to system calls, we enhanced the trace quality, allowing for more accurate correlation with application-level requests. We also observed interesting trends in scheduler event patterns that point to possible future use in dynamic system profiling.

In all three tracks, our overarching framework for goal-based adaptive tracing, illustrated in Figure 7.1, served as a flexible foundation for trace strategy generation based on the diagnosis

objective.

7.2 Limitations of the Research

Despite the contributions, several limitations emerged throughout this research. A major limitation in the first track was the lack of publicly available, labeled datasets for validating our anomaly detection methods. Although we eventually leveraged a labeled dataset from the ICPE 2025 Data Challenge, it was relatively small and required augmentation. This affected the diversity of test scenarios and may have biased evaluation results.

In the second track, we found very few publicly documented examples of SPAs in large-scale production systems. Apart from known cases in Hadoop and Cassandra (YARN-4307, HDFS-12754, CASSANDRA-13794), no real-world labeled datasets were available. Moreover, the existing literature mostly relies on privately modified systems with injected SPAs that are not openly accessible. To address this gap, we manually injected various SPA types into our testbed and ensured comprehensive coverage. While this improves reproducibility, it limits the ecological validity of the findings.

In the third track, selecting appropriate kernel-level events posed practical challenges. Network-related events such as `net_if_receive_skb` and `net_dev_queue` offered useful indicators of network load but lacked critical contextual attributes (e.g., message size). Transitioning to system call–level tracing improved the usefulness of the collected data but may have overlooked some low-level scheduling behavior that could be relevant for future work.

7.3 Generality and Scope of Goal-Based Tracing

While our framework is designed to be extensible, this thesis primarily focused on SPA detection. However, the general architecture supports a broader range of observability goals. These can be defined vertically (e.g., across layers of the software stack) or horizontally (e.g., across trace domains such as user and kernel space). The ontology presented in Figure 7.1 illustrates how observability goals, workload patterns, and degradation types can be encoded to drive trace instrumentation dynamically. Future work is needed to test this generalization in practice.

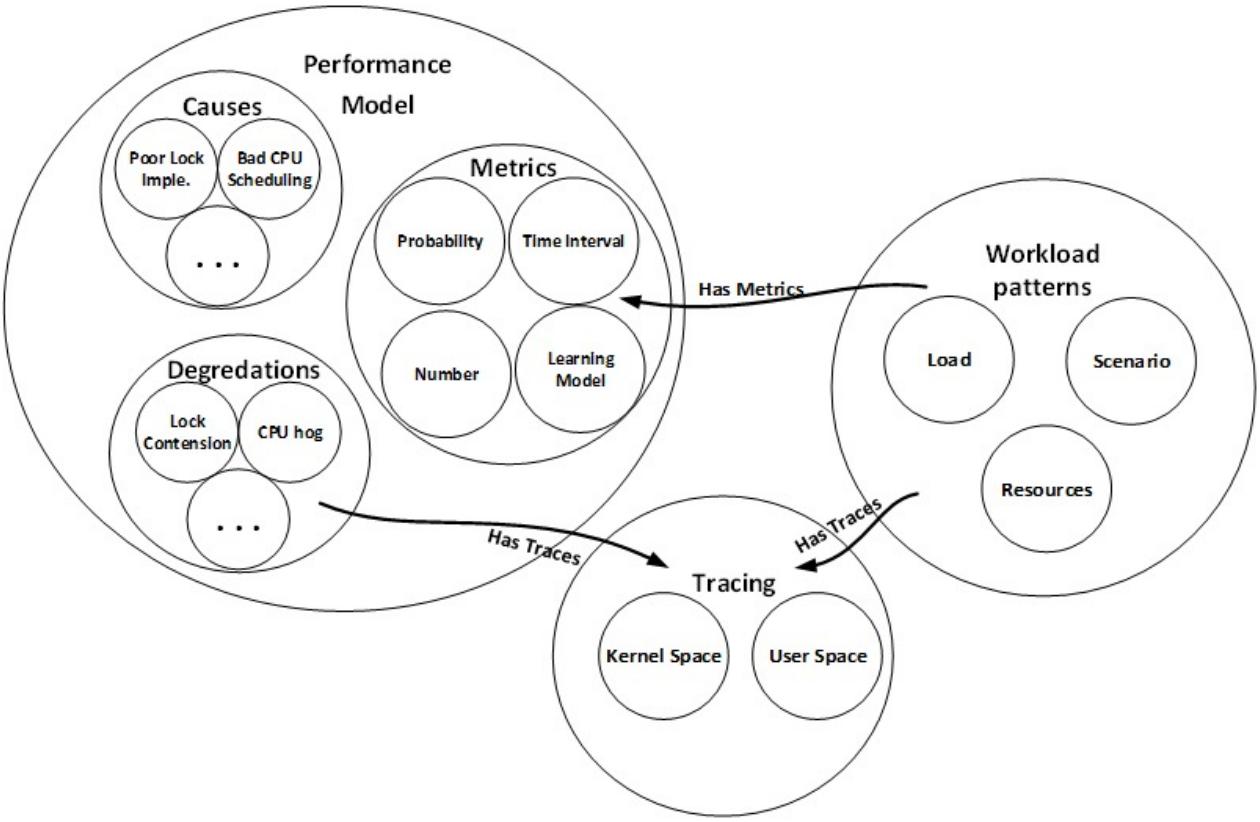

Figure 7.1 Adaptive tracing ontology, modeling tracing spaces, workload patterns, and performance models, required to build an adaptive tracing method.

The insights from this thesis extend beyond SPA detection. The proposed goal-based approach promotes a shift in how observability is engineered—moving from static trace instrumentation to dynamic, adaptive strategies guided by performance goals. This approach reduces the overhead and improves relevance, helping practitioners tailor tracing to specific problem contexts. For example, in scenarios involving intermittent degradations, targeted kernel or syscall tracing could offer more insight than broad-based user-level traces.

CHAPTER 8 CONCLUSION AND FUTURE WORK

Modern software systems handle billions of transactions every day. With such scale, maintaining performance is far from trivial. Tracing, logging, and monitoring tools have become essential for identifying slowdowns and regressions early. In this thesis, we focused on tracing, because it provides the most detailed insights into how software behaves at runtime and is especially useful for root cause analysis.

From working with both kernel-space and user-space tracers, I have seen firsthand how much unnecessary data gets collected, often far more than what is actually needed to diagnose a problem. This extra data leads to overhead in terms of storage, processing, and most importantly, human attention. The tools we have today are often not smart enough to focus only on what matters.

The goal of this thesis has been to rethink tracing: how can we make it smarter, lighter, and more targeted? Instead of collecting everything and hoping to find the issue later, what if we adaptively collect only what we need, when we need it?

To get there, we proposed a more goal-driven approach to tracing. The idea is simple: if we know what kind of problem we are trying to catch, for instance a performance regression or a resource bottleneck, then we should be able to configure the tracer to focus on areas that are likely to reveal that problem. This could mean enabling or disabling instrumentation points or changing sampling rates dynamically.

However, this assumes we first know that a performance issue has started to happen. Through this work, we identified four main reasons why software might start behaving differently:

1. **Workload changes:** either the type or intensity of the workload has changed.
2. **Application behavior changes:** a new code path is being executed, potentially revealing hidden issues.
3. **Configuration changes:** something in the environment, like memory size or timeout values, has been adjusted.
4. **Code changes:** a library has been updated or the source code has changed.

Any of these can trigger performance degradations. For example, think of an online store during a big sale, suddenly traffic surges and CPU usage spikes. Similarly, imagine a new feature that activates a previously unused function that turns out to be inefficient.

8.1 Track 1: Tracing Runtime Behavior and Workload Changes

In the first part of my research, we looked at how to detect changes in workload and behavior during execution, without relying on code or deployment updates. These runtime shifts are tricky to catch, especially in production where overhead must be kept low.

Our approach was to monitor the system execution flow and look for signs of deviation from what is "normal." For instance, similar requests should follow similar execution paths. When that pattern breaks, it can be a signal that something has gone wrong. Based on this signal, we can adjust the tracing configuration to collect just enough extra details to understand the issue, without overwhelming the system or the observer.

8.2 Track 2: Code-Level SPAs and Performance Changes

The second part of my work focused on performance problems that stem from changes in the code itself. This could be due to a developer introducing a known anti-pattern or using a poorly performing library function. The idea was to see whether we could automatically identify such issues by detecting symptoms and matching them to a catalog of known SPAs.

This track led to two key contributions:

- Automatically ranking the parts of the codebase most likely responsible for the slow-down.
- Reducing tracing overhead by focusing only on what is needed for the diagnosis.

8.3 Track 3: System-Level Tracing and Deployment Issues

The third part of the thesis dealt with execution-level performance problems that arise due to misconfigurations or poor deployment choices. These issues do not always stem from bad code, they can result for example from binding a service to the wrong CPU core or using inconsistent settings across machines.

Here, we used system-level traces and correlated them with user-level traces to pinpoint the source of these problems. By bridging the gap between high-level symptoms and low-level causes, this approach made it possible to catch issues that traditional tracing would miss or attribute incorrectly.

8.4 Future Work

For future work, several promising research directions can be pursued.

First, as discussed in Section 7, our observations indicate that certain kernel-level events exhibit distinct patterns of change during the occurrence of SPAs. A systematic analysis of these patterns, especially through techniques such as pattern mining, frequent pattern analysis [145, 146], or temporal sequence mining [147], could reveal valuable insights into the low-level system behaviors that correlate with performance anomalies. These findings could lead to new detection methods that complement or even replace higher-level tracing approaches, particularly in scenarios where application-level data is inaccessible or too costly to collect.

Second, building upon the ontology of adaptive tracing presented in Figure 7.1, there is significant potential in advancing this model through the integration of natural language processing (NLP) techniques. Specifically, domain-specific ontologies [148] could be refined and expanded to represent a broader range of performance degradation types, their causes, and associated metrics. These enriched ontologies could then be used to train large language models (LLMs) capable of automatically recommending tracing instrumentation points based on defined tracing goals [149]. Such a goal-based adaptive tracing system could dynamically adapt its data collection strategy to the specific performance problem under investigation, enabling more efficient and targeted observability. This approach would allow for the automated generation of a tracing plan by providing the system with inputs such as the observed performance degradation and workload characteristics, and producing as output the likely root causes along with the exact events and metrics needed for diagnosis.

Third, while the SPA detection methodology proposed in this thesis is generalizable, there remain many SPA types that have not yet been fully addressed. Each SPA type must be observed and studied to ensure that the most relevant events are selected for its detection. The ontology- and LLM-based method described above could significantly assist in this process by automatically identifying event subsets that are most likely to capture the behaviors associated with each SPA. Expanding SPA coverage in this way would not only make the detection method more robust, but would also increase its applicability across a wider range of distributed systems and workloads.

Finally, an important avenue for future work is the real-world validation and deployment of the proposed approaches. While the methods have been evaluated in controlled testbed environments, their performance in production systems, characterized by dynamic workloads, unpredictable failure modes, and evolving architectures, remains to be assessed. Collaborating

with industry partners to integrate these techniques into existing observability pipelines [3,8] would allow us to evaluate their scalability, overhead, and diagnostic accuracy under realistic operational conditions. Such deployments could also yield valuable feedback for refining event selection strategies, enhancing model generalization, and identifying new SPA types and patterns that do not emerge in synthetic environments. This step is essential for transitioning the work from an academic proof of concept to a practical tool that can be adopted by large-scale distributed system operators.

8.5 Final Thoughts

All three parts of this work point to the same conclusion: tracing does not need to be heavy or blind. By combining lightweight monitoring with adaptive, goal-driven data collection, we can build tracing systems that are not only more efficient but also much more useful.

In summary, this thesis contributes:

- A flexible framework for adaptive tracing based on system behavior and diagnostic goals.
- Methods for detecting performance changes due to runtime behavior, code updates, or configuration changes.
- Practical tools for identifying SPAs using a combination of system and application traces.

Together, these ideas offer a path toward more intelligent observability in complex systems, enabling faster, cheaper, and more effective performance diagnostics.

REFERENCES

[1] M. Nourollahi, A. Haghshenas, and M. Dagenais, "Tracelens: Early detection of software anomalies using critical path analysis," in *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*, 2025, pp. 21–25.

[2] J. Levin, "Viperprobe: Using ebpf metrics to improve microservice observability."

[3] Y. SHKURO, *MASTERING DISTRIBUTED TRACING: analyzing performance in microservices and complex systems*. PACKT Publishing Limited, 2019.

[4] "The devops observability platform," May 2020. [Online]. Available: https://lightstep.com/

[5] M. Scrocca *et al.*, "The kaiju project: enabling event-driven observability," in *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, 2020, pp. 85–96.

[6] L. Zhang *et al.*, "Automatic observability for dockerized java applications," *arXiv preprint arXiv:1912.06914*, 2019.

[7] S. Niedermaier *et al.*, "On observability and monitoring of distributed systems–an industry interview study," in *International Conference on Service-Oriented Computing*. Springer, 2019, pp. 36–52.

[8] B. H. Sigelman *et al.*, "Dapper, a large-scale distributed systems tracing infrastructure," Tech. Rep., 2010.

[9] J. Kaldor *et al.*, "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 34–50.

[10] J. Desfossez, M. Desnoyers, and M. R. Dagenais, "Runtime latency detection and analysis," *Softw. Pract. Exper.*, vol. 46, no. 10, p. 1397–1409, 2016.

[11] P. Las-Casas *et al.*, "Sifter: Scalable sampling for distributed traces, without feature engineering," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 312–324.

[12] LTTng. Lttng documentation, efficios. [Online]. Available: https://lttng.org/docs/doc-what-is-tracing

[13] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead," *ACM Comput. Surv.*, vol. 51, no. 2, 2018.

[14] J. Blunck, M. Desnoyers, and P.-M. Fournier, "Userspace application tracing with markers and tracepoints," in *Linux Kongress*, 2009.

[15] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *USENIX Annual Technical Conference*, 2004, pp. 15–28.

[16] V. Prasad *et al.*, "Locating system problems using dynamic instrumentation," in *Ottawa Linux Symposium (OLS)*, 2005, pp. 49–64.

[17] J. Keniston *et al.*, "Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps," in *Ottawa Linux Symposium (OLS)*, 2007, pp. 215–224.

[18] M. Desnoyers, "Low-impact operating system tracing," Ph.D. dissertation, École Polytechnique de Montréal, 2009.

[19] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

[20] F. Giraldeau and M. R. Dagenais, "Recovering system metrics from kernel trace," in *Ottawa Linux Symposium (OLS)*, 2011.

[21] P.-M. Fournier, M. Desnoyers, and M. R. Dagenais, "Combined tracing of the kernel and applications with lttng," in *Proceedings of the 2009 linux symposium*. Citeseer, 2009, pp. 87–93.

[22] D. Goulet, "A unified kernel and userspace tracing architecture for linux," Ph.D. dissertation, École Polytechnique de Montréal, 2012.

[23] H. Daoud, N. E. Jivan, and M. Dagenais, "Dynamic trace-based sampling algorithm for memory usage tracking of enterprise applications," *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2017.

[24] P. Las-Casas *et al.*, "Weighted sampling of execution traces: Capturing more needles and less hay," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 326–332.

[25] O. Ertl, "Estimation from partially sampled distributed traces," *arXiv preprint arXiv:2107.07703*, 2021.

[26] E. Ates *et al.*, "An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 165–170.

[27] B. Cantrill *et al.*, "Dynamic instrumentation of production systems." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 15–28.

[28] S. Zhang *et al.*, "Diagnostic framework for distributed application performance anomaly based on adaptive instrumentation," in *2020 2nd International Conference on Computer Communication and the Internet (ICCCI)*. IEEE, 2020, pp. 164–169.

[29] "Pira: Performance instrumentation refinement automation," in *Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*, 2018, pp. 1–10.

[30] J.-P. Lehr *et al.*, "Automatic instrumentation refinement for empirical performance modeling," in *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. IEEE, 2019, pp. 40–47.

[31] J. Sun *et al.*, "Automated performance modeling based on runtime feature detection and machine learning," in *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*. IEEE, 2017, pp. 744–751.

[32] L. Castanheira, T. A. Benson, and A. Schaeffer-Filho, "The case for more flexible distributed tracing," in *Proceedings of the Student Workshop*, 2020, pp. 27–28.

[33] S. Tjandra, "Performance model extraction using kernel event tracing," Ph.D. dissertation, Carleton University, 2019.

[34] K. Yaghmour and M. R. Dagenais, "Measuringandcharacter izing systembeh av ior usingkernel-leveleventlogging," 2000.

[35] T. A. Israr *et al.*, "Automatic generation of layered queuing software performance models from commonly available traces," in *Proceedings of the 5th International Workshop on Software and Performance*, ser. WOSP '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 147–158.

[36] Q. Fournier *et al.*, "On improving deep learning trace analysis with system call arguments," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 120–130.

[37] M. Woodside, S. Tjandra, and G. Seyoum, "Issues arising in using kernel traces to make a performance model," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 11–15.

[38] A. Borghesi *et al.*, "Anomaly detection using autoencoders in high performance computing systems," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 9428–9433.

[39] Q. Fournier *et al.*, "Automatic cause detection of performance problems in web applications," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 398–405.

[40] R. R. Sambasivan *et al.*, "Diagnosing performance changes by comparing request flows." in *NSDI*, vol. 5, 2011, pp. 1–1.

[41] L. Sturmann, "Using performance variation for instrumentation placement in distributed systems," Ph.D. dissertation, 2020.

[42] H. Nemati, S. V. Azhari, and M. R. Dagenais, "Host hypervisor trace mining for virtual machine workload characterization," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2019, pp. 102–112.

[43] I. Kohyarnejadfard, M. Shakeri, and D. Aloise, "System performance anomaly detection using tracing data analysis," in *Proceedings of the 2019 5th International Conference on Computer and Technology Applications*, 2019, pp. 169–173.

[44] M. Dymshits, B. Myara, and D. Tolpin, "Process monitoring on sequences of system call count vectors," in *2017 International Carnahan Conference on Security Technology (ICCST)*. IEEE, 2017, pp. 1–5.

[45] V. Cortellessa and L. Traini, "Detecting latency degradation patterns in service-based systems," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 161–172.

[46] A. Bhattacharyya and T. Hoefler, "Pemogen: Automatic adaptive performance modeling during program runtime," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 393–404.

[47] J. Flores-Contreras *et al.*, "Performance prediction of parallel applications: a systematic literature review," *The Journal of Supercomputing*, vol. 77, no. 4, pp. 4014–4055, 2021.

[48] F. Doray and M. Dagenais, "Diagnosing performance variations by comparing multi-level execution traces," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 462–474, 2016.

[49] C. G. Atkeson, A. W. Moore, and S. Schaal, "Locally weighted learning," *Lazy learning*, pp. 11–73, 1997.

[50] A. Calotoiu *et al.*, "Using automated performance modeling to find scalability bugs in complex codes," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[51] X. Zhou *et al.*, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 683–694.

[52] "Qdime: Qos-aware dynamic binary instrumentation," in *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2017, pp. 132–142.

[53] D. J. Dean *et al.*, "Perfcompass: Toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds," 2014.

[54] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 378–393.

[55] "Visualizing request-flow comparison to aid performance diagnosis in distributed systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2466–2475, 2013.

[56] "Pinpointing the subsystems responsible for the performance deviations in a load test," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, 2010, pp. 201–210.

[57] "Automatic detection of performance deviations in the load testing of large scale systems," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 1012–1021.

[58] S. Han *et al.*, "Performance debugging in the large via mining millions of stack traces," ser. ICSE '12. IEEE Press, 2012, p. 145–155.

[59] A. Wert, *Performance problem diagnostics by systematic experimentation.* KIT Scientific Publishing, 2018.

[60] a. M. V. Surya, K. and A. Rajam, "Prediction of resource contention in cloud using second order markov model," *Computing 103*, pp. 2339–2360, 2021.

[61] A. B. Sánchez *et al.*, "Tandem: A taxonomy and a dataset of real-world performance bugs," *IEEE Access*, 2020.

[62] C. U. Smith, "Software performance antipatterns in cyber-physical systems," in *ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 173–180.

[63] M. A. Khan and N. Ezzati-Jivan, "Multi-level adaptive execution tracing for efficient performance analysis," in *2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications (SERA)*, 2023, pp. 104–109.

[64] J. H. Wert, Alexander and L. Happe, "Supporting swift reaction: Automatically uncovering performance problems by systematic experiments," in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 552–561.

[65] R. VanDonge and N. Ezzati-Jivan, "N-lane bridge performance antipattern analysis using system-level execution tracing," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2022, pp. 83–93.

[66] C. U. Smith and L. G. Williams, "Software performance antipatterns," in *Proceedings of the 2nd international workshop on Software and performance*, 2000, pp. 127–136.

[67] ——, "New software performance antipatterns: More ways to shoot yourself in the foot," in *Int. CMG Conference*, 2002, pp. 667–674.

[68] ——, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in *Computer Measurement Group Conference*, 2003, pp. 717–725.

[69] ——, *Performance solutions: a practical guide to creating responsive, scalable software.* Addison-Wesley Reading, 2002, vol. 23.

[70] A. Wert *et al.*, "Automatic detection of performance anti-patterns in inter-component communications," in *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures*, 2014, pp. 3–12.

[71] A. Avritzer *et al.*, "A multivariate characterization and detection of software performance antipatterns," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '21)*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 61–72.

[72] "Exploiting load testing and profiling for performance antipattern detection," *Information and Software Technology*, vol. 95, pp. 329–345, 2018.

[73] N. Stadelmaier, "Using software-performance-antipatterns and profiling traces to perform code-refactorings," Master's thesis, 2020.

[74] R. Capuano and H. Muccini, "A graph-based java projects representation for antipatterns detection," in *European Conference on Software Architecture*. Springer, 2023, pp. 250–265.

[75] M. De Sanctis *et al.*, "A model-driven approach to catch performance antipatterns in adl specifications," *Information and Software Technology*, vol. 83, pp. 35–54, 2017.

[76] A. Avritzer *et al.*, "Scalability testing automation using multivariate characterization and detection of Software Performance Antipatterns," *Journal of Systems and Software*, vol. 193, p. 111446, Nov. 2022. [Online]. Available: https://doi.org/10.1016/j.jss.2022.111446

[77] N. Lui, M. al Hasan, and J. H. Hill, "Real-time detection of the more is less performance anti-pattern in mysql databases," in *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*, 2021, pp. 71–78.

[78] EfficiOS, "The lttng documentation," https://lttng.org/docs/doc-what-is-tracing, 2020, accessed: 2025-08-01.

[79] P. Arumuga Nainar and B. Liblit, "Adaptive bug isolation," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, May 2010, pp. 255–264.

[80] X. Zhang *et al.*, "Early detection of host-based intrusions in linux environment," in *2020 IEEE International Conference on Electro Information Technology (EIT)*. IEEE, July 2020, pp. 475–479.

[81] M. Janecek, N. Ezzati-Jivan, and A. Hamou-Lhadj, "Performance anomaly detection through sequence alignment of system-level traces," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 264–274.

[82] F. Giraldeau and M. Dagenais, "Wait analysis of distributed systems using kernel tracing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2450–2461, 2015.

[83] M. Noferesti and N. Ezzati-Jivan, "Enhancing empirical software performance engineering research with kernel-level events: A comprehensive system tracing approach," *Journal of Systems and Software*, vol. 216, p. 112117, 2024.

[84] F. Giraldeau, "Performance analysis of distributed and heterogeneous systems using kernel tracing," PhD Dissertation, École Polytechnique de Montréal, 2015.

[85] "Eclipse tracecompass." [Online]. Available: https://eclipse.dev/tracecompass/

[86] S. Gu *et al.*, "Logging practices in software engineering: A systematic mapping study," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 902–923, 2023.

[87] A. J. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Communications of the ACM*, vol. 55, pp. 55 – 61, 2011.

[88] J. Stearley, "Towards informatic analysis of syslogs," in *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No. 04EX935)*. IEEE, 2004, pp. 309–318.

[89] X. Zhao *et al.*, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 565–581.

[90] P. A. Nainar and B. Libli, "Adaptive bug isolation," in *010 ACM/IEEE 32nd International Conference on Software Engineering. Vol. 1. IEEE*, 2010.

[91] W. Zhang *et al.*, "Adaptive tracing and fault injection based fault diagnosis for open source server software," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, 2023, pp. 729–740.

[92] D. Jeswani, M. Natu, and R. K. Ghosh, "Adaptive monitoring: A framework to adapt passive monitoring using probing," in *2012 8th international conference on network and service management (cnsm) and 2012 workshop on systems virtualiztion management (svm)*, 2012, pp. 350–356.

[93] J. Ehlers and W. Hasselbring, "A self-adaptive monitoring framework for component-based software systems," in *Software Architecture: 5th European Conference, ECSA 2011, Essen, Germany, September*, 2011, pp. 13–16.

[94] E. Ates *et al.*, "An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications," in *ACM Symposium on Cloud Computing*, 2019, pp. 165–170.

[95] X. Chen *et al.*, "Fault diagnosis for open source software based on dynamic tracking," in *7th International Conference on Dependable Systems and Their Applications (DSA), IEEE*, 2020, pp. 263–268.

[96] Q. Guan and S. Fu, "Adaptive anomaly identification by exploring metric subspace in cloud computing infrastructures," in *32nd International Symposium on Reliable Distributed Systems, IEEE*, 2013, pp. 205–214.

[97] T. Wang *et al.*, "Workload-aware anomaly detection for web applications," *Journal of Systems and Software*, vol. 89, pp. 19–32, 2014.

[98] N. Zhao *et al.*, "Identifying bad software changes via multimodal anomaly detection for online service systems," in *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 527–539.

[99] R. B. Moghaddam, Sara Kardani and K. Ramamohanarao, "Performance-aware management of cloud resources: a taxonomy and future directions," *ACM Computing Surveys (CSUR)*, pp. 1–37, 2019.

[100] L. Molkova, "Associating sampling score with the trace," https://github.com/open-telemetry/oteps/blob/0721be603a8f80bc7c1a648957e5c3b5f04fcb11/text/trace/0107-sampling-score.md, Sep. 2020, openTelemetry Enhancement Proposal (OTEP 0107). Accessed: 2025-08-01.

[101] J. MacDonald, "Probability sampling of telemetry events," https://github.com/open-telemetry/oteps/blob/928ea4fb66a2c1a0eb3400fda99ed290a2c42a73/text/0148-sampling-probability.md, Jul. 2021, openTelemetry Enhancement Proposal (OTEP 0148). Accessed: 2025-11-08.

[102] B. Beyer *et al.*, *Site reliability engineering: How Google runs production systems*, 2016.

[103] R. R. Sambasivan *et al.*, "Principled workflow-centric tracing of distributed systems," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 401–414.

[104] J. W. W. Z. H. Z. Wang, Tao and T. Huang, "Workload-aware anomaly detection for web applications," *Journal of Systems and Software*, pp. 19–32, 2014.

[105] M. Hasnain *et al.*, "Recurrent neural network for web services performance forecasting, ranking and regression testing," in *2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, 2019, pp. 96–105.

[106] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[107] V. Cortellessa, "Performance antipatterns: State-of-art and future perspectives," in *European Performance Engineering Workshop*, 2013.

[108] G. Tene, "Understanding application hiccups: An introduction to the open source jhiccup tool," *QConLondon Conference*, 2012.

[109] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: efficient detection of data race conditions via adaptive tracking," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005, pp. 221–234.

[110] C. Trubiani and A. Koziolek, "Detection and solution of software performance antipatterns in palladio architectural models," in *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 19–30.

[111] M. Desnoyers and M. Dagenais, "Lttng: Tracing across execution layers, from the hypervisor to user-space," in *Linux symposium*, vol. 101, 2008.

[112] N. E.-J. Haghshenas, Amir and M. Dagenais, "alancing instrumentation costs and performance insights: A cost-aware approach to execution monitoring."

[113] "Opentelemetry." 2024. [Online]. Available: https://www.ibm.com/products/instana/opentelemetry

[114] "Opentracing." 2024. [Online]. Available: https://opentracing.io/

[115] "Combining distributed and kernel tracing for performance analysis of cloud applications," *Electronics*, vol. 10, no. 21, p. 2610, 2021.

[116] J. Keniston *et al.*, "Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps," in *In Proceedings of the 2007 Linux symposium*, 2007, pp. 215–224.

[117] "Dtrace: dynamic tracing framework." 2024. [Online]. Available: https://dtrace.org/

[118] "Gcc finstrument-functions." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

[119] V. Cortellessa, A. D. Marco, and C. Trubiani, "Performance antipatterns as logical predicates," in *2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, 2010, pp. 146–156.

[120] Y. Gan *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," pp. 3–18, 2019.

[121] "wrk2 http benchmarking." [Online]. Available: https://github.com/giltene/wrk2

[122] "ebpf." 2024. [Online]. Available: https://ebpf.io/

[123] "perf: Linux profiling with performance counters." 2024. [Online]. Available: https://perf.wiki.kernel.org/

[124] V. Cortellessa, A. Di Marco, and C. Trubiani, "Software performance antipatterns: Modeling and analysis," in *International School on Formal Methods for the Design of Compswpringer*.

[125] W. H. Brown, R. C. Malveau, and H. W. McCormick, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[126] A. Mos, "A framework for adaptive monitoring and performance management of component-based enterprise applications," Ph.D. dissertation, Dublin City University, 2004.

[127] J. M. Voas and G. McGraw, *Software fault injection: inoculating programs against errors*. John Wiley & Sons, Inc., 1997.

[128] I. Mägi, "Distributed tracing for dummies," 2020.

[129] A. Wert, "Uncovering performance antipatterns by systematic experiments," Ph.D. dissertation, PhD thesis, Karlsruhe Institute of Technology, 2012.

[130] A. U. H. Syed and J. Iqbal, "Pad-a: performance antipattern detector for aadl," *International Journal of Information Technology*, vol. 13, pp. 259–268, 2021.

[131] R. K. Chalawadi, "Automating the characterization and detection of Software Performance Antipatterns using a data-driven approach," Master's Thesis, Blekinge Institute of Technology, Karlskrona, Sweden, Sep. 2021, supervisor: Ricardo Britto; Examiner: Emilia Mendes. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:bth-22352

[132] ——, "Automating the characterization and detection of software performance antipatterns using a data-driven approach," 2021.

[133] M. Peiris and J. H. Hill, "Towards detecting software performance anti-patterns using classification techniques," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 1, pp. 1–4, 2014.

[134]

[135] "Babeltrace." [Online]. Available: https://babeltrace.org/

[136] C. Sas and A. Capiluppi, "Weak labelling for file-level source code classification," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 698–702.

[137] J. Zhang *et al.*, "A survey on programmatic weak supervision," *arXiv preprint arXiv:2202.05433*, 2022.

[138] G. Papandreou *et al.*, "Weakly-and semi-supervised learning of a deep convolutional network for semantic image segmentation," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1742–1750.

[139] "stress-ng." [Online]. Available: https://wiki.ubuntu.com/Kernel/Reference/stress-ng

[140] M. Nourollahi *et al.*, "Software performance problem localization: A goal-based adaptive tracing approach," 2024.

[141] "Envoy proxy." [Online]. Available: https://www.envoyproxy.io/

[142] "Chatty i/o antipattern." [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/antipatterns/chatty-io/

[143] H. M. Wallach, "Topic modeling: beyond bag-of-words," in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 977–984.

[144] F. Fui-Hoon Nah *et al.*, "Generative ai and chatgpt: Applications, challenges, and ai-human collaboration," *Journal of Information Technology Case and Application Research*, vol. 25, no. 3, pp. 277–304, 2023.

[145] J. Han, J. Pei, and Y. Yin, *Mining frequent patterns without candidate generation: A frequent-pattern tree approach*. Springer, 2004, vol. 8, no. 1.

[146] M. Margahny and A. Shakour, "Fast algorithm for mining association rule," *JES. Journal of Engineering Sciences*, vol. 34, no. 1, pp. 79–87, 2006.

[147] F. Moerchen, "Algorithms for time series knowledge mining," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2006, pp. 668–677.

[148] T. R. Gruber, "Toward principles for the design of ontologies used for knowledge sharing?" *International journal of human-computer studies*, vol. 43, no. 5-6, pp. 907–928, 1995.

[149] J. G. Meyer *et al.*, "Chatgpt and large language models in academia: opportunities and challenges," *BioData mining*, vol. 16, no. 1, p. 20, 2023.