# Towards Efficient Diagnosis of Performance Bottlenecks in Microservice-Based Applications (Work In Progress paper)

Adel Belkhiri
École Polytechnique de Montréal
Montreal, Canada
adel.belkhiri@polymtl.ca

Maroua Ben Attia
Humanitas Solutions
Montreal, Canada
maroua@humanitas.io

Felipe Gohring de Magalhaes
École Polytechnique de Montréal
Montreal, Canada
felipe.gohring-de-magalhaes@polymtl.ca

Gabriela Nicolescu
École Polytechnique de Montréal
Montreal, Canada
gabriela.nicolescu@polymtl.ca

## ABSTRACT

Microservices have been a cornerstone for building scalable, flexible, and robust applications, thereby enabling service providers to enhance their systems' resilience and fault tolerance. However, adopting this architecture has often led to many challenges, particularly when pinpointing performance bottlenecks and diagnosing their underlying causes. Various tools have been developed to bridge this gap and facilitate comprehensive observability in microservice ecosystems. While these tools are effective at detecting latency-related anomalies, they often fall short of isolating the root causes of these problems. In this paper, we present a novel method for identifying and analyzing performance anomalies in microservice-based applications by leveraging cross-layer tracing techniques. Our method uniquely integrates system resource metrics—such as CPU, disk, and network consumption—with each user request, providing a multi-dimensional view for diagnosing performance issues. Through the use of sequential pattern mining, this method effectively isolates aberrant execution behaviors and helps identify their root causes. Our experimental evaluations demonstrate its efficiency in diagnosing a wide range of performance anomalies.

## CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

## KEYWORDS

Microservices, Performance analysis, Software tracing, Distributed systems

**ACM Reference Format:**
Adel Belkhiri, Maroua Ben Attia, Felipe Gohring de Magalhaes, and Gabriela Nicolescu. 2024. Towards Efficient Diagnosis of Performance Bottlenecks in Microservice-Based Applications (Work In Progress paper). In *Companion of the 15th ACM/SPEC Conference on Performance Engineering (ICPE '24 Companion), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3629527.3651432


*ICPE '24 Companion, May 7–11, 2024, London, United Kingdom.*

ACM ISBN 979-8-4007-0445-1/24/05…$15.00
https://doi.org/10.1145/3629527.3651432

## 1 INTRODUCTION

Microservices have emerged as a paradigm of choice for cloud-based applications, thanks to their scalability and flexibility. Unlike the monolithic architecture which encompasses all functionalities within a single codebase, microservices break down applications into a set of autonomous, self-contained, and single-purpose services. These services operate independently and communicate via well-defined interfaces and lightweight APIs (e.g., RESTful APIs). Such modularity enables agile scaling and promotes polyglot programming, allowing services to be developed in the languages and frameworks best suited for their tasks. Nevertheless, the compartmentalization of services often introduces challenges, particularly in debugging performance bottlenecks. In a microservice environment, the processing of user requests often requires coordinated actions from multiple services. These intricate interdependencies among microservices create a complex chain of dependencies, where a performance bottleneck in one service can trigger cascading effects that compromise the efficiency of the entire application. Identifying the culprit service and isolating the root causes of performance degradation within such a decentralized architecture proves to be a complex endeavor.

Distributed tracing [11, 15, 17, 25, 29] is a powerful method for monitoring user requests as they move through the various components of a distributed application. It tracks the end-to-end execution of user requests through the insertion of unique identifiers into requests and the propagation of metadata between processes and system components. Hence, this method provides a comprehensive view of each request's end-to-end execution, shedding light on its life cycle from initiation to completion. A "trace" represents the journey of a single request, documenting the sequence of operations it undergoes [4]. Within a trace, individual work units are captured as "spans," each corresponding to an action (e.g., a function call, database query, or instruction blocks) executed by a service or component. Spans are nested within traces to illustrate the hierarchical relationships between different operations, offering a detailed and structured view of how a request is processed across multiple services. While distributed tracing effectively captures the flow and timing of requests, it falls short of pinpointing the root causes of unexpected latencies. The reason is that it only collects

high-level information. This limitation is especially pronounced when unexpected delays stem from operating system-level resource contention, such as waiting for CPU, disk, network, or lock availability.

To address this limitation, some efforts have attempted to enrich traces generated through distributed tracing with application- and kernel-level logs, aiming to identify slow code paths, contention for resources, and load imbalances [3, 26]. However, these approaches often fail at detecting and diagnosing transient and short-lived performance problems. Another approach involves leveraging vertical context propagation to inject application-level events into kernel traces [2, 4]. This provides a granular understanding of system behavior but comes at the cost of additional complexity and significant overhead. Additionally, several statistical and machine-learning methods have been explored for analyzing the performance of distributed applications [5, 10, 16, 18, 21, 24, 28]. While these methods offer powerful analytical capabilities, they come with many limitations, such as low detection accuracy and computational inefficiency. In short, although the proposed approaches offer insights into different types of performance problems, they mostly struggle to accurately identify the root causes of these issues.

In this paper, we propose a novel approach for identifying performance problems in microservice applications and uncovering their underlying causes. Based on this approach, we implement a cross-layer analysis enabling the characterization of request executions. Additionally, we leverage a combination of distributed and software tracing techniques to capture both kernel- and application-level events. We use a small subset of kernel events to conduct fine-grained critical path analysis of service threads, and application-level events to delimit the spans of operations involved in request processing. By utilizing a sequential pattern mining technique, we extract sequences of thread states that characterize the behavior within each request category. Using these normative patterns as a basis, we identify anomalous request executions. Anomalies are flagged when the observed behavior diverges significantly from the established patterns.

The rest of the paper is organized as follows. Section 2 introduces our approach and elaborates on the design details of the implemented framework. Section 3 evaluates the effectiveness of our framework in practical scenarios through an illustrative use case. It also assesses the overhead it induces and discusses avenues for potential improvements. Section 4 reviews relevant works in the field that have informed our research. Finally, Section 5 concludes this paper and outlines directions for future work.

## 2 PROPOSED SOLUTION

The framework we developed to implement this approach is specifically designed for seamless integration with distributed tracing infrastructures supporting OpenTelemetry (OTel) [6]. OTel is an open-source initiative that provides a comprehensive suite of APIs, libraries, agents, and instrumentation designed to enhance observability in distributed applications. Its main goal is to provide developers with a unified way to collect distributed traces and metrics through instrumentation. The vendor-neutral design of OTel makes it compatible with a wide range of distributed tracers, including but not limited to Jaeger [15] and Zipkin [29]. Consequently, this design consideration greatly simplifies the integration of our framework into existing systems.

Our framework aims to detect performance issues in microservice-based applications and uncover their root causes through offline analysis. To pinpoint performance anomalies, our framework requires two separate sets of trace data - referred to as the 'baseline' and 'test' datasets. The baseline dataset is used to generate a basis for modeling the software's normal behavior, while the test dataset is evaluated against this baseline to identify any deviation or abnormality. Fig. 1 depicts the architecture of our framework and outlines its key operations. These elements will be discussed in greater details in the following sections.

### 2.1 Capturing Execution Traces

Our approach uses cross-layer tracing to collect fine-grained data that characterizes resource consumption per request. Therefore, to avoid the overhead associated with vertical context propagation and the need to modify the application source code, we chose to instrument OTel libraries using the Linux Trace Toolkit Next Generation (LTTng) [7]. LTTng is a high-throughput tracer for Linux-based systems that is designed for low-overhead tracing of applications at kernel and user-space levels. The instrumentation we added to OTel aims to emit userspace events each time a service starts or finishes the processing of a request. Hence, we inserted tracepoints into the API methods responsible for starting and ending spans. For example, to gather data from C++-based microservices, we instrumented the *Tracer* class's *StartSpan()* and *end()* methods within the opentelemetry-cpp library. We extended our instrumentation to multiple OTel libraries, as a microservice application may consist of services developed in various programming languages. Our analysis requires also gathering specific kernel events to construct critical paths for request executions. The Linux kernel comes with hundreds of tracepoints, allowing us to capture needed events without additional instrumentation. A description of a subset of leveraged kernel events is provided in Table 1.

### 2.2 Classification of Traces

After collecting execution traces, our framework starts analyzing them to identify potential performance anomalies. The analysis is based on the hypothesis that operations in traces of the same type should exhibit similar performance characteristics when processed by the same services. Therefore, categorizing traces based on their types is critical. We consider traces to be of the same type if they present the same workflow. A trace workflow outlines the order in which operations and requests are executed within individual processes and across the various services that compose a microservice-based application. Our framework recreates trace workflows by generating tree structures from the operation names exported via the userspace events mentioned earlier. The tree's root node is labeled with the name of the initiating request, which is also known as the root span. The remaining nodes are labeled with the names of their respective requests/operations. Unique identifiers for tasks performed by services are formulated by pairing the "service_name" with the "operation_name". After building the workflow trees, we use a hash function to generate an identifier for the trace type that captures both the labels of the nodes and their

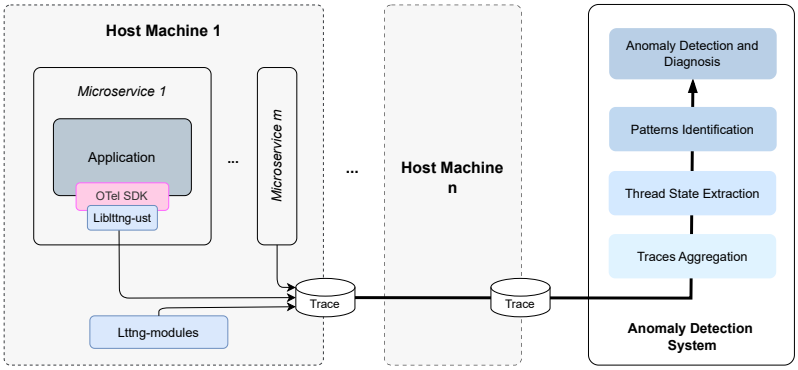


**Figure 1: The operation of our framework is based on collecting kernel-level events and leveraging an instrumented version of OTel to add information about the start and end of spans to the kernel trace**

**Table 1: A subset of the kernel events required for our analysis**

| Tracepoint | Description |
|---|---|
| *sched_switch* | Signals that a new thread has taken over from a previously active thread on a CPU. |
| *sched_wakeup* | Triggered when a thread, previously in a blocked state, is now ready to execute. |
| *softirq_entry/exit* | Indicates the start/end of a software interrupt handler's execution. |
| *irq_handler_entry/exit* | Indicates the start/end of a hardware interrupt handler's execution. |
| *timer_expire_entry/exit* | Indicates the start/end of a timer interrupt's execution. |
| *sched_process_fork* | Fires when a new process is created by the kernel. |

relative positions within the tree structure. Hence, traces whose workflows produce identical type identifiers are classified in the same category.

## 2.3 Extraction of Thread States

Our second hypothesis is that when processing operations of the same type, the service threads will follow a consistent sequence of states. For example, let us consider a basic authentication service and its thread states during operation. When a login request is received, the service initially validates the provided username and password against predefined criteria (state: running). It then retrieves the associated hashed password from a disk (or a database), keyed by the username (state: *blocked for disk*). After that, the service compares the stored hashed password with the hashed version of the received password (state: *running*). Finally, the outcome of this comparison is transmitted back to the originating service (state: *blocked for network*). Therefore, it is reasonable to assume that the service thread will follow the same sequence of states when processing future requests. If it deviates from the expected sequence, either there is something wrong with its behavior or other unknown factors at play. This unexpected behavior could potentially reflect performance issues like contention for resources, defective hardware, or slow functions.

To ascertain the states through which operations progress during their execution, our framework identifies the critical path of the subsequent service threads. In software engineering, the "critical path" refers to the sequence of dependencies that inherently limit the speed at which a thread can be completed [27]. Threads rely on hardware and software resources for execution. Their hardware dependencies include but are not limited to the need for CPU time, disk access, or network bandwidth. As for the software dependencies, it may involve waiting for data from other threads or requiring certain locks or semaphores to be available for synchronization. Practitioners often use critical path analysis to assess software resource bottlenecks and gain insight into a process's interactions with system resources and other processes.

Based on the algorithm proposed in [12], we developed an analysis in Trace Compass [9] to extract the critical path of services' threads and obtain their states during operations' execution. We converted operations execution into a text-based representation, wherein each unique thread state is encoded as a distinct letter of the alphabet. For example, the "Running" state is represented by the letter 'R', the state "blocked for Disk" is represented by 'D', the state "blocked for Network" is denoted by 'N', and the state "blocked for Timer" is denoted by 'T'. It is worth noting that our analysis is based on 8 thread states as we excluded some states that are not indicative of application behavior (e.g., the 'Interrupted' and 'Preempted' states). In addition, we introduced an extra state, symbolized by the letter 'Z', to represent the execution of sub-operations. Our framework leverages this state to achieve accurate anomaly detection as the conducted analysis is guided by the operations hierarchy (see Fig. 2).

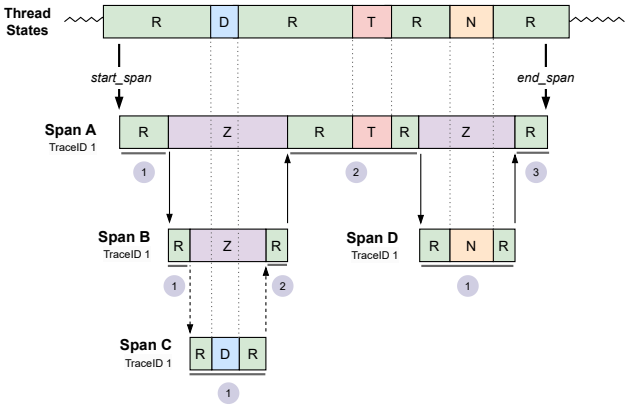


**Figure 2: Thread states are transposed to corresponding spans and sub-spans. Spans' intervals are highlighted with a bold grey line underneath it**

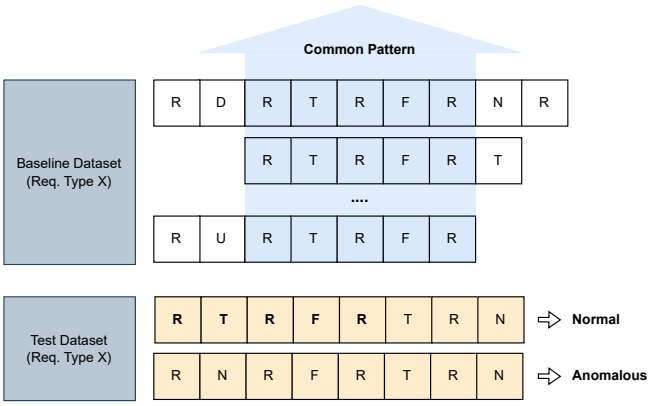


**Figure 3: Sequential pattern mining is used to extract execution state patterns from same-type trace operations.**

## 2.4 Frequent Pattern Mining

Establishing a baseline for what is considered normal behavior while processing operations is crucial for detecting performance bottlenecks and anomalies. Our approach focuses on identifying recurring patterns in thread execution states during these periods. We achieve this through sequential pattern mining, a technique specifically designed to uncover recurring subsequences within a set of sequences. As we can observe in Fig. 3, these subsequences correspond to frequent sequences of execution states. We assess the relevance of discovered state subsequences based on their lengths and frequency of occurrence.

Sequential pattern mining is not limited to our context but finds broad applications in various areas like financial market prediction, and text analysis. It is widely used to identify frequently occurring ordered events or subsequences in various types of datasets. Formally speaking, let $D = \{s_1, s_2, \ldots, s_n\}$ be a set of sequences, where each sequence $s_i$ is an ordered list of items $\langle a_1, a_2, \ldots, a_m \rangle$. An itemset $X$ is said to be a "sequential pattern" if it appears in at least *minsup* number of sequences in $D$, where *minsup* is a predefined minimum support threshold. On the other hand, a sequence $s = \langle a_1, a_2, \ldots, a_m \rangle$ is said to "contain" an itemset $X = \{x_1, x_2, \ldots, x_k\}$ if there exists a subsequence $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_k} \rangle$ such that $a_{i_j} = x_j$ for all $j$ from 1 to $k$. The aim is to find all such itemsets $X$ that satisfy the minimum support condition in the given dataset $D$.

Our approach is based on the implementation of the algorithm proposed in [20] to identify all closed sequential patterns of thread states that are present in the dataset. A closed sequential pattern can be defined as a sequential pattern that is not a strict subset of any other pattern with identical support. This algorithm further allows us to impose constraints on the gaps between states, thereby offering a more flexible way to mine recurring sequential states. The minimal support value required for identifying patterns is a user-defined parameter, but we recommend setting it at 95% or higher. Setting this parameter at a high value allows prioritizing patterns that are more common in the dataset and filtering out infrequent ones. This would improve the effectiveness of our anomaly detection analysis and ensure its scalability for larger datasets.

## 2.5 Performance Anomaly Diagnosis

Our methodology, as we explained in the previous section, relies on extracting from the baseline dataset patterns that encode the service runtime behavior during operations execution. This involves clustering similar traces based on their type identifiers. Then, critical path analysis is used to extract, for each operation, threads' execution states as illustrated in Fig. 2. Moreover, the execution of each operation is divided into intervals based on the occurrence of the 'Z' state. Thus, the number of intervals is equal to the number of sub-operations plus one. For instance, in Fig. 2, *Span A* is divided into three intervals, *Span B* into two, and *Span C* and *Span D* each into a single interval. Subsequently, we apply sequential pattern mining to identify patterns in thread states within these intervals, and we compute the minimum and maximum latencies for each state encompassed by these patterns. These latency thresholds are established using the three-sigma rule, thereby enriching the pattern states with mean and standard deviation information.

The second step in our methodology is the evaluation of the test dataset to ascertain whether its operations are normal or anomalous. For each trace in this set, we identify its type and extract the thread states occurring while executing its operations. We also determine the states that correspond to the operation intervals and require validation. By comparing the generated trace type identifiers with type identifiers in the baseline dataset, we determine the frequent state patterns that must be validated against the state sequences in each operation interval. Therefore, for every operation interval, we check if the observed state sequence matches the expected pattern. If this is the case, we check whether the states of the sequence matching the pattern are within the identified limits. This is done through a bottom-up approach, where sub-spans are evaluated before their parent spans. If a discrepancy is found during this verification process, the operation in question is flagged as anomalous, triggering a more detailed investigation to pinpoint the cause of the deviation. Furthermore, this hierarchical evaluation serves as a structured way to address potential issues at the granular level of sub-spans and execution intervals, which simplifies isolating and resolving performance problems.

# 3 EVALUATION AND DISCUSSION

To demonstrate the effectiveness of our approach in diagnosing performance anomalies in microservice applications, we leverage "Bank-of-Sirius," an HTTP-enabled web application that emulates a banking system [23]. This application allows users to create bank accounts and execute financial transactions. We chose Bank-of-Sirius for our evaluation because it comes pre-instrumented with OTel and features a diverse architecture, comprising nine microservices implemented in Python, Java, and C (Fig. 4).

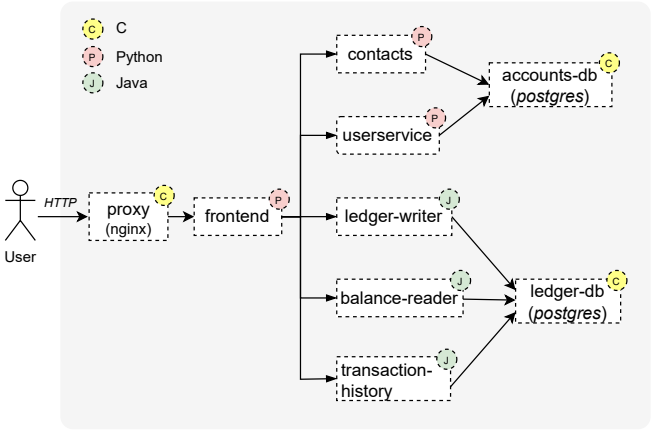


**Figure 4: Bank-of-sirius Architecture**

## 3.1 Use case

In this case study, we allocated one virtual machine (VM) to host the Java-based services and the *ledger-db* service, and another VM for the remaining services. Each service was containerized using Docker to ensure isolated execution environments. We then developed and executed a benchmark script simulating the activities of 100 concurrent users. Each user interacts with the *frontend* service through a specific sequence of HTTP requests: a "/login" request, followed by a "/home" request, and lastly a "/logout" request. It is important to note that the "/home" request is responsible for loading the user's home page, which displays profile information, account balance, and transaction history. Accomplishing this requires the *frontend* service to make sub-requests to the *contacts*, *balance-reader*, and *transaction-history* services. We traced the benchmark execution and used our framework to establish the sequences of thread states involved in the processing of the "/login", "/home", and "/logout" requests. This experiment was repeated tenfold at various time intervals to create a baseline dataset representing application normal performance.

To create our test datasets, we conducted two separate interactions with our target application. For the first, we emulated typical user behavior: logging in, visiting the homepage, and logging out. In the second interaction, we altered the configuration of the *contacts* service by changing the type of Gunicorn workers from *synchronous* to *gThread*. This is a notable change given that all Python-based services in Bank-of-Sirius are Flask applications serviced by Gunicorn servers. Following this modification, we duplicated the initial user behavior. Both interactions were traced, allowing us to capture and compile requests' state sequences into two distinct test datasets.

To identify potential anomalies within the test datasets, our framework scrutinized the captured state sequences for consistency with the established patterns from the baseline dataset. The analysis revealed no anomalies in the first dataset; however, it flagged irregularities in the "contacts" service state sequences in the second dataset. Specifically, we observed that the latencies of this service's spans were unexpectedly lower, and there was a recurrent absence of the "N" (Network) states across numerous spans. This aligns with the variation in Gunicorn operations observed when employing the Sync and gThread models, as illustrated in Fig. 5. Gunicorn uses a pre-fork worker model where a master process manages a set of worker processes dedicated to handling client requests. In the Sync model, each worker handles connections and executes requests one at a time, leading to a simple but less concurrent workflow. Conversely, in the gThread model, each worker pools connections, spawns multiple threads, and distributes tasks across them to efficiently handle simultaneous requests. That explains why the latencies of requests in the latter configuration were lower and the 'N' states were missing from the sequence of states related to the contact service.

## 3.2 Discussion

In this section, we delve into the intricacies of state pattern recognition and its implications for our anomaly detection mechanism. A notable observation was the presence of repetitive pairs of states within the identified patterns, suggestive of specific activities like prolonged network communication or disk read operations (e.g., R-T-R-D-R-D-R-D-R). These repetitive sequences, while indicative of certain behaviors, posed a limitation in the versatility of our pattern-matching algorithm. To enhance our framework's adaptability and precision, we have introduced a transformative step that condenses these repetitive pairs into regular expressions when their occurrence is frequent (e.g., R-T-(R-D)+-R). This refinement not only streamlines the pattern detection process but also enriches our framework's ability to discern more complex behaviors while simplifying the representation of state sequences.

Additionally, we encountered scenarios where the latencies associated with certain states displayed substantial variability, challenging our framework's ability to discern normative from anomalous behavior. For instance, the 'Running' state in a computationally intensive operation, such as factorial computation, can exhibit significant latency fluctuations based on the input magnitude. To address this, we propose the analysis of the latency distributions, particularly for operations marked by a high coefficient of variation. This strategy involves the use of call stack profilers to add a finer granularity to our analysis by accounting for function calls as subspans. This approach allows us to reclassify requests from certain types based on the unique footprint of the executed function calls and their parameters. It would also enable a more tailored detection process that accounts for the distinctive nature of each operation within our microservice architecture.

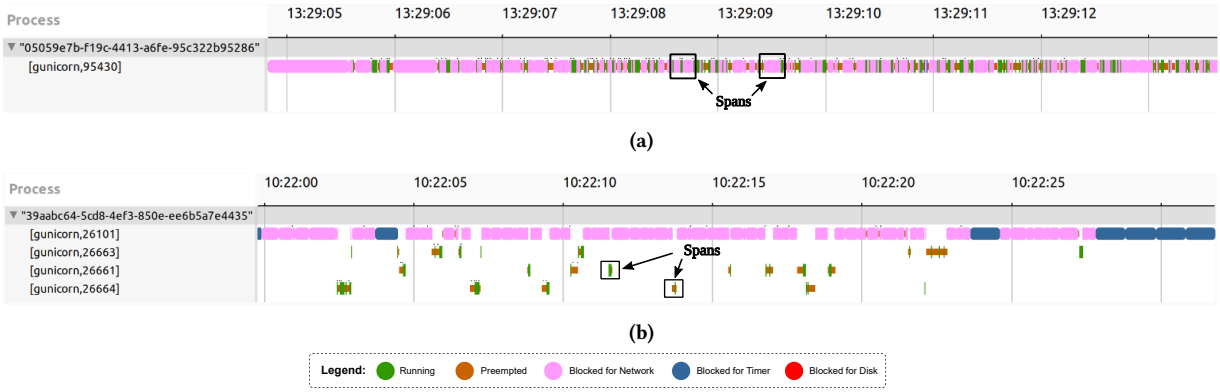


**Figure 5: Critical paths of a single Gunicorn worker in two configurations: Sync (a) and gThread (b). In the Sync configuration, the worker manages both communication and task processing, reflected in running and network states. In contrast, gThread configuration involves the worker handling communication while task processing is distributed among its threads**

## 3.3 Overhead Analysis

Minimizing tracing overhead is essential to prevent skewed results. Excessive overhead may alter the system's normal operation, making the tracing solution impractical for use in production environments. Therefore, to evaluate the overhead incurred by tracing, we conducted performance benchmarks on the Bank-of-Sirius application both with tracing enabled and disabled. We subjected this application to varying workloads and observed its response times. For these tests, we employed Locust [13], an open-source load testing utility, to simulate clients issuing requests at different rates. These clients issue different types of requests to provide a comprehensive view of the system's performance under load. The benchmarking was carried out on a machine equipped with 16 GB of RAM and an Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz. The software environment consisted of Ubuntu 22.04, featuring the 5.15.0-60 kernel version, LTTng 2.12 for tracing, and Docker 24.0.5 for containerization.

The outcomes of our investigation are presented in Fig 6. The latter illustrates the application's response time to "/home" requests—identified as the most resource-intensive requests. Within our test environment, the application under test achieves a peak throughput of 45 requests per second. The graph indicates that the activation of the required tracepoints introduces a negligible performance impact, with tracing causing only a 2 to 4% increase in response time. Interestingly, our benchmark's results also show a slight improvement in the application's response time with tracing enabled at a rate of 5 requests per second. Under identical conditions, this improvement would be unexpected. Nevertheless, a degree of fluctuation is inherent in operating system operations due to many factors such as the scheduling of system processes, and memory page faults. Given that the overhead from tracing a relatively small number of events is almost imperceptible, it becomes challenging to measure and can be smaller than the natural variability of the operating system's performance. In short, our overhead analysis shows that tracing Bank-of-Sirius introduces a marginal increase in its response times, thus confirming the suitability of our framework for production environments.

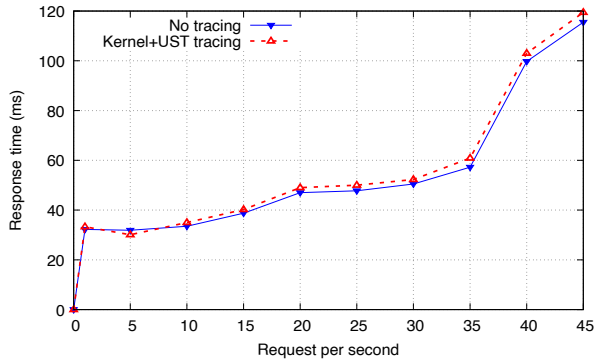


**Figure 6: Bank-of-Sirius's response time to "/home" requests with tracing disabled and enabled**

## 4 RELATED WORK

There is extensive prior work on monitoring and debugging performance problems in distributed and microservice-based applications. Most of it is based on the use of distributed tracing for collecting monitoring data from a distributed system. Distributed tracing indeed provides a broad overview of end-to-end request processing in microservice-based applications. Nonetheless, the information it produces is insufficient to pinpoint the causes of detected latency issues. Many strategies were hence proposed to enrich the span-based traces with data collected from application- and kernel-level logs and tracepoints [3, 26]. For example, authors in [3, 26] proposed an automated instrumentation framework that runs alongside the distributed tracing infrastructure. Their framework combines distributed tracing and variance-based control logic to explore at runtime where logs/tracepoints need to be enabled to effectively help diagnose performance problems. The main limitation of the proposed framework is its incapacity to provide value in diagnosing transient and short-lived performance problems.

On the other hand, various cross-layer tracing techniques have been used in literature to enhance the understanding of distributed

applications behaviors [1, 2, 4, 19]. For example, authors. in [2] inject application-level events into kernel traces by executing a series of innocuous system calls for each high-level event of interest (e.g., the start and end of an RPC call). These system calls serve as synchronization points in the trace to merge high-level and low-level events. Also, in [4], Belkhiri et al. relied on vertical context propagation to inject high-level request identifiers into the kernel. The weakness of this approach is that it poses scalability challenges as it requires a system call each time the target application starts or completes the processing of a request. There are also numerous attempts to diagnose performance anomalies by applying statistics, graph theory, and trace comparison techniques on collected traces [8, 14, 22]. For instance, Huang et al. in [14] leverage the structure within the distributed traces to group similar traces and provide detailed statistics at each level of the trace hierarchy. Their tool can assist practitioners in identifying the relevant operations to focus on when debugging but cannot identify the cause of the issue automatically.

## 5 CONCLUSION

Microservices often complicate the debugging of unexpected latencies in application operations and pinpointing their root causes. This paper addresses this issue by proposing an innovative approach for diagnosing performance anomalies in microservice applications. Our approach leverages cross-layer tracing to enhance the granularity of observability, providing a multi-dimensional view that correlates system resource metrics with user requests. The use of sequential pattern mining enables the isolation of anomalous behavior patterns and facilitates the identification of their root causes. Our evaluations have not only confirmed the efficacy of our framework in identifying performance anomalies but also demonstrated its operational efficiency by maintaining minimal overhead.

As distributed systems evolve, diagnosing performance grows increasingly complex. Our contribution represents a step forward in mitigating this challenge by equipping developers and system operators with a tool capable of identifying and diagnosing performance issues without invasive instrumentation or prohibitive performance penalties. We expect that our findings will incite further research into optimizing distributed tracing infrastructures and developing even more sophisticated analysis techniques. Future work could explore the potential for real-time anomaly detection and automated remediation, which would enhance further the resilience and reliability of microservice-based applications.

## REFERENCES


[1] Marcelo Amaral, Tatsuhiro Chiba, Scott Trent, Takeshi Yoshimura, and Sunyanan Choochotkaew. 2022. MicroLens: A Performance Analysis Framework for Microservices Using Hidden Metrics With BPF. *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)* 00 (2022), 230–240.

[2] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Vol. 00. USENIX Association, Renton, WA, 405–417.

[3] Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K. Coskun, and Raja R. Sambasivan. 2019. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. *Proceedings of the ACM Symposium on Cloud Computing* (2019), 165–170.

[4] Adel Belkhiri, Ahmad Shahnejat Bushehri, Felipe Gohring de Magalhaes, and Gabriela Nicolescu. 2023. Transparent Trace Annotation for Performance Debugging in Microservice-oriented Systems (Work In Progress Paper). *International Conference on Performance Engineering (ACM/SPEC)* (2023), 25–32.

[5] Yang Cai, Biao Han, Jinshu Su, and Xiaoyan Wang. 2021. TraceModel: An Automatic Anomaly Detection and Root Cause Localization Framework for Microservice Systems. *2021 17th International Conference on Mobility, Sensing and Networking (MSN)* 00 (2021), 512–519.

[6] CNCF. 2023. OpenTelemetry: high-quality, ubiquitous, and portable telemetry to enable effective observability. https://opentelemetry.io/

[7] Mathieu Desnoyers and Michel R Dagenais. 2006. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*, Vol. 2006. Citeseer, 209–224.

[8] Prem Devanbu, Myra Cohen, Tao Xie, and Liangfei Su. 2020. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020).

[9] Ericsson. 2023. Trace Compass. http://tracecompass.org/

[10] Dan Feng, Steffen Becker, Nikolas Herbst, Philipp Leitner, Zheng Papadopoulos, Hyunseok Chang, Sarit Mukherjee, and Eric Eide. 2022. LongTale: Toward Automatic Performance Anomaly Explanation in Microservices. *International Conference on Performance Engineering (ACM/SPEC)* (2022), 5–16.

[11] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USENIX Association, Cambridge, MA.

[12] Francis Giraldeau and Michel Dagenais. 2015. Wait Analysis of Distributed Systems Using Kernel Tracing. *IEEE Transactions on Parallel and Distributed Systems* 27, 8 (2015), 2450–2461.

[13] Jonatan Heyman, Joakim Hamrén, Carl Byström, and Hugo Heyman. 2023. Locust: An open-source load testing tool. https://locust.io/

[14] Lexiang Huang and Timothy Zhu. 2021. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. *Proceedings of the ACM Symposium on Cloud Computing* (2021), 76–91.

[15] Jaegertracing.io. 2023. Jaeger: Open Source, End-to-End Distributed Tracing. http://jaegertracing.io

[16] Madeline Janecek, Naser Ezzati-Jivan, and Seyed Vahid Azhari. 2021. Container Workload Characterization Through Host System Tracing. *2021 IEEE International Conference on Cloud Engineering (IC2E)* 00 (2021), 9–19.

[17] Jonathan Kaldor, Jonathan Mace, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), 34–50.

[18] Iman Kohyarnejadfard, Daniel Aloise, Seyed Vahid Azhari, and Michel R. Dagenais. 2022. Anomaly detection in microservice environments using distributed tracing data analysis and NLP. *Journal of Cloud Computing* 11, 1 (2022), 25.

[19] Cheryl Lee, Tianyi Yang, Zhuangbin Chen, Yuxin Su, and Michael R Lyu. 2023. Eadro: An End-to-End Troubleshooting Framework for Microservices on Multi-source Data. *arXiv* (2023). arXiv:2302.05092

[20] Chun Li and Jianyong Wang. 2008. Efficiently Mining Closed Subsequences with Gap Constraints. *Proceedings of the 2008 SIAM International Conference on Data Mining* (2008), 313–322.

[21] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, and Dan Pei. 2020. Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks. *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)* 00 (2020), 48–58.

[22] Lun Meng, Feng Ji, Yao Sun, and Tao Wang. 2021. Detecting anomalies in microservices with execution trace comparison. *Future Generation Computer Systems* 116 (2021), 291–301.

[23] Inc. NGINX. 2023. Bank of Sirius. https://github.com/nginxinc/bank-of-sirius

[24] Tim Sherwood, Emery Berger, Christos Kozyrakis, Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), 135–151.

[25] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc.

[26] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K. Coskun, and Raja R. Sambasivan. 2021. Automating instrumentation choices for performance problems in distributed applications with VAIF. *Proceedings of the ACM Symposium on Cloud Computing* (2021), 61–75.

[27] Dean M. Tullsen and Brad Calder. 1998. *Computing along the critical path.* Technical Report. Technical report, University of California, San Diego.

[28] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 655—672.

[29] Zipkin.io. 2022. Zipkin. https://zipkin.io