Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

---

# Investigating Performance Overhead of Distributed Tracing in Microservices and Serverless Systems

---

**Author:** Anders Nõu (2779368)

*1st supervisor:* Dr. Daniele Bonetta
*daily supervisor:* Ir. Sacheendra Talluri
*2nd reader:* Prof. Dr. Alexandru Iosup

*A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science*

January 29, 2025

## Abstract

In modern software architecture, distributed tracing has become essential for achieving observability in distributed systems. However, distributed tracing also introduces a trade-off between the depth of visibility and overhead. This thesis explores the performance impact of distributed tracing on microservices and serverless applications by measuring the throughput and latency of microservices and sererless applications across different configurations. We use the TechEmpower Framework Benchmarks and the Serverless Benchmark Suite (SeBS) for evaluation. We categorize and quantify overhead sources into configuration, instrumentation, and exporting to understand which operations contribute the most to performance degradation. We designed and conducted three experiments with different programming frameworks and instrumentation tools. The results show that distributed tracing significantly affects the throughput and latency of microservices, with throughput decreasing in the range of 19-80%. In addition, we observe that the performance impact on serverless applications is significant, with short-duration applications exhibiting up to 175% latency increase and longer-duration applications showing an increase of 6.7%. The median latency increased around 7-42% for microservice applications. The main contributors to performance degradation were the configuration and export stages, with configuration significantly impacting serverless cold-start scenarios. The findings from this research help us understand the performance impact of distributed tracing on applications and what are the main contributors to overhead.

# Contents


**List of Figures** **iii**

**List of Tables** **v**

**1 Introduction to Distributed Tracing, Visibility, and Overhead** **1**

1.1 Concept of Visibility, Monitoring, and Overhead . . . . . . . . . . . . . . 2

1.2 Problem Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.3 Research Questions and Methodology . . . . . . . . . . . . . . . . . . . . 5

1.4 Main Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.5 Thesis Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

**2 Background on Distributed Tracing, Automatic Instrumentation, and Serverless Applications** **11**

2.1 Distributed Tracing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.2 Instrumentation and Instrumentation Frameworks . . . . . . . . . . . . 12

2.3 Function as a Service Platform . . . . . . . . . . . . . . . . . . . . . . . 15

2.4 Experiment Environment . . . . . . . . . . . . . . . . . . . . . . . . . . 17

**3 Evaluation of Request-Based Applications' Tracing Overhead: Experimental Design and Results** **21**

3.1 Experiment Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

3.2 Metrics and Data Collection . . . . . . . . . . . . . . . . . . . . . . . . 28

3.3 Experiment Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3.4 Experiment Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . 36

3.5 Experiment Deployment . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

3.6 Experiment Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

**4 Evaluation of Serverless Applications' Tracing Overhead: Experimental Design and Results 47**
4.1 Experiment Design . . . 47
4.2 Metrics and Data Collection . . . 51
4.3 Experiment Setup . . . 53
4.4 Experiment Configuration . . . 55
4.5 Experiment Deployment . . . 59
4.6 Experiment Results . . . 61

**5 Sources of Overhead in Distributed Tracing 69**
5.1 Experiment Design . . . 69
5.2 Metrics and Data Collection . . . 72
5.3 Experiment Setup . . . 76
5.4 Experiment Configuration . . . 78
5.5 Experiment Deployment . . . 79
5.6 Experiment Results . . . 82

**6 Related Work 89**
6.1 Research Gaps . . . 89
6.2 Performance Overhead of Tracing . . . 90
6.3 Comparative Analysis of Tracing Tools . . . 91

**7 Conclusion 95**
7.1 Research Questions . . . 96

**Appendix 99**
A Results Data . . . 99
B Task-based application experiment benchmark execution record . . . 99
C Task-based application *graph-pagerank* manual instrumentation code example . . . 100

**References 103**

# List of Figures

1.1 Illustration of the relationship between monitoring, visibility, and overhead. The intersection of the concepts represents balanced monitoring with sufficient visibility and acceptable overhead. . . . . . . . . . . . . . 3
1.2 Overview of the three experiments conducted in the thesis. The figure highlights the technology stacks and tracing tools used in the experiments. Experiment 1 is described in Chapter 3, experiment 2 in Chapter 4, and experiment 3 in Chapter 5. . . . . . . . . . . . . . . . . . . . . . . . 7
1.3 The thesis contents and reading structure. . . . . . . . . . . . . . . . . . . 9

2.1 Example of a trace displaying a request traveling through four different services. The request is an example from the OpenTelemetry Demo system[1]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.1 Request-based application requests-per-second throughput calibration workflow. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.2 Architecture overview of the request-based (microservices) applications latency and throughput overhead experiment. . . . . . . . . . . . . . . . . 28
3.3 Aggregated throughput (requests per second) for request-based applications with different instrumentation configurations (Standard, OpenTelemetry, Elastic APM) across the evaluated programming stacks (Python Flask, Java Spring, Go http, Node.js). . . . . . . . . . . . . . . . . . . . 40
3.4 The median HTTP request duration in milliseconds for request-based applications across different instrumentation configurations for four benchmark endpoints. The request durations are aggregated over the programming frameworks (Python Flask, Java Spring, and Go http). . . . . . . . 43

## LIST OF FIGURES

4.1 Task-based application experiment architecture diagram. Kubernetes hosts the serverless platform OpenWhisk, which deploys the benchmark applications, and the SeBS invokes the benchmarks. The OpenTelemetry Collector and Kibana are containerized in Docker. . . . . . . . . . . . . . . 48
4.2 Configuration example for manually instrumented Python application with OpenTelemetry. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.3 Comparison of Python and Node.js code snippets for the graph-pagerank application. This figure shows part of the code responsible for manually instrumenting using OpenTelemetry. The code examples highlight the similarities in how the instrumentation is implemented in both frameworks. Both examples demonstrate span creation, context propagation, and attribute setting. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.4 Comparison of performance impacts of distributed tracing for different task-based benchmarks, showing the mean benchmark time for non-instrumented and instrumented configurations across multiple benchmarks. The results are combined from both Python and Node.js applications. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

5.1 Flame graph of the *dynamic-html* task-based application with OpenTelemetry instrumentation. The CPU threshold for displaying function calls is set to 2.5%. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
5.2 DOT graph of a task-based application with OpenTelemetry instrumentation. The graph is generated with gprof2dot (1). The edge and node CPU threshold is set to 9.5%, and the depth of the graph is set to two for readability. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
5.3 Experiment architecture for evaluating the performance impact of distributed tracing processes. . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.4 Distribution of tracing overhead categories (Configuration, Instrumentation, Export, Task) across 100 runs for task-based applications under cold start and warm start scenarios. . . . . . . . . . . . . . . . . . . . . . 85
5.5 Comparison of distributed tracing overhead across configuration, instrumentation, and export phases for serverless applications and microservices. 86

# List of Tables


2.1 Software components and their versions used throughout the experiments. 19

3.1 The evaluation of request-based applications' tracing overhead experiment design overview. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

3.2 Initial requests per second configuration for the throughput evaluation for request-based applications. . . . . . . . . . . . . . . . . . . . . . . . . 26

3.3 Requests per second incrementation values for the throughput evaluation of request-based applications. . . . . . . . . . . . . . . . . . . . . . . . 26

3.4 The list of metrics used to evaluate the request-based applications' performances in the experiments. . . . . . . . . . . . . . . . . . . . . . . . . 29

3.5 Experiment configuration parameters for the request-based application experiments. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

3.6 Throughput (requests per second) comparison of OpenTelemetry, Elastic APM, and non-instrumented request-based applications across Python Flask, Java Spring, Go http, and Node.js frameworks. . . . . . . . . . . 41

3.7 Request-based applications latency comparison across different instrumentation configurations and programming languages. All values are displayed in milliseconds. . . . . . . . . . . . . . . . . . . . . . . . . . . 44

4.1 Task-based applications memory limit and timeout configuration. . . . . 59

4.2 Performance metrics (mean and p99) for the serverless benchmarks comparing instrumented (Instr.) and non-instrumented (Non-Instr.) configurations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

4.3 Benchmark and Client results for Python and Node.js overhead percentages. Overhead indicates the latency increase with the instrumented version of the benchmark. . . . 64

5.1 Parameters used for the request-based and task-based benchmarks for evaluating the sources of overhead. . . . 80
5.2 Sources of distributed tracing overhead in request-based and task-based applications based on profiling data. The percentages for each category represent the average contribution of the operations. . . . 83

6.1 Performance overhead comparison of various tracing tools across frameworks and metrics. . . . 93

# 1

# Introduction to Distributed Tracing, Visibility, and Overhead

In recent years, the adoption of serverless computing has changed the way applications are developed and deployed, offering significant advantages such as reduced operational complexity, automatic scaling, and cost efficiency (2, 3). Serverless computing allows developers to execute task-based applications without managing the underlying infrastructure, significantly simplifying the development process and reducing the operational overhead (4, 5). This paradigm shift has led to widespread adoption in various domains, including web development, data processing, and microservice architectures (6). However, as serverless computing becomes more prevalent, the necessity for reliable monitoring and debugging tools has grown significantly more important (7).

Distributed tracing has emerged as a necessary tool for monitoring and diagnosing the performance of distributed systems, including microservices and serverless applications (8, 9). By tracking requests as they move through various services and components, distributed tracing provides insights into the behavior and interactions within systems (10, 11). This level of observability is necessary for identifying performance bottlenecks, understanding system dependencies, and ensuring the reliable operation of distributed applications (9). Despite its benefits, distributed tracing introduces additional overhead, which can impact the throughput and performance of the applications being monitored (12, 13, 14, 15, 16).

Understanding the performance implications of distributed tracing is important for optimizing applications (10). Applications that break work into independently pro-

cessed tasks can be particularly sensitive to the overhead introduced by tracing (17, 18). These applications involve low-latency operations, where even small increases in latency can significantly impact overall performance. As a result, to understand the impact of distributed tracing, we need to quantify the performance overhead of distributed tracing and identify the main contributors to the performance overhead.

## 1.1 Concept of Visibility, Monitoring, and Overhead

We introduce and define three key concepts addressed throughout this thesis: visibility, monitoring, and overhead. Visibility refers to understanding a system's state and behavior, while monitoring involves collecting the data needed to achieve the understanding. Overhead is the cost in terms of the performance impact that comes with data collection. We explore each concept and the interactions between these concepts.

### 1.1.1 What is Visibility?

Visibility is the ability to understand the state and behavior of a system. Collecting data is part of gaining visibility, but gaining useful insights that can be used for diagnosing issues, optimizing performance, or making informed decisions is a challenge. Effective visibility means having the right level of detail to understand how a system behaves. It involves not only the quantity of the data but also the quality of the data and how it is analyzed and interpreted.

### 1.1.2 What is Monitoring?

Monitoring is the process of collecting, analyzing, and visualizing data, such as metrics, logs, and traces, about a system (19, 20). The data can be about various indicators of a system's behavior, such as resource usage, error counts, and request durations. Monitoring is essential to achieve visibility but does not guarantee it (19). The effectiveness of monitoring depends on selecting insightful information to monitor.

### 1.1.3 What is Overhead?

In this thesis, we refer to overhead as the additional resource consumption and performance impact introduced by monitoring tracing activities (15). Collecting detailed metrics from a system requires additional processing and storage capabilities, which

can impact the system's performance. Tracing and instrumentation can add latency, consume more resources, and increase the network traffic by exporting the trace data.

### 1.1.4 Relations Between Monitoring, Visibility, and Overhead

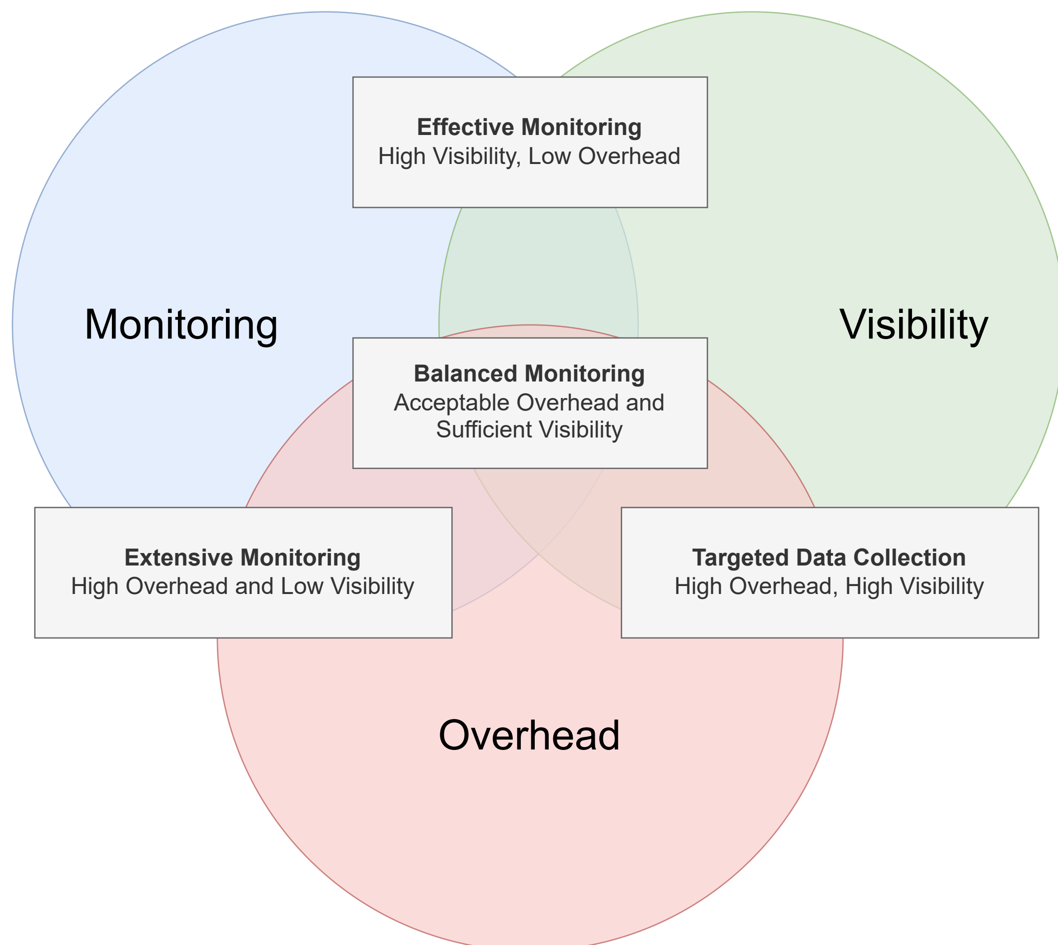


**Figure 1.1:** Illustration of the relationship between monitoring, visibility, and overhead. The intersection of the concepts represents balanced monitoring with sufficient visibility and acceptable overhead.

The goal is to balance monitoring, visibility, and overhead to achieve as much visibility as needed to understand the system while keeping the overhead as low as possible. However, it is also important to understand the relations between these concepts in

order to achieve the goal. Figure 1.1 displays some of the relations between monitoring, visibility, and overhead. Ideally, we want to have *Effective Monitoring* with high visibility and minimal overhead. However, due to the trade-off between visibility and overhead, extensive monitoring can lead to high overhead but still with limited visibility. Targeted data collection or optimizing the monitoring can reduce the amount of monitoring and increase visibility. The overhead can remain high if a large amount of data is collected. We highlight that excessive monitoring does not guarantee better visibility, and higher visibility can be achieved without increasing monitoring (21).

Some of the relations we observe between monitoring, visibility, and overhead are the following:

**Extensive Monitoring or High Overhead $\not\Rightarrow$ High Visibility** Collecting large amounts of data through monitoring or having high overhead does not always imply high visibility. The data collected must be relevant and helpful in diagnosing and understanding the system. Excessive data can create additional noise, making the effort counterproductive.

**High Visibility Without Extensive Monitoring** High visibility does not always require extensive monitoring. Using efficient sampling techniques or limiting the amount of data collected can provide sufficient information about the system while minimizing the amount of data collected. For example, in high-throughput services, the majority of interesting events are likely to occur often enough to be recorded (15).

**Reducing Monitoring While Maintaining Visibility** Reducing the scope or intensity of monitoring is possible without reducing visibility by focusing on infrequent and interesting data instead of noise (21). This allows us to maintain visibility in the system while decreasing overhead.

**Balancing Overhead and Visibility** Monitoring aims to achieve sufficient visibility to support the system's reliability. Meanwhile, we also want to keep the overhead at acceptable levels to reduce the performance impact on the service. Additional monitoring can increase the overhead, but this data might be needed for higher visibility. Therefore, a balanced trade-off point should be found between the amount of data and the performance impact it causes.

## 1.2 Problem Statement

In this thesis, we address two main problems: the trade-off between system visibility and performance and the unclear sources of overhead in the distributed tracing process.

**Performance and Visibility Trade-Off** The integration of distributed tracing introduces a challenge: a trade-off between visibility into the system and the overhead caused by tracing. While tracing significantly enhances our understanding of distributed systems by shedding light on interactions and dependencies, it also introduces additional overhead, which can negatively impact performance, leading to increased latency and reduced throughput. Therefore, it's essential to understand the extent of this overhead and its effects on the performance of various applications.

**Sources of Overhead** Another aspect of the problem is that the sources of overhead within distributed tracing process are not clearly understood. The tracing lifecycle consists of multiple stages, such as initialization, propagation, data collection, and export, each may contribute differently to performance degradation. Identifying and quantifying the most resource-intensive stages is important for effectively mitigating the performance penalties associated with distributed tracing.

## 1.3 Research Questions and Methodology

This study aims to examine the effects of distributed tracing on applications by studying different types of applications—focusing on request-based (microservices) and task-based (serverless) models—and finally identifying the sources of the overheads associated with tracing implementation. We establish three research questions and describe the methodology used to study each question. The research questions (RQ) are the following:

**RQ1. How does the implementation of distributed tracing affect the throughput and latency of request-based applications?** This research question explores the impact of distributed tracing on the performance and latency of request-based applications. Request-based applications have a client-server model, where a client sends a request to the server and the server provides a response. The goal

is to measure how tracing affects the throughput and the latency of such applications. By comparing applications with and without tracing, we aim to quantify the performance impact and trade-offs for request-based applications (22).

The methodology involves designing and setting up a controlled experiment with various web application endpoints (23, 24). We also set up the endpoints on four frameworks (Python Flask, Java Spring Boot, Go http, and JavaScript Node.js). We configure three sets of applications: non-instrumented, instrumented with OpenTelemetry, and instrumented with Elastic APM. We analyze throughput and latency to analyze and understand the tracing overhead (23, 25). The experiment is discussed in Chapter 3.

**RQ2. What are the effects of distributed tracing instrumentation on the performance of task-based applications?** This research question aims to assess the impact of distributed tracing instrumentation on the performance of task-based applications (22). These applications, which, for example, provide image processing, data processing, and object management, can break work into independently processed tasks. While distributed tracing provides insight into the operation of such applications, it may also introduce a significant performance overhead (12, 13).

We study the effect of the performance impact on task-based applications by designing an experiment that measures the latency of these applications with and without tracing instrumentation on the OpenWhisk platform (26, 27). The benchmarks are implemented in Python and Node.js to compare the impact of tracing with different technology stacks. We quantify the performance impact by comparing the latency of instrumented and non-instrumented variations of each benchmark. The experiment is described in Chapter 4.

**RQ3. What are the primary sources of overhead introduced by distributed tracing, and which of them are the main contributors to overhead?** We desire to understand which components of distributed tracing systems most significantly contribute to the overhead and how these contributions occur. This research seeks to quantify the overhead of distributed tracing processes, such as

initialization, instrumentation, and export. After categorizing and quantifying the overhead sources, we provide the main contributors of overhead in distributed tracing.

The methodology to investigate this research question consists of two steps: (1) categorizing distributed tracing processes into distinct groups and (2) measuring the performance impact caused by each group. We categorize the processes by analyzing and visualizing the profiling data. Then, we evaluate each group's overhead by profiling the applications and calculating the total time spent on each category. We discuss the experiment design, setup, and results in Chapter 5.

Figure 1.2 displays an overview of the experiments we conduct to answer the research questions RQ1, RQ2, and RQ3. It displays the technology stack of the applications and the instrumentation tools used for each experiment.

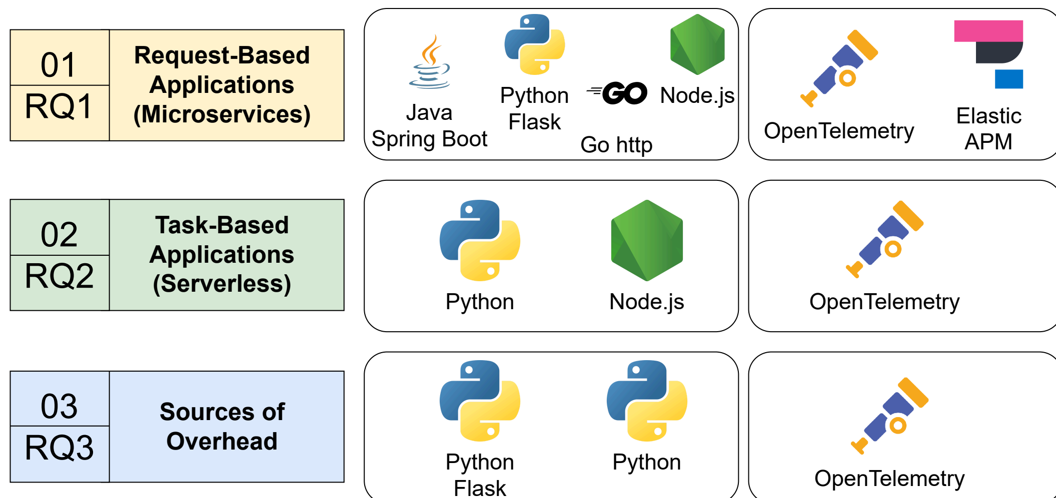


**Figure 1.2:** Overview of the three experiments conducted in the thesis. The figure highlights the technology stacks and tracing tools used in the experiments. Experiment 1 is described in Chapter 3, experiment 2 in Chapter 4, and experiment 3 in Chapter 5.

## 1.4 Main Contributions

The thesis has the following research contributions (RC):

**RC1. Developed a framework to quantify the overhead in microservice applications:** We designed an experimental framework to quantify the performance

overhead introduced by distributed tracing in microservice applications. The framework measures the overhead for two metrics: throughput and request duration. The experimental framework assessed multiple programming environments: Python Flask, Java Spring Boot, Go HTTP, and JavaScript Node.js.

**RC2. Extended an existing framework to quantify the overhead in serverless applications:** We adapted and extended an existing evaluation framework (SeBS (28)) to benchmark the serverless applications, enabling the measurement of tracing overhead in Node.js and Python-based serverless workloads. As part of the extension, we developed the missing Node.js implementations in the framework and the instrumented versions for each application. This work focused on evaluating the performance impact of tracing with the execution time of the serverless workloads.

**RC3. Quantifying the overhead in microservices and serverless applications:** We conducted experiments on request-based (microservices) and task-based (serverless) applications. This research provides a comparative insight into the effect that distributed tracing has on different types of applications and frameworks. We also examine two instrumentation tools (OpenTelemetry and Elastic APM) and how the instrumentation overhead between these tools varies. The study examines performance degradation in terms of latency and throughput.

**RC4. Categorizing and analyzing the overhead sources in distributed tracing:** We categorized the tracing function calls into three distinct groups: configuration, instrumentation, and exporting. The configuration includes initializing and configuring the tracing tool, setting up the exporter, sampling, and creating high-level metadata for the tracer. Instrumentation consists of adding the trace points to capture and enrich the tracing data. Exporting includes transmitting the trace data to an external system for storage and analysis. Furthermore, we measured the overhead of each category and found that exporting and configuration are the main contributors to the overhead. Instrumentation had a relatively low impact on the performance of the system.

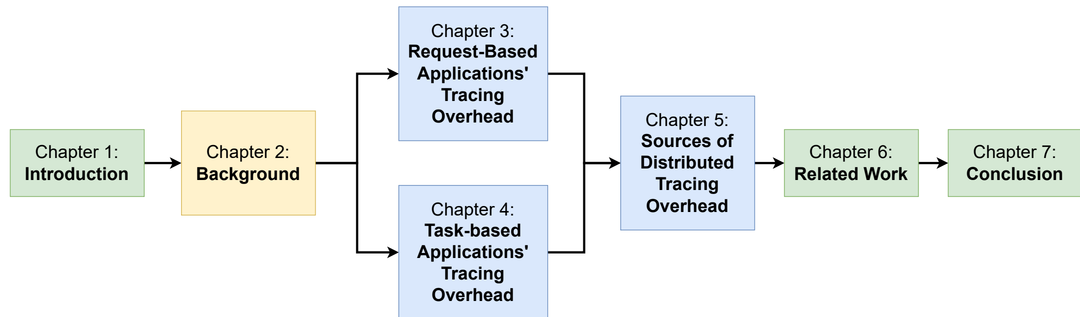


**Figure 1.3:** The thesis contents and reading structure.

## 1.5 Thesis Structure

The thesis is organized into six chapters, and the structure is shown in Figure 1.3.

The Background chapter (Chapter 2) describes the background knowledge required for the thesis. We cover the following topics: distributed tracing, instrumentation tools used in the experiments, Function as a service (FaaS) platforms, Kubernetes, and the software and hardware specifications used.

Chapter 3 details the experimental design, setup, and results for assessing tracing overhead in request-based applications. The experiment studies the performance impact of distributed tracing (instrumentation with OpenTelemetry and Elastic APM) on a selection of programming frameworks (Python Flask, Go http, Java Spring, Node.js).

Chapter 4 extends the evaluation to task-based applications, focusing on independently processed serverless tasks. The experiment compares the performance impact of Python and Node.js on different kinds of workloads.

In Chapter 5, we explore the sources of overhead introduced by distributed tracing. First, we formulate a list of categories for the tracing processes and then evaluate the overhead caused by each category's operations.

Chapter 6 gives an overview of research gaps in the current literature, provides a list of distributed tracing overheads measured in research, and briefly compares the instrumentation tools used and analyzed in the literature.

We conclude the thesis with Chapter 7 that summarizes the key insights from the research, reflects on the contributions, and answers the research questions proposed in the introduction (RQ1, RQ2, RQ3).

# 2

# Background on Distributed Tracing, Automatic Instrumentation, and Serverless Applications

## 2.1 Distributed Tracing

Distributed tracing is a technique used to monitor and track requests as they move through different services in a distributed system. Capturing a request's lifecycle provides insights into how services interact, the sequence of operations, and where time is spent within the system. This visibility offers several key benefits, including:

- **End-to-End Visibility:** Allows comprehensive monitoring in complex, distributed architectures.
- **Performance Insights:** Provides performance data such as the latency of requests and individual operations within the applications.
- **Root Cause Analysis:** Provides additional context for troubleshooting errors, making it easier to pinpoint the cause of issues.
- **Understanding Service Dependencies:** Offers a clear picture of service relationships and interactions within the architecture (9).

The following key concepts need to be understood about distributed tracing, which are frequently referenced throughout this thesis: We define the following key concepts related to distributed tracing used and mentioned in this thesis:

**Trace** A trace represents the journey of a single request as it travels through one or multiple services within a system. Trace consists of a collection of spans, representing an operation carried out by one of the services. It provides detailed information such as the sequence of the operations, time spent on each process, and any errors that occur. An example of a trace is shown in Figure 2.1.

**Span** Span represents a single operation in a request's lifecycle. This operation can be an operation such as a database query, HTTP request, or an external API call. A span records information about its start time, duration, and metadata that provides additional context about the specific request. Spans are organized hierarchically to show the relationship between different operations and how they are placed inside the trace. In Figure 2.1, each colored rectangle is a separate span.

**Sampling** Sampling is a technique used in distributed tracing to reduce the trace volume generated by the applications (12, 15, 21). Capturing every single trace in systems with high traffic can quickly cause significant performance overhead and increase storage costs (13, 15, 29). Sampling strategies, such as probabilistic sampling (randomly capturing a percentage of traces) or rate-limited sampling (limiting the number of traces collected over a specified time), help mitigate these issues. By selectively capturing traces, sampling maintains visibility while minimizing the performance impact on the system.

## 2.2 Instrumentation and Instrumentation Frameworks

Instrumentation is the process of adding code to an application to collect trace data. In this study, we use two methods to instrument and collect the tracing data in our experiments: manual instrumentation and zero-code (automatic) instrumentation. Our experiments use two instrumentation tools: OpenTelemetry and Elastic APM. We selected these tools mainly because of two reasons: (1) both support instrumentation for a wide range of languages and frameworks (manual and automatic), and (2) both offer the installation and instrumentation for free (30). In addition, Elastic APM is part

[2] `https://github.com/open-telemetry/opentelemetry-demo`

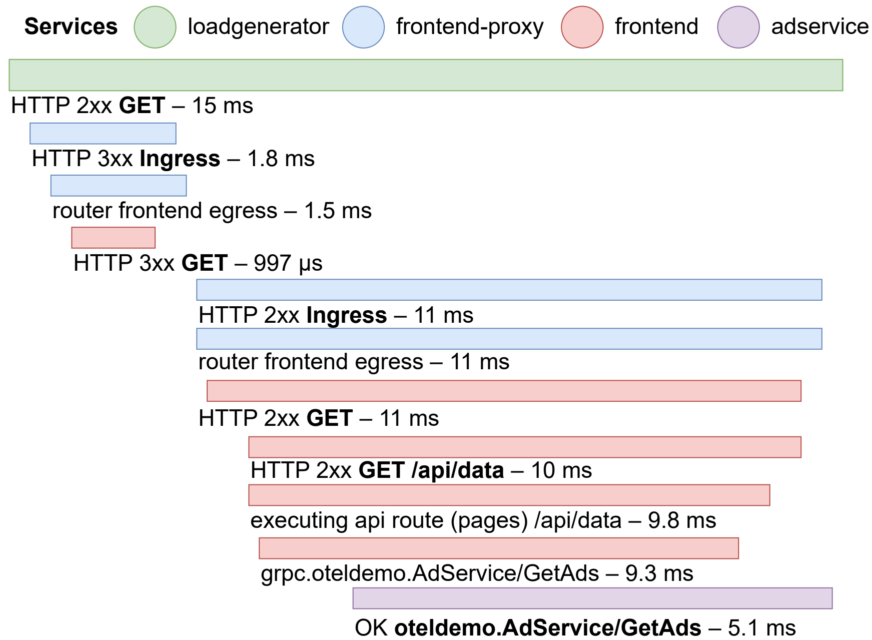


**Figure 2.1:** Example of a trace displaying a request traveling through four different services. The request is an example from the OpenTelemetry Demo system[2].

of the Elastic stack, which also contains a locally deployable observability backend to visualize traces.

**Manual Instrumentation** Manual instrumentation involves explicitly adding code to an application to capture telemetry data. Developers have fine-grained control over specifying where spans start and end, what metadata to include, and configuring trace exporters and sampling rates. While this method offers precise control over the collected data, it can be time-consuming to implement, especially in complex architectures with thousands of services.

**Zero-Code Instrumentation:** Zero-code, or automatic instrumentation, simplifies the process by automatically adding the necessary hooks to an application without modifying its code. This is usually achieved by attaching an agent or initializing libraries at runtime. Although this method reduces the effort needed to set up instrumentation, it is less flexible than manual instrumentation because developers have limited control over the details of the collected data.

### 2.2.1 OpenTelemetry

OpenTelemetry is an open-source observability framework that provides standardized methods for collecting, processing, and exporting telemetry data, including traces, metrics, and logs (31). As part of the Cloud Native Computing Foundation (CNCF), it aims to offer consistent observability across various programming languages and environments.

OpenTelemetry consists of several core components. The key components used or mentioned in the thesis are the following:

1. **API:** The OpenTelemetry API provides interfaces for adding instrumentation to an application. It defines methods for creating and managing traces, metrics, and logs, ensuring a consistent approach across different programming languages and environments.

2. **SDK:** The OpenTelemetry SDK works alongside the API to collect and process telemetry data. It offers flexibility through configurable options like samplers and processors, giving developers control over how the data is gathered, managed, and sent to different systems.

3. **Collector:** The OpenTelemetry Collector is a vendor-agnostic component that can receive the traces generated by the application, process them, and then export to a backend for storage and visualization (32). It acts as a standalone service, providing flexibility and scalability in the observability architecture. In our experiments, the Collector forwards data to Elasticsearch for storage, enabling trace visualization in Kibana.

### 2.2.2 Elastic APM

Elastic APM is an application performance monitoring system designed to provide detailed performance data and end-to-end distributed tracing for applications (33). It is part of the Elastic Stack, an observability framework intended for collecting, analyzing and visualizing logs, metrics, and traces.

In this thesis, we use the following Elastic Stack components to collect and store the traces for offloading, storage, and visualization:

- **Elastic APM Server:** The APM Server functions similarly to the OpenTelemetry Collector, serving as the primary ingestion point for trace data collected from Elastic APM agents or manually instrumented applications. It processes and forwards the data to Elasticsearch for storage.

- **Elasticsearch:** Elasticsearch is responsible for storage, searching, and analytics in the Elastic Stack (34). It indexes and stores the incoming trace data and provides fast querying and retrieval of traces for analysis.

- **Kibana:** Kibana provides the visualization interface for the Elastic Stack, enabling users to explore and analyze telemetry data. It offers a detailed view of individual traces and overall application performance. This experiment uses Kibana to validate the successful collection and visualization of trace data from both OpenTelemetry and Elastic APM instrumented applications.

## 2.3 Function as a Service Platform

Function as a Service (FaaS) is a computing model used for deploying individual functions without managing host machines for the workloads (35, 36). The infrastructure manages the scaling of the functions in response to incoming traffic, which provides more efficient resource utilization. With the FaaS model, the code is packed into stateless functions that are triggered by events, such as HTTP requests or changes in a data store.

Several cloud platforms offer FaaS, such as AWS Lambda[3], Google Cloud Functions[4], and Azure Functions[5]. While these managed services do not require any underlying infrastructure management, some FaaS platforms can also be deployed in a self-managed environment using Kubernetes.

For the experiments discussed in Chapter 4, we considered three platforms that are deployable on Kubernetes: OpenWhisk[6], OpenFaas[7], and Knative[8]. We ultimately chose OpenWhisk due to its compatibility with our selected experimental framework.

[3] https://aws.amazon.com/lambda/
[4] https://cloud.google.com/functions?hl=en
[5] https://azure.microsoft.com/en-us/products/functions
[6] https://openwhisk.apache.org/
[7] https://www.openfaas.com/
[8] https://knative.dev/

Below, we briefly overview each platform and the considerations that influenced our choice.

**OpenWhisk** Apache OpenWhisk is an open-source serverless platform that can be deployed on Kubernetes or Docker Compose environments (35). It is well-documented, with comprehensive installation guides and a large contributor community, making it a widely adopted option for serverless computing. According to GitHub metrics, OpenWhisk is the second most popular among the platforms considered[9].

The primary reason for selecting OpenWhisk was its built-in support within the Serverless Benchmark Suite (SeBS) framework (28, 37), which we used for our experiments. This compatibility eliminated the need for additional integration work, streamlining the setup process and allowing us to focus on the experiment.

**OpenFaaS** OpenFaaS is another open-source serverless platform designed to be run on Kubernetes. It supports various programming languages through templates, making it flexible for different use cases. At the time of this research, OpenFaaS was the most popular of the three platforms, as measured by GitHub repository activity[10]. It also offers a built-in user interface, a command-line tool, and an easy installation process.

While OpenFaaS presented a viable option for our serverless experiments, we chose not to proceed with it due to its lack of native support in the SeBS framework. Integrating OpenFaaS into our experimental setup would have required significant additional development effort.

**Knative** Knative is a Kubernetes-based serverless platform designed to simplify the deployment and management of serverless applications (36). It features built-in traffic splitting, allowing users to direct traffic across multiple versions of a deployed application. However, our initial and brief research showed that Knative seemed to be the most difficult to set up compared to the other platforms considered. Additionally, the SeBS framework did not natively support Knative, which would have increased the complexity of our experimental setup. As a result, we decided against using Knative for our experiments.

[9] `https://github.com/apache/openwhisk`
[10] `https://github.com/openfaas/faas`

### 2.3.1 Introduction to Kubernetes

Kubernetes is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications (38). Originally developed by Google, it is now maintained as a project under the Cloud Native Computing Foundation[11] (CNCF). Kubernetes enables applications to run reliably at scale across both on-premise and cloud environments. It follows a declarative model, where users specify the desired state of their applications, and Kubernetes continuously works to achieve and maintain that state (38). The platform simplifies the management of distributed systems by handling essential tasks such as load balancing, auto-scaling, and self-healing.

The following key concepts are essential to understanding how Kubernetes operates and are used throughout this thesis:

**Cluster** A Kubernetes cluster, consisting of a group of machines (nodes) that run containerized applications, forms the platform's foundational structure. A cluster is divided into two main components: the control plane and one or more worker nodes. The control plane is responsible for managing the overall state and operation of the cluster while the worker nodes execute the application workloads.

**Node** A node is a single machine within a Kubernetes cluster, which can be virtual or physical. It serves as a worker that runs containerized applications. There are two types of nodes: control plane nodes, which manage the cluster, and worker nodes, which handle the execution of application containers.

**Pod** A pod is the smallest deployable unit in the Kubernetes ecosystem. It consists of a group of one or more tightly coupled containers. The network and storage are shared between the containers inside a single pod.

## 2.4 Experiment Environment

This section outlines the software and hardware configurations used for the experiments presented in Chapters3, 4, and 5. We also describe the hardware specifications of the

[11] `https://www.cncf.io/`

machines used to host the experiments.

### 2.4.1 Software Versions

Table 2.1 describes the software components and their respective versions used in all the experiments designed in this thesis. The *General* section includes software used in multiple experiments, while the experiment-specific sections detail software exclusive to individual experiments.

### 2.4.2 Hardware Environment

The experiments are conducted on a single node of the AtLarge Research[12] group's cluster. We use the research group's node to host the experiment for two reasons: a consistent and stable experiment environment and cost-efficiency. An alternative would be to run the experiments in a cloud environment, which provides a more replicable environment but is less cost-efficient for this thesis.

The hardware specifications are the following:

- **CPU:** Intel® Xeon® Silver 4416+ Processor[13] at 2.00 GHz, providing 20 cores and 40 threads.
- **Memory:** 256.0GB of DDR4 RAM, consisting of four RAM units configured to operate at 1200 MHz (2400 MT/s).
- **Storage:** 1.5 TB of SSD storage.
- **Operating System:** Ubuntu 22.04.4 LTS.

[12] https://atlarge-research.com/

[13] `https://www.intel.com/content/www/us/en/products/sku/232378/intel-xeon-silver-4416-processor-37-5m-cache-2-00-ghz/specifications.html`

**Table 2.1:** Software components and their versions used throughout the experiments.

| **Software** | **Version** |
| --- | --- |
| **General** | |
| Python | 3.9 |
| Flask | 3.0.3 |
| OpenTelemetry Python | 1.25.0 |
| Elastic APM Flask Agent | 6.22.0 |
| opentelemetry-instrumentation-flask | 0.46b0 |
| opentelemetry-instrumentation-sqlalchemy | 0.46b0 |
| Node.js | 18 |
| @opentelemetry/api | 1.9.0 |
| @opentelemetry/auto-instrumentations-node | 0.47.0 |
| JavaScript OpenTelemetry | 1.25.1 |
| Docker | 27.2.1 |
| Docker Compose | 2.29.2 |
| OpenTelemetry Collector | 0.100.0 |
| Elastic APM Server | 7.17.20 |
| **Experiment 1 (Request-Based)** | |
| Java | 17 |
| OpenTelemetry Java Instrumentation | 2.3.0 |
| Elastic APM Agent Java | 1.49.0 |
| Go | 1.21 |
| OpenTelemetry Go | 1.25.0 |
| Elastic Go apmhttp package | 1.15.0 |
| PostgreSQL | 16.2 |
| k6 | 0.50.0 |
| **Experiment 2 (Task-Based)** | |
| Kubernetes | 1.20 |
| Kind | 0.20.0 |
| OpenWhisk | ef725a6 |
| MinIO | RELEASE.2024-09-13T20-26-02Z |
| **Experiment 3 (Sources of Overhead)** | |
| flameprof | 0.4 |
| gprof2dot | 2024.06.06 |

# 3

# Evaluation of Request-Based Applications' Tracing Overhead: Experimental Design and Results

In this chapter, we investigate the impact of distributed tracing on the performance of request-based (microservice) applications. We aim to answer the research question RQ1 established in Section 1.3: How does distributed tracing affect the performance and latency of request-based applications? Section 3.1 outlines the design of the experiments, the research questions, and the hypotheses. Section 3.2 defines the metrics utilized in the experiments and details the data collection methods. Section 3.3 describes the configurations for the web applications, tracing tools, and supporting services. Section 3.4 specifies the parameters used in the experiments. Section 3.5 describes the deployment strategy. Finally, Section 3.6 presents the results of the experiments analyzing the throughput and latency impacts of distributed tracing.

## 3.1 Experiment Design

This section outlines the overall design of the experiment, including research questions, hypotheses, and mentions the metrics used to assess the impact of distributed tracing on microservices. We describe our experiments in Sections 3.1.5 and 3.1.6. We give an overview of the experiments in Table 3.1.

**Table 3.1:** The evaluation of request-based applications' tracing overhead experiment design overview.

| Experiment | Group | Hypotheses | Metrics |
|---|---|---|---|
| §3.1.5 Throughput Comparison of Instrumented and Non-instrumented Applications | E3.1 | H3.1, H3.3, H3.4 | CPU Utilization, Requests per second |
| §3.1.6 Latency Comparison of Instrumented and Non-instrumented Applications | E3.2 | H3.2, H3.3, H3.4 | Request duration, latency distribution, total request count |

### 3.1.1 Motivation

Distributed tracing is critical for monitoring and debugging microservices. However, it can introduce latency and reduce throughput (12, 15, 39). Understanding these impacts is crucial for maintaining high application performance. This research aims to quantify the overhead across different frameworks (Python Flask, Java Spring, Go http, and Node.js) and tracing tools (OpenTelemetry and Elastic APM), providing insights into balancing the benefits of tracing with its performance costs.

### 3.1.2 Research Questions

To organize the experiments, we categorize them into two groups based on their objectives:

**E3.1. What is the impact of distributed tracing on the throughput of request-based applications?** This group investigates how distributed tracing affects request-based applications' overall performance and throughput. Metrics such as CPU utilization and requests per second (RPS) are analyzed to determine how tracing influences the system's ability to handle load. We discuss the evaluation method of this group in Section 3.1.5.

**E3.2. What is the impact of integrating distributed tracing on the latency of request-based applications?**

This group assesses how the implementation of distributed tracing affects the

latency of microservices built with different frameworks. Key metrics include request duration, latency distribution, and total requests received. We examine the assessment of this question in Section 3.1.6.

### 3.1.3 Hypotheses

The following hypotheses guide the experiments:

**H3.1. Integrating distributed tracing will introduce measurable performance overhead, leading to decreased requests per second and increased CPU utilization.**

Distributed tracing involves capturing detailed telemetry data, which adds extra processing and data handling tasks. As a result, the CPU utilization increases as more resources are required for the tracing operations (12). We expect the throughput to decrease significantly after implementing distributed tracing.

**H3.2. Integrating distributed tracing will result in noticeable performance degradation in latency.**

The process of capturing and logging trace data introduces additional steps in the request-handling workflow, leading to increased latency (15). The instrumentation of code to capture trace data adds extra operations, such as exporting and processing. This hypothesis aims to investigate how implementing distributed tracing impacts the performance of applications.

**H3.3. The performance overhead introduced by tracing will vary across different frameworks (Python Flask, Go http, Java Spring, Node.js).**

Each framework has unique runtime characteristics, memory management strategies, and performance profiles. The frameworks handle concurrency, garbage collection, and system calls differently, which can influence how distributed tracing impacts their performance. This hypothesis explores if the overhead of distributed tracing is different for various frameworks.

**H3.4. The performance impact of distributed tracing will differ between OpenTelemetry and Elastic APM, with one being more efficient.**

Previous studies show a difference in performance impact between distributed tracing tools (40, 41). By comparing the performance metrics (request duration

and RPS) between the two tracing tools, the study explores whether there is a noticeable difference between the performance of OpenTelemetry and Elastic APM. There is little research comparing these specific tools, so we do not have the evidence to predict which tool is more effective.

### 3.1.4 Implementation of Distributed Tracing Instrumentation With OpenTelemetry and Elastic APM

The experiment uses OpenTelemetry and Elastic APM for distributed tracing instrumentation across all frameworks. Both tools offer automatic instrumentation, simplifying integration and ensuring consistent trace data collection.

For OpenTelemetry, the automatic instrumentation involves adding OpenTelemetry SDK, agent, or libraries to the application. The trace data is exported to the OpenTelemetry Collector, which processes and forwards it to the Elastic APM Server to be further forwarded to Elasticsearch and Kibana for visualization.

For Elastic APM, the Elastic APM Agent or SDK is integrated into the application. The Agent and SDK automatically capture the trace data and send the trace data to the Elastic APM Server. We validate the tracing workflow for both instrumentation methods by visualizing the traces in Kibana (42).

### 3.1.5 Experiment Design for Throughput Comparison of Instrumented and Non-instrumented Applications

This experiment evaluates the performance impact of distributed tracing by measuring Requests Per Second (RPS) under realistic conditions simulating peak traffic. We establish the impact of instrumentation on the applications by determining the maximum RPS each application can handle while maintaining an average of 75% CPU utilization. With a higher target value for CPU utilization, the experiment can yield inconsistent and unreliable results if the application is overloaded. However, when the applications are underutilized, the performance data can be unrepresentative. The hypotheses H3.1, H3.3, and H3.4 are tested in this experiment. We use the RPS values obtained in this evaluation in experiment E3.2 to evaluate the latency under realistic conditions.

The workflow of the RPS calibration is shown in Figure 3.1. The load test iterations are run until an average of 75% CPU utilization is achieved throughout the 60-second test iteration. We repeat this process until we obtain the target RPS for each application

and endpoint, a total of 36 times (9 applications $\times$ 4 endpoints). To ensure the stability and reliability of the results, we repeat the experiment five times. This repetition helps mitigate any variability and provides a more accurate representation of the performance metrics. In each iteration, we use warm-up and cool-down periods to obtain more accurate measurements.

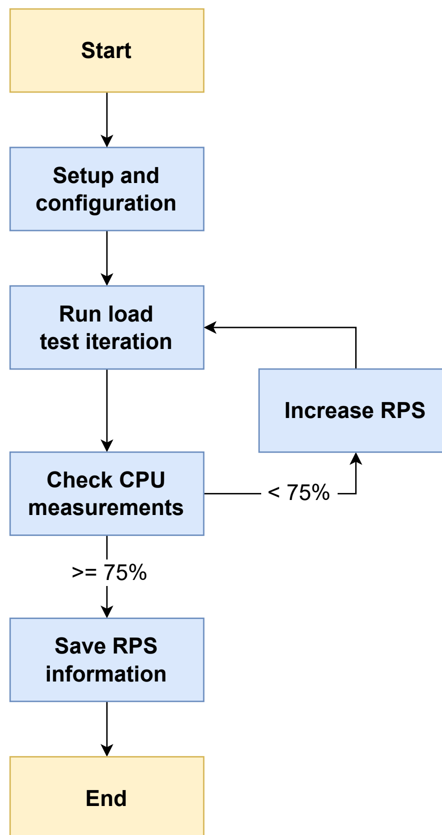


**Figure 3.1:** Request-based application requests-per-second throughput calibration workflow.

**1. Setup and Configuration** We evaluate the requests per second (RPS) for each variant of the applications (non-instrumented, OpenTelemetry-instrumented, and Elastic APM-instrumented) across all programming frameworks. The initial RPS values are set conservatively to establish a baseline for the calibration process, with identical starting points for both instrumented and non-instrumented configurations. The initial

RPS values, shown in Table 3.2, are adjusted for each framework and endpoint based on workload characteristics. Notably, lower initial values are set for the *queries* and *updates* endpoints due to their higher impact on CPU utilization.

**Table 3.2:** Initial requests per second configuration for the throughput evaluation for request-based applications.

| Framework/Endpoint | json | db | queries | updates |
|---|---|---|---|---|
| **Python Flask** | 100 | 100 | 10 | 10 |
| **Java Spring** | 100 | 100 | 25 | 25 |
| **Go http** | 100 | 100 | 25 | 25 |
| **Go http** | 100 | 100 | 25 | 25 |

**2. Incremental Load Testing** The core of the evaluation process involves incremental load testing of the applications with various RPS configurations. Each load test runs for 60 seconds, allowing sufficient time to observe the application's performance under stress. The increment values for each framework and endpoint are presented in Table 3.3.

For Python Flask, smaller increments are used across all endpoints because initial tests indicated that this framework handles significantly lower loads than the other frameworks. This finer granularity allows for more precise calibration. Similarly, the *queries* and *updates* endpoints have smaller increment values than the *json* and *db* endpoints across all frameworks, reflecting their higher sensitivity to increased load and corresponding CPU utilization.

**Table 3.3:** Requests per second incrementation values for the throughput evaluation of request-based applications.

| Framework/Endpoint | json | db | queries | updates |
|---|---|---|---|---|
| **Python Flask** | 50 | 50 | 10 | 10 |
| **Java Spring** | 100 | 100 | 25 | 25 |
| **Go http** | 100 | 100 | 25 | 25 |
| **Node.js** | 100 | 100 | 25 | 25 |

**3. Monitoring and Adjustment** Throughout each calibration iteration, the application container's CPU utilization is monitored, with nine measurements taken during the 60-second load test. Measurements start at the 10-second mark to allow the CPU metrics to stabilize. The measurements are collected every five seconds until the 50-second mark, excluding the last ten seconds, to avoid fluctuations if the load should decrease at the end of the iteration. The average CPU utilization from these measurements is calculated to determine if it reaches the target of 75%.

- If the average CPU utilization is below 75%, the RPS is incremented according to the specified step size for that framework and endpoint, and the load test is repeated.
- If the average CPU utilization meets or exceeds 75%, the RPS value achieved is recorded, and calibration for that endpoint concludes.

**4. Final Results** We save the request per second value of each calibration workflow. The calibration process is repeated five times, and each run includes iterations for each application (nine) and each endpoint (four). These calibrated RPS values serve not only as the final output of this experiment but also as a baseline for the latency evaluation experiment described in Section 3.1.6.

### 3.1.6 Experiment Design for Latency Comparison of Instrumented and Non-instrumented Request-Based Applications

The experiment aims to measure the impact of distributed tracing on latency across different frameworks using OpenTelemetry and Elastic APM to instrument the applications. Within this experiment, we check the hypotheses H3.2, H3.3 and H3.4.

Figure 3.2 shows the experiment architecture components. All components are deployed as separate containers inside Docker within the same internal Docker network. We use k6[1], an open-source load testing tool, to generate the application load.

**Setup and Configuration** We use the RPS configurations determined from the Group E3.1 phase to compare instrumented and non-instrumented applications. We take the minimal RPS value of each framework and endpoint combination and divide

[1] `https://k6.io`

the value by two. We divided the RPS value by two for this experiment to prevent excessive load on the applications and, therefore, avoid inconsistent latency values. The experiment includes running k6 load tests on 12 applications: four frameworks (Python Flask, Go http, Java Spring, Node.js) and three variations for each framework (non-instrumented, OpenTelemetry instrumented, Elastic APM instrumented). For each application, we run four tests: one per endpoint. In total, we have 48 test executions.

**Test Execution** For each combination of application and endpoint, we execute the load test with the configuration defined in Table 3.5: each test is run for 60 seconds, and the graceful stop time is 15 seconds. We collect the metrics from the results output, including various request metrics, such as total amount and duration. We also count the successful and failed requests to validate that the applications do not drop requests.

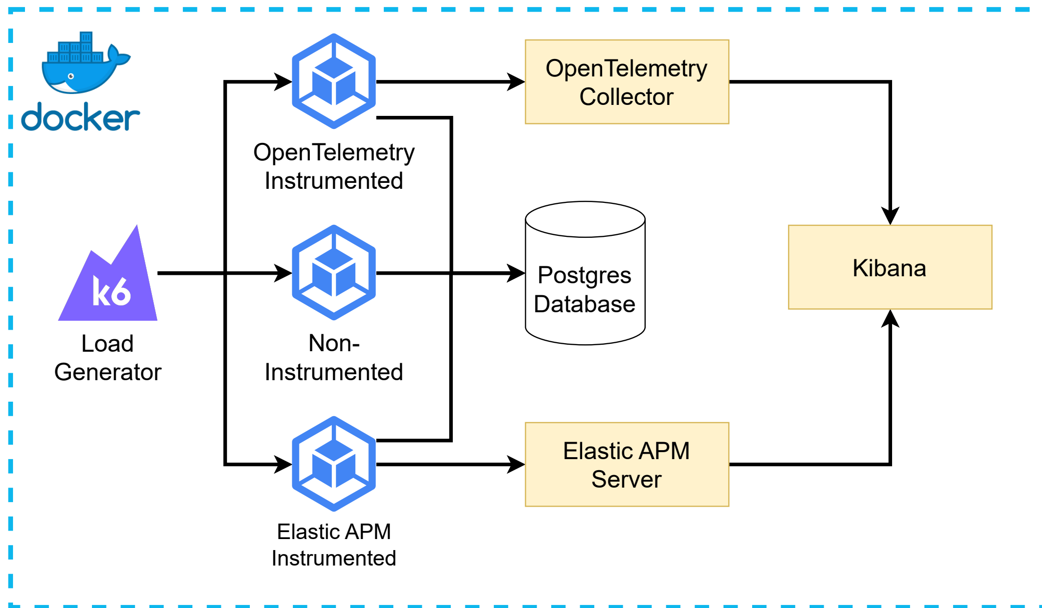


**Figure 3.2:** Architecture overview of the request-based (microservices) applications latency and throughput overhead experiment.

## 3.2 Metrics and Data Collection

This section describes the metrics used to evaluate system performance in the experiments, explaining their significance and describing the data collection methods used.

### 3.2.1 Metric Definitions

Table 3.4 lists the metrics used to assess and compare the performance of request-based applications. The two main metrics we evaluate are throughput and request duration. Throughput is evaluated in Experiment E3.1, while request duration is studied in Experiment E3.2. These metrics are selected based on methodologies commonly used in previous studies for measuring system overheads (12, 15, 18, 43, 44, 45, 46, 47).

**Table 3.4:** The list of metrics used to evaluate the request-based applications' performances in the experiments.

| Metric | Description | Unit |
|---|---|---|
| Requests per second | Rate at which HTTP requests are made per second during the test. | Requests/second |
| CPU utilization | Percentage of CPU resources used during the test. | Percentage |
| Request duration | Total time for the HTTP request. Includes the sending, waiting, and receiving time | Milliseconds |
| Latency distribution | Various latency percentiles (e.g., 50th, 75th, 95th, 99th) to understand the spread and consistency of response times. | Milliseconds |

**CPU Utilization** CPU utilization is measured during the throughput evaluation experiment. The target CPU utilization is set at 75%, following industry practices where systems are stressed without being fully saturated to avoid overwhelming the system while still revealing potential performance issues and bottlenecks (22, 48, 49). This target ensures that the applications are sufficiently loaded to highlight the performance impacts of distributed tracing.

**Requests per Second (RPS)** RPS measures the number of requests the application handles per second, serving as an indicator for the throughput. During RPS evaluation in experiment E3.1, each application's target RPS value is determined by identifying the rate at which 75% CPU utilization is achieved. Higher RPS values represent better

system performance and help identify thresholds at which the system begins to degrade under load. These RPS values are also used as baselines for comparing request duration in experiment E3.2.

**Request Duration** Request duration tracks the time an application takes to process and respond to a request, helping assess the overhead introduced by distributed tracing instrumentation. Shorter request durations indicate more efficient processing. Analyzing the request duration across different configurations (instrumented and non-instrumented) allows for quantifying the impact of tracing on latency.

**Latency Distribution** We analyze the various percentiles (e.g., 50th, 95th, and 99th) to capture the variability in the request duration. Percentiles offer us information about the consistency of the response times. For example, analyzing the 99th percentile allows us to confirm if the majority of requests are processed in an acceptable time.

### 3.2.2 Data Collection Methods

**CPU Utilization Measurement** CPU utilization is monitored by a separate thread integrated into the experimental framework. This monitoring thread uses the Docker SDK for Python[1] to periodically collect CPU metrics from the application container during the RPS evaluation phases. We use the Docker SDK so we can programmatically save and store the CPU utilization measurements within the experiment framework.

**RPS Value Collection** RPS values are determined during the evaluation phase using an iterative approach (detailed in Figure 3.1). The process involves running k6 load tests and measuring the CPU utilization at different RPS levels. If the average CPU utilization in an iteration is below 75%, the RPS is increased according to the step size specified in Table 3.3, and another iteration is performed. This process continues until an RPS level achieves at least an average of 75% CPU utilization.

**Request Duration and Total Requests** Request duration and latency metrics are extracted from the output of the k6 load testing tool. The data includes measurements at specific intervals, providing a detailed view of request duration, total requests, and

[1] `https://docker-py.readthedocs.io/en/stable/`

error rates. With this data, we can assess the effects of tracing on request processing times and identify any trends or anomalies in latency distributions.

## 3.3 Experiment Setup

This section describes the benchmark framework, the applications used, the instrumentation setup, database configuration, and load generation methods.

### 3.3.1 Benchmarking Framework

The TechEmpower framework[1] is a comprehensive benchmarking suite designed to measure the performance of web frameworks. It provides a standardized methodology for a fair comparison between different web technologies. The suite includes seven test types: JSON serialization, single and multiple database queries, server-side templating, database updates, plaintext processing, and caching mechanisms. These endpoints cover various application workflows, enabling a robust evaluation of applications under different workloads.

We develop the applications in accordance with the TechEmpower test endpoint requirements and specifications documentation (50). We selected four endpoints for our experiments, covering multiple use cases to compare the applications thoroughly under various workloads. The implemented endpoints are the following:

- **/json**: Serializes a "Hello, World!" message into a JSON object. This lightweight test measures baseline application performance without interactions with external services.
- **/db**: Fetches a single random entry from the database, evaluating the impact of tracing on database operations.
- **/queries:** Similar to the */db* endpoint, but fetches multiple rows based on a query parameter to simulate higher load conditions. In this study, ten rows are retrieved per request.
- **/updates**: Updates multiple entries in the database. For our tests, five entries are updated with each request, assessing the impact of write operations.

[1] `https://www.techempower.com/benchmarks`

### 3.3.2 Request-Based Applications

We evaluate four web applications in four frameworks: Python Flask, Java Spring, Go http, and JavaScript's Node.js. The four frameworks allow us to cover a broad spectrum of use cases in software development due to the popularity of the frameworks (51, 52, 53). The variety also ensures a comprehensive analysis of how distributed tracing performs across different programming paradigms and runtime environments.

The reasoning for selecting each framework is described below:

- **Python Flask**: We selected Flask due to its popularity, simplicity, and ease of integration with OpenTelemetry. The authors are familiar with this framework, and its minimalistic design makes it ideal for benchmarking overhead without excessive framework abstractions. We also considered the Django[1] framework, which also has similar characteristics we mention about the Flask Framework.

- **Java Spring**: Java Spring is chosen for its popularity in enterprise applications and the authors' knowledge of its configuration. Spring's support for distributed tracing tools like OpenTelemetry is well-documented, ensuring smooth integration. Using Spring instead of other Java frameworks minimizes the risk of misconfigurations during experiments, making it a reliable choice for benchmarking distributed tracing overhead.

- **Go http**: The Go http framework was selected because it uses Go's standard *http* package, eliminating the impact of third-party abstractions on our overhead measurements. Go is well-known for its high-performance capabilities and minimal runtime overhead, making it a good fit for evaluating low-latency scenarios in distributed tracing. Furthermore, not all Go packages and frameworks are supported by OpenTelemetry and Elastic APM.

- **Node.js:** Node.js is widely used in modern web development, especially in high-performance environments (54). Through automatic instrumentation, Node.js integrates seamlessly with distributed tracing tools like OpenTelemetry and Elastic APM, requiring minimal code changes.

[1] https://www.djangoproject.com/

We implement four endpoints for Python Flask and Go http applications described in Section 3.3.1. For the Java Spring and Node.js applications, we did not develop the application ourselves but instead used TechEmpower's provided implementation of Spring[1] and Node.js[2], which includes the implementation of all TechEmpower benchmark endpoints. This shows that the experiment can be replicated without requiring development effort on the application side.

### 3.3.3 Tracing Tools and Instrumentation

In this experiment, we use OpenTelemetry and Elastic APM to instrument the applications. We configure a total of three sets of applications: (1) non-instrumented, (2) OpenTelemetry instrumented, and (3) Elastic APM instrumented.

We add automatic instrumentation for OpenTelemetry and Elastic APM configurations in the following way for the frameworks:

- **Python Flask**

  (1) **OpenTelemetry:** We use the OpenTelemetry Python agent to add the instrumentation. The Python Flask application is started with the *opentelemetry-instrument* command[3].

  (2) **Elastic APM:** In the Elastic APM configuration, we use the Elastic APM agent to instrument Flask[4]. This configuration requires modifying the application by importing the Elastic APM library and wrapping the Flask application instance with the agent to enable automatic trace collection.

- **Java Spring**

  (1) **OpenTelemetry:** In Java Spring, we use the OpenTelemetry Java Agent for zero-code instrumentation [5]. The agent is pre-installed on the application image and then initialized with the *JAVA_ TOOL_ OPTIONS* environment variable.

[1] `https://github.com/TechEmpower/FrameworkBenchmarks/tree/master/frameworks/Java/spring`

[2] `https://github.com/TechEmpower/FrameworkBenchmarks/tree/master/frameworks/JavaScript/nodejs`

[3] `https://opentelemetry.io/docs/zero-code/python/`

[4] `https://www.elastic.co/guide/en/apm/agent/python/current/flask-support.html`

[5] `https://opentelemetry.io/docs/zero-code/java/agent/`

(2) **Elastic APM:** The setup for Elastic APM configuration is similar to OpenTelemetry: we install the agent on the application image and then initialize it via the *JAVA_ TOOL_ OPTIONS* environment variable[1].

- **Go http**

  (1) **OpenTelemetry:** For the Go http framework, we use the OpenTelemetry Go SDK[2] to instrument the application. We have to initialize the library and add an OpenTelemetry HTTP handler in the code to instrument the HTTP endpoints. We could not use an agent for Go since it is still a work in progress at the time of conducting the research. The manual integration in Go could introduce additional overhead compared to automatic instrumentation methods.

  (2) **Elastic APM:** The setup is similar to the OpenTelemetry instrumentation. The Elastic APM libraries for the *http* package are imported in the code similar to the OpenTelemetry instrumentation method for Go, and then we implement the APM HTTP handler to instrument the application[3].

- **Node.js**

  (1) **OpenTelemetry:** For Node.js, we use its *–require* flag to set up the automatic instrumentation via the *auto-instrumentations-node*[4] package. We set the flag through an environment variable named *NODE_ OPTIONS*. Therefore, no code changes are needed, because we add the additional environment variable to the Docker Compose file.

  (2) **Elastic APM:** The configuration of the instrumentation with Elastic APM is similar to OpenTelemetry: we use the *–require* flag to initialize the agent.

### 3.3.4 Experiment Framework's Database

In this experiment, we use PostgreSQL[5], an open-source object-relational database system. The choice is based on the widespread use and support within the TechEmpower Framework, making it a suitable choice for the experiment's architecture. Alternative

[1] `https://www.elastic.co/guide/en/apm/agent/java/current/setup-attach-api.html`

[2] `https://opentelemetry.io/docs/languages/go/instrumentation/`

[3] `https://www.elastic.co/guide/en/apm/agent/go/master/builtin-modules.html#builtin-modules-apmhttp`

[4] `https://www.npmjs.com/package/@opentelemetry/auto-instrumentations-node`

[5] `https://www.postgresql.org/`

database systems, such as MongoDB and MySQL, are also viable for the experiment (these are also supported within the TechEmpower Framework at the time of the research). We use PostgreSQL version 16, matching the version and configuration used by TechEmpower to maintain comparability. The key settings include:

- **max_connections**: Configured to accommodate many concurrent connections from the applications. Set to 2000.
- **work_mem**: Set to 64MB to handle complex queries more efficiently.
- **synchronous_commit**: Set to *off* to improve transaction throughput.

### 3.3.5 Load Generation

To generate load in the experiments, we use k6, an open-source load testing tool designed to test web applications' performance and reliability. k6 offers scripting capabilities and a variety of executors to simulate real-world traffic patterns and stress test the applications. We selected k6 because it can generate consistent and customizable load patterns and supports distributed and large-scale performance testing.

**k6 Configuration** In our experiment, we configure k6 to use scenarios and the *constant-arrival-rate*[1] executor to run the load tests. This executor allows us to maintain a consistent request rate throughout the duration of the test, ensuring that the applications are subjected to a steady load for stable results.

**Test Scenarios** We define a k6 test scenario for each endpoint, instrumentation method, and framework combination. Each scenario is configured with the duration, number of virtual users, and the cooldown period. We also add tags to each scenario to distinguish test results in the k6 test result output file. These tags include information such as the application name, endpoint, requests per second, and the programming framework. This tagging allows us to easily filter and analyze the results to compare performance across different configurations and scenarios.

[1] `https://grafana.com/docs/k6/latest/using-k6/scenarios/executors/constant-arrival-rate/`

## 3.4 Experiment Configuration

This section outlines the parameters used in the experiments, which configure the test conditions and impact the performance and outcomes. The parameters define how the applications are evaluated under load, with and without tracing instrumentation, ensuring consistency and reproducibility across tests. Table 3.5 overviews the key parameters used in the request-based application experiments.

**Table 3.5:** Experiment configuration parameters for the request-based application experiments.

| Parameter | Value |
|---|---|
| Requests Issued per Second | Configured in experiment E3.1 |
| Throughput Evaluation Duration | 60 seconds |
| Latency Load Test Scenario Duration | 60 seconds |
| Graceful Stop | 15 seconds |
| Endpoint */queries* Parameter | 10 db entries |
| Endpoint */updates* Parameter | 5 db entries |

- **Requests Issued per Second:** This parameter defines the number of requests sent to the server per second. We obtain the value for this parameter for each framework and application combination in experiment E3.1. The parameter is used in experiment E3.2 to evaluate the latency of the applications.
- **Throughput Evaluation Duration:** This is the duration for each k6 load test iteration during RPS calibration. A 60-second duration was deemed sufficient to stabilize the load and collect CPU utilization metrics, avoiding unnecessarily long iterations. We avoid longer durations due to the large number of iterations required for calibration, which would increase computational overhead without significantly benefiting the experiment's accuracy.
- **Latency Load Test Scenario Duration:** The load test duration refers to how long each K6 scenario is run. The duration affects the endurance of the tracing mechanisms and the overall system performance under prolonged load. After testing multiple durations (60, 300, and 600 seconds), we chose 60 seconds because the results were stable throughout the test duration.

- **Graceful Stop:** The graceful stop duration defines the time for k6 to finish existing requests before shutting down the system. This parameter ensures that requests are completed, preventing any abrupt termination that could skew performance results, particularly for CPU utilization measurements between load tests. A 15-second window was selected to ensure clean shutdowns after the load test.

- **Application Queries Parameter (/queries and /updates):** This parameter specifies the number of queries or updates executed on the database for each request, which are defined as queries in the TechEmpower benchmarks. The value simulates varying workloads on the applications by adjusting the number of database interactions per request. For this experiment, we used a fixed value of 10 for the */queries* endpoint and 5 for the */updates* endpoint to maintain consistency throughout the tests. These values reflect typical scenarios based on the TechEmpower benchmark documentation, which tests query counts ranging from 1 to 20 (50).

## 3.5 Experiment Deployment

This section outlines the methodology and processes used to deploy the experiment environment. We utilize a containerized approach to ensure consistent, reproducible settings across all experiment executions. The setup enhances uniformity and simplifies orchestration.

### 3.5.1 Deployment Process

The deployment environment relies on containerization to achieve consistency and repeatability across all tests. Docker and Docker Compose are used to containerize the applications, tracing infrastructure, and database, ensuring the isolation of each component and removing the possibility of external environmental interference.

- **Containers:** Each programming framework used in the experiment has a separate Dockerfile for building the necessary Docker images. The application's instrumented and non-instrumented versions use the same base Dockerfile to ensure uniformity. This guarantees that the only difference between non-instrumented

and instrumented configurations is the presence or absence of tracing instrumentation. For each application variation (non-instrumented, OpenTelemetry, and Elastic APM), the appropriate libraries or agents are added to the Docker image. We build 12 Docker images (three variations for each of the four frameworks). The database uses the official Postgres image, with an initialization script, to set up the necessary tables, users, and entries for the experiments. The tracing infrastructure is also deployed using official Docker images for the collectors and backends.

- **Service Orchestration:** We use Docker Compose to orchestrate the deployment of the containers, allowing us to manage multiple services efficiently. Each application variation has a separate Docker Compose file, which includes the necessary environment variables to configure the tracing instrumentation and tools. All applications are limited to a single CPU core, and all containers (including the database and tracing tools) are deployed on the same Docker network to ensure consistency. Using an isolated internal network prevents interference from external network traffic or processes running on the host machine, ensuring consistent results across all executions.

- **Tracing Tools:** The tracing tools—OpenTelemetry and Elastic APM—are configured directly within the Dockerfiles. For OpenTelemetry-instrumented applications, traces are exported to the OpenTelemetry collector. Applications instrumented with Elastic APM send their traces to the Elastic APM server. These collector services are deployed in separate containers, mimicking real-world environments where traces are sent to a centralized backend instead of logged locally. This allows for a more realistic simulation of distributed tracing workflows.

- **Benchmark Services**: The load testing and benchmarking services, which execute the k6 load tests and measure system performance, are also containerized. This ensures that the benchmark tools operate under the same controlled environment as the applications. The necessary test scripts and configurations are bundled into the Docker image, and the load tests are triggered automatically upon container startup.

Overall, the deployment process is designed to ensure consistency, reproducibility, and isolation across all experiments. Encapsulating each application, tracing tool, and benchmarking service within containers ensures that dependencies, configurations, and external factors remain uniform across different test runs. This approach minimizes environmental variability, allowing us to focus solely on the performance and overhead caused by the tracing instrumentation.

## 3.6 Experiment Results

This section presents the results of experiments E3.1 and E3.2, which evaluate the performance of request-based applications under different configurations. Section 3.6.1 covers the throughput analysis (E3.1), while Section 3.6.2 discusses the impact of tracing on latency (E3.2).

### 3.6.1 Throughput Analysis of Instrumented and Non-Instrumented Microservices

This experiment assesses the impact of distributed tracing on web application throughput. By comparing the throughput of non-instrumented applications with those instrumented using OpenTelemetry or Elastic APM, we explore how tracing affects performance. Additionally, we investigate (1) whether the overhead varies across different programming frameworks and (2) how the overhead differs between instrumentation tools.

The main findings in this section are:

**MF3.1** Distributed tracing reduces throughput across all frameworks, with declines ranging from 19.55% to 80.18%.

**MF3.2** Java Spring exhibits the lowest overhead with microservice applications, while Node.js shows the most significant impact on throughput.

**MF3.3** OpenTelemetry generally delivers higher throughput than Elastic APM across most frameworks.

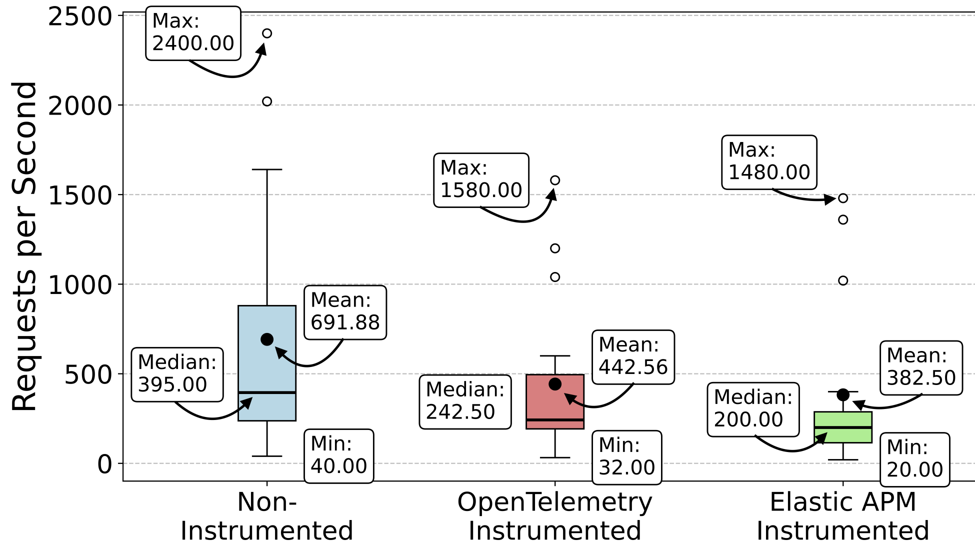


**Figure 3.3:** Aggregated throughput (requests per second) for request-based applications with different instrumentation configurations (Standard, OpenTelemetry, Elastic APM) across the evaluated programming stacks (Python Flask, Java Spring, Go http, Node.js).

#### 3.6.1.1 Throughput Comparison of Non-Instrumented and Instrumented Configurations

Figure 3.3 shows the aggregated throughput in requests per second across three configurations: non-instrumented, OpenTelemetry instrumented, and Elastic APM instrumented.

The non-instrumented configuration has the highest throughput, with a median of 395 RPS and a mean of 691.88 RPS, reaching a maximum of 2400 RPS. When tracing is enabled, the throughput declines significantly. With OpenTelemetry, the median throughput decreases by 38.6% compared to the non-instrumented median. The mean throughput falls by 36.0%, and the maximum throughput falls to 1580 RPS (34.20% decrease). The Elastic APM instrumented configuration has an even higher impact on throughput, with the median falling to 200 RPS–a 49.4% decrease. The mean throughput drops to 382.5 RPS, marking a 44.7% reduction, while the maximum RPS value is reduced to 1480 RPS.

These results indicate that distributed tracing significantly impacts the throughput of microservices, with up to 49.4% reduction in the median requests per second. We

observe that Elastic APM introduces a more significant performance overhead regarding throughput than OpenTelemetry.

#### 3.6.1.2 Throughput Comparison of Frameworks

**Table 3.6:** Throughput (requests per second) comparison of OpenTelemetry, Elastic APM, and non-instrumented request-based applications across Python Flask, Java Spring, Go http, and Node.js frameworks.

| Framework | Standard Mean RPS | OpenTel[1] Mean RPS | Elastic[2] Mean RPS | OpenTel vs Standard (%) | Elastic vs Standard (%) | OpenTel vs Elastic (%) |
|---|---|---|---|---|---|---|
| Python Flask | 167.5 | 109.0 | 90.0 | -34.93% | -46.27% | 21.11% |
| Java Spring | 952.5 | 766.25 | 737.5 | -19.55% | -22.57% | 3.90% |
| Go http | 815.0 | 498.75 | 537.5 | -38.80% | -34.05% | -7.21% |
| Node.js | 832.5 | 396.25 | 165.0 | -52.40% | -80.18% | 140.15% |

Table 3.6 compares requests per second OpenTelemetry, Elastic APM, and non-instrumented applications across different frameworks.

The table shows a trend of throughput reduction with instrumentation enabled, but the impact varies between frameworks. Java Spring exhibits the smallest overhead, with throughput reductions of 19.55% for OpenTelemetry and 22.57% for Elastic APM. On the other hand, Node.js shows the most significant decrease, with OpenTelemetry reducing throughput by 52.4% and Elastic APM by 80.18%. Python Flask and Go http fall in between, with reductions ranging from 34.05% to 46.27%, indicating relatively moderate overhead levels compared to the other frameworks under evaluation.

OpenTelemetry generally achieves higher throughput than Elastic APM for most frameworks. The performance advantage is most pronounced in Node.js (140.15% difference) and Python Flask (21.11%). With Java Spring, the OpenTelemetry's advantage is relatively low at only 3.90%. Only with the Go http framework, Elastic APM outperforms OpenTelemetry by 7.21%.

Overall, we observe that distributed tracing introduces a significant performance trade-off, with overhead up to 80.18%. Secondly, the impact of distributed tracing is considerably different depending on the framework. Finally, the instrumentation tools

[1] OpenTelemetry
[2] Elastic APM

impact the performance of frameworks differently as well, with OpenTelemetry being more efficient with the majority of the chosen frameworks, but Elastic APM being more efficient with Go http.

### 3.6.2 Latency Analysis of Instrumented and Non-Instrumented Microservices

This section analyzes the impact of distributed tracing on the latency of request-based applications. We compare various configurations, including non-instrumented, OpenTelemetry-instrumented, and Elastic APM-instrumented setups, across different programming frameworks and endpoints. The primary objective is to understand how tracing affects latency and to compare the overhead introduced by different tracing tools.

The main findings in this section are:

**MF3.4** Distributed tracing increases median request latency for all evaluated microservices, with increases ranging from 7% to 42%.

**MF3.5** OpenTelemetry and Elastic APM introduce comparable median latency overhead, but OpenTelemetry generally results in slightly higher median latency across most microservices.

**MF3.6** The performance impact on median latency varies significantly across frameworks, with increases ranging from around 10% in some cases to as much as 179% in others.

#### 3.6.2.1 Latency Overhead in Instrumented Microservices

Figure 3.4 illustrates the median latency in milliseconds for different instrumentation configurations (non-instrumented, OpenTelemetry instrumented, and Elastic APM instrumented) for four benchmark endpoints: *json*, *db*, *updates*, and *queries*. The data is aggregated over the evaluated frameworks to present a general view of latency overhead across different requests.

Figure 3.4 shows that enabling distributed tracing consistently increases latency compared to the non-instrumented configuration. The extent of overhead highly varies depending on the endpoint. For the *json* endpoint, latency rises by approximately

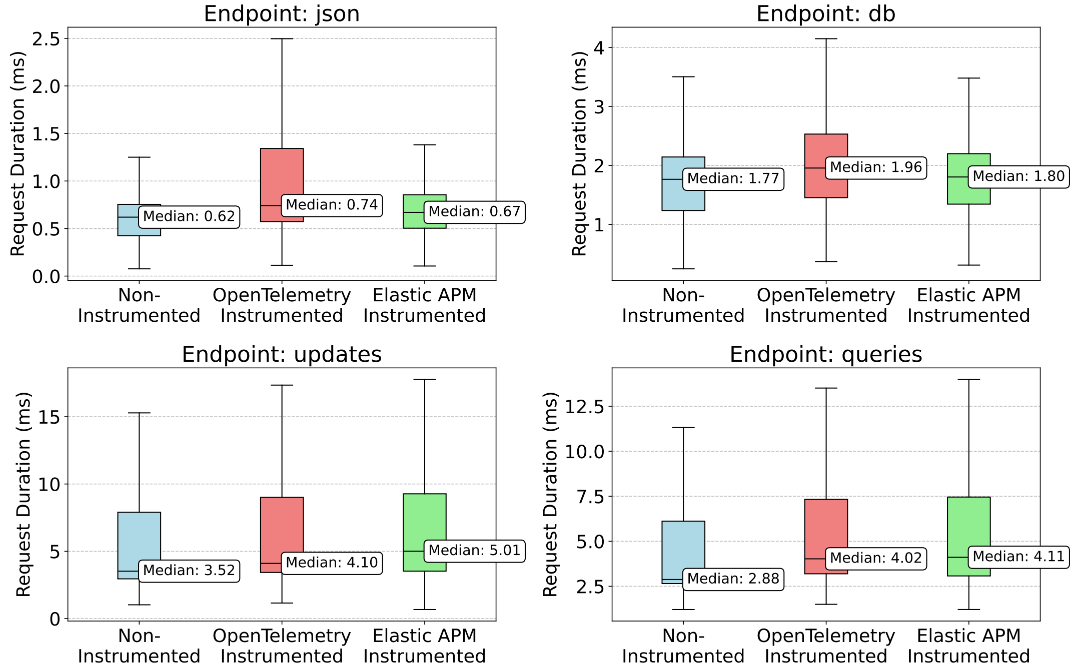


**Figure 3.4:** The median HTTP request duration in milliseconds for request-based applications across different instrumentation configurations for four benchmark endpoints. The request durations are aggregated over the programming frameworks (Python Flask, Java Spring, and Go http).

19%, from a median of 0.62 ms to 0.74 ms with OpenTelemetry, while Elastic APM introduces a lower 7% increase. The upper quartile value is also significantly higher for OpenTelemetry.

The impact is more significant for the *updates* and *queries* endpoints. The median latency for the *updates* endpoint increases by 16.5% for OpenTelemetry and approximately 42% for Elastic APM. For the *queries*, the performance overhead for both instrumentation tools is similar: 39.6% for OpenTelemetry and 42.7% for Elastic APM.

Overall, the less intensive endpoints *json* and *db* incurred (1) less overhead overall and (2) less overhead with Elastic APM compared to OpenTelemetry. However, more intensive benchmarks *updates* and *queries* exhibited less overhead with OpenTelemetry in comparison to Elastic APM.

#### 3.6.2.2 Latency Comparison of Frameworks

**Table 3.7:** Request-based applications latency comparison across different instrumentation configurations and programming languages. All values are displayed in milliseconds.

| Framework | Configuration | Median | Mean | P95 | P99 |
|---|---|---|---|---|---|
| Python Flask | Non-Instrumented | 2.10 | 8.67 | 86.87 | 90.77 |
| | OpenTelemetry | 5.58 | 11.59 | 91.37 | 93.82 |
| | Elastic APM | 2.94 | 10.21 | 100.47 | 106.48 |
| Java Spring | Non-Instrumented | 0.84 | 81.13 | 6.75 | 3447.38 |
| | OpenTelemetry | 0.93 | 112.85 | 19.06 | 4069.24 |
| | Elastic APM | 0.95 | 121.72 | 15.00 | 4799.15 |
| Go http | Non-Instrumented | 0.51 | 2.23 | 9.64 | 24.87 |
| | OpenTelemetry | 0.73 | 4.74 | 10.76 | 77.73 |
| | Elastic APM | 0.69 | 2.88 | 9.92 | 14.87 |
| Node.js | Non-Instrumented | 1.00 | 22.78 | 4.93 | 779.95 |
| | OpenTelemetry | 2.79 | 66.73 | 297.00 | 1452.00 |
| | Elastic APM | 2.71 | 7.23 | 4.82 | 89.17 |

Table 3.7 provides a comparison of request latency across different instrumentation configurations and programming frameworks.

The results confirm that distributed tracing introduces additional latency across all frameworks, but the impact varies depending on the framework and the instrumentation tool. In Python Flask, OpenTelemetry adds more latency to the median (5.58 ms, a 165.7% increase) than Elastic APM (2.94 ms, a 40% increase). However, Elastic APM shows higher latency for the 95th and 99th percentiles, indicating more variability with Elastic APM. The overhead for Java Spring is also significant–39% with OpenTelemetry and 50% with Elastic APM.

For Go http, the latency increase is relatively modest, with Elastic APM showing a lower median latency (0.69 ms, a 35.3% increase) compared to OpenTelemetry (0.73 ms, a 43.1% increase), but there is a more notable difference at the higher percentiles. The Node.js framework exhibits a significant rise in latency, with the median for both instrumentation methods (179% for OpenTelemetry and 171% for Elastic APM). However,

Elastic APM shows lower values for all other metrics compared to non-instrumented and OpenTelemetry. This could be due to the significantly lower RPS value used to run the experiment–the mean RPS for non-instrumented configuration is 832.5, and only 165 for Elastic APM.

These results demonstrate that tracing overhead is not uniform across frameworks and endpoints, with some combinations exhibiting more significant latency increases.

### 3.6.3 Results Overview

For E3.1 about the performance overhead of tracing in microservices, the results confirm the hypothesis H3.1 that integrating tracing leads to measurable performance overhead, leading to reduced throughput. The main finding **MF3.1** shows that throughput ranges from 19.55% up to 80.18% across different frameworks. Additionally, the main finding **MF3.2** confirms that the impact is not uniform–Java Spring exhibits the lowest overhead in terms of throughput, while Node.js experiences the most significant decline. This supports our hypothesis H3.3 that performance impact introduced by tracing varies between different frameworks. We also observe that OpenTelemetry impacts the throughput less than Elastic APM, upholding the hypothesis H3.4 in regard to throughput.

Regarding E3.2 on the impact of distributed tracing on the latency of request-based applications, the results confirm the hypothesis (H3.2), showing noticeable performance degradation in latency. **MF3.4** outlines that the median request latency increased for all endpoints, ranging from 7% to 42%. Main finding **MF3.6** also indicates that some frameworks experienced up to a 179% increase in median latency. Although, in other cases, the overhead was relatively low at 10%. We also examine the overhead that OpenTelemetry and Elastic APM introduced, which varied by metric. Both tools had similar impacts on median latency, with OpenTelemetry often slightly higher. However, at the 95th and 99th percentiles, differences were more noticeable.

Overall, the experiments support all hypotheses, showing that tracing introduces significant overhead for both throughput and request duration (H3.1, H3.2), with varying effects across frameworks and different tracing tools H3.3, H3.4.

# 4

# Evaluation of Serverless Applications' Tracing Overhead: Experimental Design and Results

In this chapter, we study the performance overhead of tracing in task-based applications. Section 4.1 describes the experiment design and details the objectives and hypotheses. Section 4.2 explains the metrics and data collection methods used in the experiment. Section 4.3 covers the experiment setup, including the experiment framework, application configuration, and the serverless platform used. Section 4.4 describes the configurations of the benchmark applications and the instrumentation process. Section 4.5 explains the deployment process of the infrastructure, applications, tracing tools, and the serverless platform. Finally, section 4.6 presents the experiment results, providing an analysis of the performance metrics, comparisons between configurations, and insights into the implications of our findings.

## 4.1 Experiment Design

This experiment evaluates the performance overhead introduced by distributed tracing in serverless task-based applications. We examine how tracing impacts application latency and compare these effects across different programming frameworks. The study aims to provide insights into the trade-offs involved in implementing tracing in serverless applications.

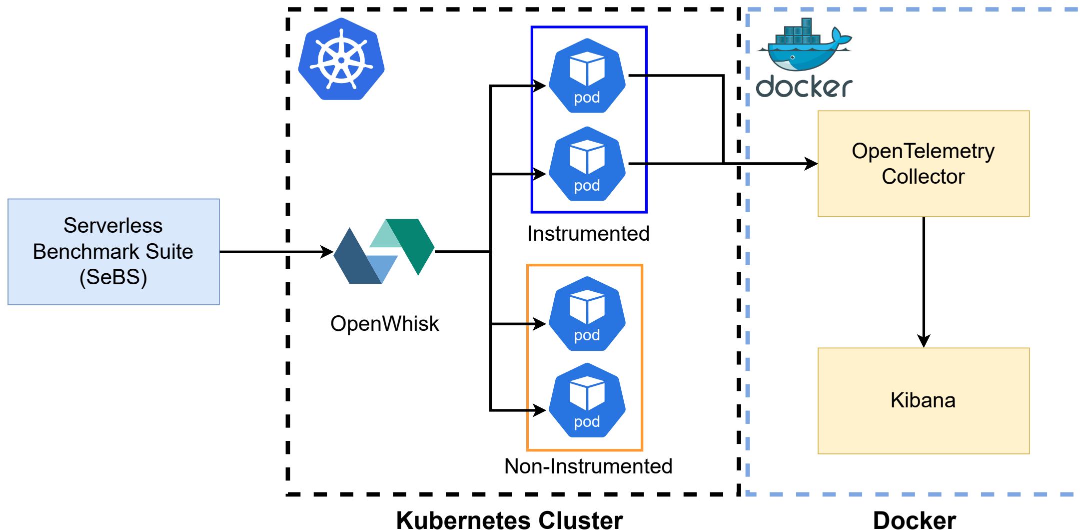


**Figure 4.1:** Task-based application experiment architecture diagram. Kubernetes hosts the serverless platform OpenWhisk, which deploys the benchmark applications, and the SeBS invokes the benchmarks. The OpenTelemetry Collector and Kibana are containerized in Docker.

### 4.1.1 Motivation

Serverless computing enables the execution of task-based applications without managing the underlying infrastructure, making it an appealing model for scalable and flexible deployments. However, the performance impact of distributed tracing in such environments remains a critical concern (13, 15, 26). Understanding how tracing affects performance is essential to optimize serverless application execution and ensure that the benefits of tracing do not come at the cost of excessive latency.

### 4.1.2 Research Questions

This experiment aims to answer the following research questions about task-based applications:

**RQ4.1. What is the performance overhead of tracing in task-based applications?** This question seeks to quantify the latency increase when tracing is enabled in task-based applications. We aim to identify the overhead introduced and better understand the trade-offs involved in adopting tracing.

**RQ4.2. How does the performance overhead of tracing in task-based applications differ between different programming frameworks?** This question explores whether the impact of tracing on performance varies across different programming frameworks. Specifically, we compare the overhead in Python and Node.js applications, helping determine whether tracing affects specific frameworks more than others.

**RQ4.3. How does the performance overhead of distributed tracing vary across different workloads?** This question examines the variability in performance overhead introduced by tracing when applied to different types of workloads. By analyzing task-based applications with different performance characteristics, we aim to identify patterns in tracing overhead and provide guidelines for its use across various application types.

### 4.1.3 Hypotheses

We propose the following hypotheses to answer the proposed research questions:

**H4.1. Enabling tracing in task-based applications will result in a measurable increase in latency.** This hypothesis suggests that the integration of tracing into task-based applications will introduce additional latency due to the extra processing required to capture and export log trace data (12, 14, 15, 16). This hypothesis addresses the research question RQ4.1.

**H4.2. Performance overhead of distributed tracing considerably differs between Python and Node.js.** The hypothesis suggests that the impact of distributed tracing varies between Python and Node.js due to their differences in runtime environments and performance (39). This hypothesis aligns with research question RQ4.2

**H4.3. Distributed tracing adds significant overhead for low-latency applications and insignificant overhead for high-latency applications.** For low-latency applications, minimal tracing overhead can significantly impact performance, leading to increased runtime. In contrast, high-latency applications may see negligible additional overhead because their baseline execution times are already large (12, 15). This hypothesis addresses research question RQ4.3.

### 4.1.4 Latency Comparison of Serverless Task-Based Applications

This experiment compares the latency of serverless task-based applications under different configurations and programming frameworks. We aim to answer research questions RQ4.1 and RQ4.2.

**Test Scenarios** We design a series of test scenarios across multiple configurations and environments to evaluate the performance overhead of tracing in task-based applications. The experiments are run on two programming frameworks, Python and Node.js, utilizing five benchmark applications. Each application is executed in two configurations: instrumented (with tracing enabled) and non-instrumented (without tracing). The setup allows us to analyze the impact of tracing across different applications and frameworks, resulting in 20 individual benchmark executions (10 per framework: 5 instrumented and five non-instrumented).

**Workloads** The benchmark applications represent a range of task-based workloads, including low and high-latency tasks. We aim to isolate the overhead introduced by tracing and assess how it varies across different frameworks and workloads.

**Objectives** Key aspects analyzed in these test scenarios include:

- **Performance Impact:** Assessing the overall performance impact of tracing on task-based applications.
- **Framework Comparison:** Evaluating differences in tracing overhead between Python and Node.js.
- **Workload Variation:** Examining how performance overhead varies across task-based workloads.

### 4.1.5 Serverless Benchmark Suite (SeBS)

The Serverless Benchmark Suite (SeBS) is a framework designed to evaluate the performance and cost-efficiency of serverless platforms and functions (28, 37). SeBS facilitates the deployment, execution, and measurement of serverless functions across multiple cloud platforms, including AWS Lambda, Azure Functions, Google Cloud Functions,

and Apache OpenWhisk. The framework provides various benchmark applications that reflect typical serverless workloads, such as web applications, multimedia processing, data compression, and scientific computations.

**Methodology** The SeBS framework consists of the following steps for benchmarking serverless functions:

1. **Configuration:** The benchmark, deployment, and experiment settings are configured using JSON files. The configuration includes details such as the benchmark application name, input size, number of repetitions, and memory configurations.

2. **Deployment:** SeBS automates the deployment of serverless functions to the selected serverless platform. It handles packaging the function code, deployment on the platform, and setting up necessary resources such as storage and triggers.

3. **Execution:** Benchmarks are executed according to the configured parameters, with support for both cold and warm starts (except for OpenWhisk). Several invocation patterns—such as burst, sequential, and concurrent—are available to evaluate performance under various conditions.

4. **Data Collection and Analysis:** During execution, SeBS collects detailed performance metrics stored in JSON files for further analysis. These metrics provide insights into client-side and server-side timings and statistical summaries for each experiment.

By leveraging SeBS, we ensure consistent and repeatable experiments, enabling a systematic analysis of tracing's impact on task-based applications.

## 4.2 Metrics and Data Collection

In this section, we outline the key metrics used to evaluate the performance of the benchmark applications, as well as the methods employed for data collection. These metrics are crucial for addressing the research questions and evaluating the hypotheses formulated in the experiment.

### 4.2.1 Metric Description

To assess the performance of the benchmark applications, we rely on the following metrics provided by the Serverless Benchmark Suite (SeBS) framework:

1. **Benchmark Time:** Benchmark time refers to the total time taken for the function execution on the serverless platform. This metric measures the time spent executing the task and does not include platform overhead or network latency (28). The metric reflects the raw execution performance of the function itself.

2. **Client Time:** This metric measures the total time taken from the client initiating the request to receiving the response, encapsulating the end-to-end latency experienced by the user (28). The Client Time consists of several measurements: 1) Benchmark Time, which is the execution time on the serverless platform; 2) Provider Time, which consists of serverless platform measurements, including the platform's overheads; 3) latency of scheduling, network, and the overhead of the SeBS wrapper. Client time represents the total experience from the user's perspective.

While we initially considered measuring actual memory consumption and provider time, these were excluded due to limitations in the OpenWhisk platform, which does not expose memory usage metrics. Additionally, the SeBS framework does not provide provider-specific measurements for the OpenWhisk environment.

### 4.2.2 Data Collection Methods

The SeBS framework records each benchmark execution and stores the results in JSON files. Each execution record includes various details, such as execution times, configuration settings (e.g., cold vs. warm starts), function outputs, and error statuses. In cases where the serverless platform supports it, additional data like cost and memory usage are also collected. Appendix B contains an example of a typical execution record.

The SeBS framework gathers the metrics through a custom wrapper that runs the experiment and collects benchmark and client time measurements at key points in the execution process. Additionally, SeBS retrieves data directly from the serverless platform when available, ensuring that platform-specific metrics are captured and integrated into the analysis.

## 4.3 Experiment Setup

This section describes the environment setup used for conducting the task-based application tracing overhead experiment. It describes the configuration of the Kubernetes cluster, object storage, serverless platform, and benchmark applications. The setup ensures a robust, efficient, reproducible environment for running the benchmarks and collecting performance data.

### 4.3.1 Environment

The experiment is conducted within a Kubernetes cluster deployed on a single virtual machine. The virtual machine hardware specifications are described in Section 2.4.2.

#### 4.3.1.1 Kubernetes

The Kubernetes cluster is configured using Kind[1], consisting of three nodes: one control plane node and two worker nodes. The worker nodes host the OpenWhisk services, enabling the concurrent execution of serverless functions. Setting up the experiment on Kubernetes allows us to ensure a consistent, replicable, and reliable environment.

#### 4.3.1.2 OpenWhisk – Serverless Platform

Apache OpenWhisk[2] is deployed on the Kubernetes cluster to manage and execute the serverless benchmark applications. OpenWhisk supports various runtimes, including Python, JavaScript, and Java, allowing us to deploy all the benchmarks used in this experiment. The Kubernetes worker nodes host the various OpenWhisk components and other required services, including:

- **Nginx:** Acts as an HTTP server and reverse proxy.
- **Kafka:** Provides a publish-subscribe messaging system.
- **CouchDB:** Serves as a NoSQL database for metadata storage.

OpenWhisk's architecture includes two key services:

[1] `https://kind.sigs.k8s.io/`
[2] `https://openwhisk.apache.org/`

- **Controller:** Manages the lifecycle of actions and triggers. It handles incoming requests and translates these requests into invocations.
- **Invoker:** Executes the actions by running them in isolated environments. Each action runs inside a pod within the Kubernetes cluster, allowing for efficient scaling and resource management (55).

### 4.3.2 MinIO Object Storage

MinIO[1] is used as the object storage system for the benchmark applications in this experiment. MinIO is an open-source distributed object storage server for data storage and retrieval. The benchmark applications utilize MinIO to download and upload files.

### 4.3.3 Benchmark Applications

We use five benchmark applications from the Serverless Benchmark Suite (SeBS), each selected to represent different workload types to assess the performance impact of tracing comprehensively. The benchmarks include web application tasks, multimedia processing, and a scientific computation task. This diversity helps to evaluate tracing overhead across a wide range of scenarios.

All applications are implemented in both Python and Node.js. For benchmarks without a native Node.js version, we develop the Node.js implementations based on the Python version and existing code examples. Each application has both instrumented (tracing enabled) and non-instrumented versions, resulting in 20 different configurations (10 per programming language).

The selected benchmarks are:

1. **Dynamic HTML:** Generates dynamic HTML content from a template. This benchmark, categorized under web applications, helps assess the impact of tracing on tasks typically found in web development.
2. **Uploader:** Downloads a file from a provided URL and uploads it to a specified bucket. This benchmark evaluates the overhead introduced by tracing with data transfer tasks.

[1] `https://min.io`

3. **Thumbnailer:** Resizes an image to generate a thumbnail. The application downloads the image from an object storage bucket, processes it, and uploads the resized image back to the bucket. This benchmark represents multimedia workloads, which involve both computation and I/O operations.

4. **Video Processing:** Adds a watermark to a video and generates a GIF. This compute-intensive benchmark evaluates the effect of tracing on video processing tasks that require substantial processing power and involve multiple I/O operations.

5. **Graph PageRank:** Executes the PageRank algorithm to rank nodes in a graph based on their importance (56). The Python implementation uses the igraph library[1], while the Node.js implementation uses the ngraph library[2]. This benchmark measures tracing's impact on computationally intensive tasks.

By evaluating these benchmarks, we can analyze how tracing affects various types of serverless tasks, providing insights into its suitability for different applications.

## 4.4 Experiment Configuration

This section describes the specific configurations and parameters used in the experiment. These configurations ensure a controlled and consistent environment, enabling accurate measurement of the performance overhead introduced by tracing. We also describe the instrumentation process with OpenTelemetry, detailing the levels of tracing applied and the steps involved in integrating tracing across the benchmark applications.

### 4.4.1 Tracing Instrumentation Levels

We implement two levels of tracing instrumentation to assess the impact of tracing on performance. The tracing instrumentaiton levels applied are:

- **No Tracing:** This level involves running the benchmark applications without any tracing instrumentation, serving as a baseline for evaluating the performance impact of adding tracing.

[1] `https://python.igraph.org/`
[2] `https://www.npmjs.com/package/ngraph.graph`

- **Full Tracing:** This level includes comprehensive instrumentation, capturing detailed spans enriched with child spans and attributes throughout the applications. The sampling rate is set to 100% to capture all the traces. Evaluating these benchmarks provides insights into the extent of the overhead tracing introduces.

### 4.4.2 Instrumentation with OpenTelemetry

We integrate OpenTelemetry into Python and Node.js applications to trace the benchmark applications. We configure manual instrumentation for all applications with extensive instrumentation. Overall, the integration consists of three steps: (1) configuring OpenTelemetry, (2) manually instrumenting the application, and (3) exporting the trace. We explain the details of each step below. An example of a manually instrumented Python *graph-pagerank* benchmark is displayed in Appendix C.

**Configuration** For both Python and Node.js, we install the necessary OpenTelemetry SDKs and libraries, including the OpenTelemetry API, SDK, and Exporter packages. Configuration includes setting up the service name for each application trace, validating that the traces reach the backend and exporting traces to the OpenTelemetry Collector via a gRPC endpoint. Figure 4.2 illustrates the configuration process for the Python-based *thumbnailer* benchmark, covering attributes setup, span exporting, span processing, and tracer initialization.

**Manual Instrumentation** We use manual instrumentation instead of automatic instrumentation due to the environment constraints of the OpenWhisk platform that prevent the use of instrumentation agents. The instrumentation adds spans to various functions, tracking execution details such as function names, execution times, and key parameters. Child spans are created for significant events, capturing essential actions during execution. For example, in the *uploader* benchmark, separate spans are created for download and upload events. Manual instrumentation is applied consistently across Python and Node.js applications, minimizing disparities by maintaining similar logic, spans, and attribute usage in both implementations (see Figure 4.3).

```python
resource = Resource(attributes={
    "service.name": "630.thumbnailer-opentelemetry"
})

trace.set_tracer_provider(TracerProvider(resource=resource))
tracer = trace.get_tracer("function")

otlp_exporter = OTLPSpanExporter(
    endpoint="http://192.168.1.104:4317",
    insecure=True
)

span_processor = SimpleSpanProcessor(otlp_exporter)
trace.get_tracer_provider().add_span_processor(span_processor)
```



**Figure 4.2:** Configuration example for manually instrumented Python application with OpenTelemetry.

**Exporting Traces** Traces are exported to the OpenTelemetry Collector, which is configured to receive traces from Python and Node.js applications through the gRPC endpoint. The Collector forwards traces to Elastic APM for visualization and validation in Kibana. This setup ensures accurate trace collection, and the export process is verified by confirming the traces' arrival in Kibana.

### 4.4.3 Benchmark Applications

The benchmark framework supports various runtime configurations for each application, including the number of function instances, concurrent invocations, repetitions, action type (cold or warm start), memory limit, and timeout. Below are the key runtime parameters used:

- **Number of Function Instances:** Each benchmark application is run with multiple replicas. Each replica is a separate pod running on the Kubernetes cluster. The number of replicas is equal to the *Concurrent Invocations* value.

**Python graph-pagerank**

```python
def handler(event):
    span = tracer.start_span("handler")
    ctx = trace.set_span_in_context(span)

    size = event.get('size')
    span.set_attribute("size", size)

    generate_graph_span = tracer.start_span("generate_graph", context=ctx)
    graph = igraph.Graph.Barabasi(size, 10)
    generate_graph_span.end()
```

**Node.js graph-pagerank**

```javascript
exports.handler = async function(event) {
    const span = tracer.startSpan('handler');
    const ctx = opentelemetry.trace.setSpan(opentelemetry.context.active(), span);

    const size = event.size;
    span.setAttribute('size', size);

    const generateGraphSpan = tracer.startSpan('generate_graph', undefined, ctx);
    const graph = generateBarabasiAlbertGraph(size, 10);
    generateGraphSpan.end()
```

**Figure 4.3:** Comparison of Python and Node.js code snippets for the graph-pagerank application. This figure shows part of the code responsible for manually instrumenting using OpenTelemetry. The code examples highlight the similarities in how the instrumentation is implemented in both frameworks. Both examples demonstrate span creation, context propagation, and attribute setting.

- **Concurrent Invocations:** Set to **10** for all experiments; this parameter represents the batch size for executions, allowing us to observe the application's performance over a more extended period with consistent load.
- **Repetitions:** Each benchmark is executed **10 000** times to ensure statistical significance and minimize the impact of anomalies.
- **Action Type:** The experiments are conducted in warm start mode, where the serverless functions utilize already-initialized environments for lower latency. Cold start measurements are discarded to focus on steady-state performance.

- **Memory Limit:** Defines the maximum memory allocated to each application during execution. Memory limits vary based on the application's requirements.
- **Timeout:** Specifies the maximum execution duration for each function before termination, preventing excessive resource consumption.

Table 4.1 summarizes the memory limits and timeout configurations for each benchmark application. These settings are consistent across Python, Node.js, instrumented, and non-instrumented versions.

**Table 4.1:** Task-based applications memory limit and timeout configuration.

| Benchmark | Memory Limit (MB) | Timeout (Seconds) |
|---|---|---|
| Dynamic HTML | 128 | 10 |
| Uploader | 128 | 30 |
| Thumbnailer | 256 | 60 |
| Video Processing | 512 | 60 |
| Graph PageRank | 512 | 120 |

## 4.5 Experiment Deployment

This section outlines the deployment process for the components used in the task-based application tracing overhead experiment, including Kubernetes cluster, OpenTelemetry Collector, Elastic APM, Kibana, OpenWhisk, and the benchmark applications.

### 4.5.1 Kubernetes Cluster Deployment

The Kubernetes cluster is deployed using Kind (Kubernetes IN Docker) on a single virtual machine. The Kind tool enables the setup of Kubernetes clusters within Docker containers. We create the cluster with the Kind CLI tool installed on the virtual machine.

### 4.5.2 Tracing Tools Deployment

The OpenTelemetry Collector, Elastic APM, Kibana, and Elasticsearch are deployed using Docker and Docker Compose, following a setup process similar to that outlined in

Section 3.5.1. By containerizing the tracing tools and hosting them outside the Kubernetes cluster, we reduce the potential impact of tracing on the benchmark applications and simulate real-world conditions where applications export traces to an external tracing backend.

### 4.5.3 OpenWhisk Deployment

Apache OpenWhisk is deployed within the Kubernetes cluster using the official Helm chart (57). We use the chart version 1.0.1 to install and configure OpenWhisk's components, including the Controller, Invoker, Kafka, CouchDB, Nginx, and Elasticsearch. We configure the following OpenWhisk action limits and values via the Helm chart:

- **Actions Invokes Per Minute:** This limit specifies the amount of invocations per minute. Configured to 1000 invocations per minute to prevent throttling during the experiment.
- **Actions Invokes Concurrent:** This limit defines the maximum number of actions that can run concurrently at any moment. We set the value to 100 to have sufficient concurrency for our experiments.
- **Triggers Fires Per Minute:** The trigger fires per minute parameter sets the maximum number of triggers that can be fired per minute. A trigger in OpenWhisk is an event that causes one or more actions to be invoked. We set the value to the same as the *Actions Invokes Per Minute* value – 1000.
- **Actions Memory Max:** This limit defines the maximum amount of memory a single OpenWhisk can consume. Increased to 1024 MB to accommodate the memory requirements of serverless applications, with benchmark applications using a maximum of 512 MB.

### 4.5.4 Benchmark Application Deployment Process with SeBS

The benchmark applications are deployed using the Serverless Benchmarks (SeBS) framework (37), The deployment process involves several steps to ensure the applications are correctly packaged and deployed on the OpenWhisk platform:

1. **Cache Check:** The framework first checks for up-to-date builds of the benchmark functions, avoiding unnecessary rebuilds.

2. **Code and Data:** Benchmark application code and data are copied to the build location.
3. **Dependencies:** Additional dependencies, such as files, images, and videos, are included if required. This step ensures that all external dependencies are available for the benchmarks.
4. **Platform-Specific Wrappers:** Shims are added to adapt the benchmark code to platform-specific requirements, providing compatibility with OpenWhisk's API.
5. **Deployment Packages:** Dependencies specific to the platform are installed, including the MinIO SDK for object storage integration.
6. **Code Packaging:** The codebase is organized according to the platform's requirements, preparing it for deployment.
7. **Docker Image Build:** Given OpenWhisk's 48 MB limit on code package sizes, all functions are deployed as Docker images. The application code and dependencies are copied into the Docker image and then pushed to the Docker registry.
8. **Creating OpenWhisk Action**: The final step involves creating or updating the OpenWhisk action using the Docker image, allowing the function to be executed on the platform.

## 4.6 Experiment Results

This section presents the results of the experiments evaluating the performance impact of distributed tracing on task-based applications. The following sections discuss the key findings and analyze the experiment results.

The experiment results are organized into three groups, each addressing one of the research questions and checking the corresponding hypothesis. We first present the overall performance impact of tracing on task-based applications in Section 4.6.1, answering the research question RQ4.1 and testing the hypothesis H4.1. We also compare the overhead introduced by tracing in different programming frameworks in Section 4.6.2, addressing the research question RQ4.2. Section 4.6.3 discusses the results of the tracing overhead across various workloads (research question RQ4.3).

We present the following main findings from this experiment:

**MF4.1** Tracing consistently increases latency across all task-based workloads, with low-latency tasks experiencing higher relative overhead percentages.

**MF4.2** Compute-intensive task-based applications, such as *graph-pagerank*, show lower percentage overhead (6.69%) despite substantial absolute latency increases.

**MF4.3** As baseline non-instrumented request duration increases, the overhead percentage generally decreases, indicating a relatively reduced impact.

**MF4.4** With serverless applications, Node.js framework generally shows a higher percentage overhead than Python, especially for low-latency tasks.

**MF4.5** Despite higher overhead percentages, the absolute overhead values for serverless applications in Python and Node.js are comparable.

### 4.6.1 Performance Impact of Tracing On Serverless Applications

**Table 4.2:** Performance metrics (mean and p99) for the serverless benchmarks comparing instrumented (Instr.) and non-instrumented (Non-Instr.) configurations.

| Benchmark | Mean Non-Instr. (ms) | P99 Non-Instr. (ms) | Mean Instr. (ms) | P99 Instr. (ms) |
|---|---|---|---|---|
| **Benchmark Time** | | | | |
| dynamic-html | 2.53 | 6.58 | 6.96 | 17.00 |
| uploader | 25.39 | 40.75 | 28.80 | 49.29 |
| thumbnailer | 112.61 | 177.00 | 121.94 | 186.00 |
| video-processing | 1411.57 | 2134.00 | 1797.22 | 3828.54 |
| graph-pagerank | 4579.33 | 9348.00 | 4885.69 | 10950.02 |
| **Client Time** | | | | |
| dynamic-html | 64.20 | 143.87 | 66.48 | 138.38 |
| uploader | 106.72 | 208.39 | 121.94 | 224.60 |
| thumbnailer | 220.16 | 370.60 | 314.87 | 746.39 |
| video-processing | 1463.01 | 2188.63 | 1848.05 | 3877.28 |
| graph-pagerank | 5244.06 | 9396.89 | 5556.00 | 10995.80 |

Table 4.2 presents the performance metrics for the evaluated serverless benchmarks, comparing the mean and 99th percentile (P99) latency for instrumented (Instr.) and non-instrumented (Non-Instr.) configurations. The table is divided into two categories: (1) *Benchmark Time*, which measures only the execution time of the function, and *Client Time*, which includes end-to-end latency experienced by the client.

For *Benchmark Time*, all benchmarks show higher mean and P99 latency when instrumented. For example, the *dynamic-html* benchmark shows a mean latency increase of 175%, rising from 2.53 ms (non-instrumented) to 6.96 ms (instrumented), while the P99 latency increases by 158% (6.58 ms to 17.00 ms). The *thumbnailer* benchmark, with a more moderate latency compared to others, shows an increase of 8.3% (from 112.61 ms to 121.94 ms) for mean latency and the P99 latency rising by 5.1%. More intensive tasks, such as *graph-pagerank*, exhibit a latency increase of 6.7% (from 4579.33 ms to 4885.69 ms), and the P99 latency increasing by 17.1%.

For *Client Time*, latency increases are evident across almost all benchmarks, with more significant performance degradation observed for longer-running applications. For *dynamic-html* benchmark, the mean latency increased by 3.55%, but the P99 value of the non-instrumented benchmark is higher than the instrumented counterpart. This suggests that for applications with low request durations, the instrumentation overhead may be insignificant in some cases because of network delays or the serverless platform overhead. However, other benchmarks show consistent performance degradation similar to the *Benchmark Time* results.

### 4.6.2 Tracing overhead comparison between Python and Node.js

Table 4.3 presents the overhead of the instrumentation for both Python and Node.js applications. The overhead represents the percentage increase from the non-instrumented mean latency to the instrumented mean latency. The table also includes the non-instrumented mean latency values for each benchmark, providing context for the baseline performance before tracing was enabled

We observe that the tracing overhead varies significantly between Python and Node.js across different benchmarks. For *dynamic-html*, Node.js exhibits a much higher overhead percentage (574.76%) compared to Python (72.77%). However, the non-instrumented mean latency for Node.js is much lower (1.03 ms) than Python's (4.04 ms), indicating

**Table 4.3:** Benchmark and Client results for Python and Node.js overhead percentages. Overhead indicates the latency increase with the instrumented version of the benchmark.

| **Benchmark** | **Python Overhead** | **Python Non-Instr. Mean** | **Node.js Overhead** | **Node.js Non-Instr. Mean** |
|---|---|---|---|---|
| dynamic-html | 72.77 | 4.04 | 574.76 | 1.03 |
| uploader | 10.57 | 34.35 | 19.48 | 16.43 |
| thumbnailer | 5.60 | 79.93 | 9.75 | 145.30 |
| video-processing | 70.69 | 1080.26 | 0.44 | 1742.89 |
| graph-pagerank | 1.28 | 1008.87 | 7.36 | 8149.79 |

that while the percentage increase is significant, the absolute overhead is more negligible for Node.js.

The results generally show that the higher the request duration, the lower the overhead tends to be. However, there are exceptions, such as *video-processing* for Python, where the overhead is almost the highest with 70.69% overhead, despite a high mean latency of 1080.26 ms. In addition, the *graph-pagerank* benchmark for Node.js shows a 7.36% overhead, which is considerably higher overhead than, for example, the Node.js *video-processing* benchmark, which has significantly lower mean latency.

The results indicate a noticeable difference in tracing overhead between the two languages. While Node.js frequently shows higher overhead percentages, the absolute mean latency is lower than Python's in some cases, but the absolute overhead value is comparable to the absolute overhead for Python.

### 4.6.3 The variation of performance impact across different workloads.

Figure 4.4 presents box plots for five serverless workloads across non-instrumented and instrumented configurations. The results of the benchmarks are aggregated over both Python and Node.js evaluations.

The results indicate that the performance impact of tracing instrumentation varies widely depending on the workload type. For example, the *dynamic-html* workload, with a relatively low baseline latency of 2.53 ms, shows an increase of median latency to 6.96 ms when instrumented, exhibiting an overhead of approximately 175%.

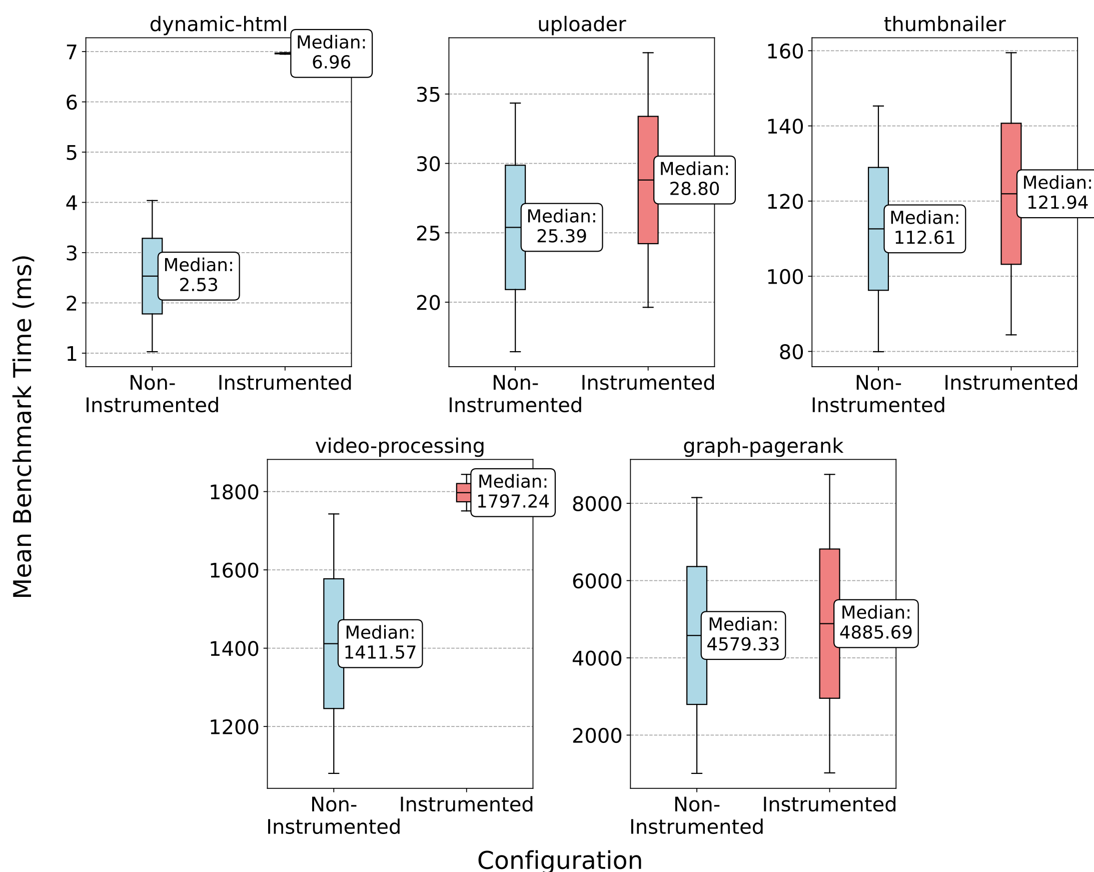


**Figure 4.4:** Comparison of performance impacts of distributed tracing for different task-based benchmarks, showing the mean benchmark time for non-instrumented and instrumented configurations across multiple benchmarks. The results are combined from both Python and Node.js applications.

As the median request duration increases, the overhead generally decreases. For instance, the *uploader* and *thumbnailer* with moderate median request durations, show a more modest increase–13.43% overhead for *uploader* and 8.29% for *thumbnailer*. However, the *video-processing* workload with a median latency of approximately 1411 ms exhibits a substantial increase of 27.32%. This considerable increase is the result of the 70.69% overhead in Python as Table 4.3 shows. The *graph-pagerank* workload, which has the highest baseline latency among the workloads with a median of 4579.33 ms, shows the lowest overhead with 6.69%.

The data highlights that tracing introduces a considerable performance impact

across the workloads. The functions with lower request duration show the highest increase but still the lowest increase in absolute values. The more compute-intensive workloads incur lower overhead in percentage but higher in terms of absolute values.

### 4.6.4 Results Overview

The results provide insight into the performance impact of distributed tracing in task-based applications. We address each of the research questions and validate whether the initial hypotheses set for each question hold.

**RQ4.1: What is the performance overhead of tracing in task-based applications?** The findings reveal that enabling tracing leads to a measurable increase in latency across all the tested applications, validating hypothesis H4.1. Results in Section 4.6.1 show that for short-running benchmarks, the median latency can increase by 175%. With an increased non-instrumented latency, the relative overhead generally decreases, but the absolute mean latency increases with instrumentation enabled. These results emphasize the need to weigh the benefits of tracing against the associated latency overhead, especially in applications where performance is a priority.

**RQ4.2: How does the performance overhead of tracing in task-based applications differ between different programming frameworks?** The experiment results reveal some differences in overhead between Python and Node.js While Node.js often exhibited higher overhead percentages–especially in low-latency tasks like *dynamic-html*– the actual absolute latency increase was comparable to Python. The cause of the high percentage overhead for Node.js tends to originate from lower non-instrumented latency values. These findings highlight that even though there are some differences between the frameworks, but there is not enough proof that the difference is significant between Python and Node.js.

**RQ4.3: How does the performance overhead of distributed tracing vary across different workloads?** The results indicate that tracing overhead varies significantly across different workload types. We observe that the low-latency benchmarks experience a high relative performance impact from tracing, with up to a 175% increase

in latency. Meanwhile, high-latency tasks see a much smaller relative increase (for example, 6.69%) despite a substantial absolute latency increase. This suggests that for high-latency applications, the relative impact of tracing is minimized due to the already large execution times, whereas low-latency tasks are more sensitive to the additional overhead tracing introduces.

Overall, the results confirm the hypotheses H4.1 and H4.3, but not hypothesis H4.2. We observe that tracing introduces measurable increased latency across serverless applications and that the workload length significantly influences the relative impact of tracing overhead.

# 5

# Sources of Overhead in Distributed Tracing

This chapter examines the detailed performance costs associated with distributed tracing, aiming to identify, categorize, and measure the overhead in various application environments. The objective is to provide insight into which stages of the tracing lifecycle impact the performance the most. The following sections discuss the experiment's design, metrics, data collection, setup, configuration, and deployment. Finally, we present and analyze the result to identify the primary sources of overhead in tracing.

## 5.1 Experiment Design

Within this experiment, we have two objectives: (1) categorize the processes in distributed tracing into distinct categories: configuration, instrumentation, and export, and (2) measure each category's overhead to understand each step's impact on the system performance. The experiment involves profiling applications from the previous experiments described in Chapter 3.3.2 and Chapter 4.

### 5.1.1 Motivation

The motivation for examining the sources of overhead in distributed tracing is behind the need to understand the balance trade-off between system observability and performance. The breakdown of overhead into different categories is not widely explored in the current research. We aim to fill the gap in the literature by identifying the key contributors to

the overhead. The goal is to provide insight into which processes in distributed tracing cause the most overhead to understand where the optimization effort should be focused.

### 5.1.2 Research Questions

**RQ5.1. What are the main contributors to overhead in distributed tracing?** This question seeks to identify which stages in the distributed tracing process—configuration, instrumentation, or export—contribute the most to performance overhead. We analyze each stage to pinpoint the area of tracing that contributes the most. We aim to provide insight into where optimizations can yield the most significant improvements

**RQ5.2. How does the overhead of distributed tracing vary between cold start and warm start scenarios?** This research question examines the difference in distributed tracing overhead between cold start and warm start scenarios. The cold start includes the initialization and configuration of the OpenTelemetry tracer, exporter, and tracer attributes. We aim to quantify how tracing impacts the performance of both scenarios, which are relevant in environments such as serverless computing where cold starts are frequent.

**RQ5.3. How does the overhead in our specified categories of distributed tracing vary between request-based and task-based applications?** This question investigates how different applications (microservices and serverless) overhead varies across the tracing processes–configuration, instrumentation, and exporting. The research aims to identify whether certain parts of the tracing process are more costly in one application type than the other.

### 5.1.3 Hypotheses

**H5.1. Exporting data is the main contributor to overhead in distributed tracing.** We anticipate that exporting is the largest contributor to the overhead in distributed tracing. Exporting involves processing and transmitting the trace data to a backend system. In systems where numerous events are traced, we expect that the amount of trace data and the trace size can lead to potential performance degradation in data processing and transmission (10, 47, 58). We expect that the

instrumentation overhead is relatively low (15). This hypothesis is related to research question RQ4.1.

**H5.2. Cold start scenarios incur significantly higher overhead than warm start scenarios.** Cold start scenarios in serverless computing incur a significant reduction in performance due to initialization of configuration or dependencies (59, 60). In contrast, warm start scenarios benefit from pre-initialized resources, which improves performance. The hypothesis suggests that the cold start will have significantly higher overhead due to additional time spent initializing and configuring the tracer. This hypothesis refers to research question RQ4.2.

**H5.3. Both serverless and microservices architectures face similarly high overhead in the same categories.** Due to the frequent and distributed nature of function invocations and service interactions in both architectures, we anticipate that the processes of instrumenting each function or service and exporting trace data impose similar performance overheads in both environments. This hypothesis addresses research question RQ4.3.

### 5.1.4 Sources of Overhead

In distributed tracing, various sources contribute to the overall overhead observed in the application and its instrumentation. In this experiment, we evaluate the sources of overhead in the computational part of distributed tracing (47). To understand and quantify the overhead of distributed tracing, we divide the function calls into four categories: configuration, instrumentation, export, and workload. The configuration, instrumentation, and export categories refer to the work executed by distributed tracing, and the workload refers to executing the application's primary tasks. The description and content of each category is below:

- **Configuration:** Configuration overhead arises from the initial setup required to enable distributed tracing in an application (61). This includes the time and resources consumed during the initialization of the tracing framework, establishment of connections to trace exporters, and the configuration of tracing parameters such as sampling rates and context propagation settings. Although configuration

overhead is typically a one-time cost at the start of the application, it can affect the startup time and overall responsiveness, especially in environments where applications are frequently started and stopped, such as serverless applications.

- **Instrumentation:** Instrumentation refers to the operations that are responsible for capturing trace data. This includes inserting trace points into the code, where spans are started and stopped, and what metadata is collected (62).
- **Export:** Export overhead is incurred when trace data is transmitted from the application to a trace collector or backend (10, 13, 14). This involves formatting, batching, and sending of the trace information. The impact of export overhead depends on factors such as the volume of traces, batching strategy, and network latency.
- **Workload:** The workload category in this context refers to the time required to execute only the application task or request, excluding any tracing operations. We classify the workload separately to assess the overall impact of tracing on the system's performance.

**Categorization Process** The distribution of the categories (configuration, instrumentation, and export) mainly reflect the general documentation structure of the instrumentation tools. We noticed that several guides, such as OpenTelemetry and Elastic APM, are structured by first starting with the setup and the configuration of the instrumentation libraries and the configuration. The configuration process is followed by the instrumentation specifics (adding spans, events, and attributes), and finally, the documentations conclude with exporting the trace data. Each category can be further expanded and examined more in-depth (for example, breaking down the instrumentation processes into context propagation and span creation). However, our goal is first to establish the general categories to ensure a structured analysis.

## 5.2 Metrics and Data Collection

In this section, we describe the metrics and the data collection methods used in this experiment to evaluate the sources of overhead in distributed tracing. The primary tool

used for data collection is Python's built-in library *cProfile*, which captures function-level measurements of performance. We focus on two key metrics: Cumulative Time and Total Time, which we use to evaluate the time spent by tracing operations in each of the categories defined earlier.

### 5.2.1 Metrics

**Cumulative Time** Cumulative time refers to the total time spent in a function and all the other functions it calls. Using cumulative time allows us to capture the full impact of tracing operations because the sub-calls might not always have tracing-related names, making it challenging to capture their impact otherwise.

**Total Time** Total time measures the direct time spent in a function itself, excluding the time spent in the functions it calls. This metric helps us distribute the tracing processes into correct categories more accurately because some of the sub-calls of the operations might also include operations from other categories. By visualizing and analyzing the profiling data, we can measure the impact of each category more accurately through the combination of *total time* and *cumulative time*. For example, if a tracing operation A calls tracing operation B, we can not add the cumulative time of both measurements due to the overlap.

### 5.2.2 Profiling

To capture the performance data and identify the sources of overhead introduced by distributed tracing, we use Python's built-in profiler library called *cProfile* (63, 64) for our framework. We also evaluate several alternative profiles such as *pprofile* (65), *py-spy* (66), and *Scalene* (67, 68). We evaluated the tools based on three requirements: whether the tool provides us (1) a sufficient amount of information about function calls, (2) precise, detailed measurements about the application calls and (3) a method to save the measurements for each test iteration. The measurements are needed to quantify the amount of overhead caused by distributed tracing, Moreover, information about the function calls is required to identify and categorize them into specific categories, such as exporting and configuration. The evaluation consisted of reading the documentation, studying the examples and performing profiling on a test program. We give a brief overview of the tools and the reasoning behind using or not using the tool.

- **cProfile:** *cProfile* is a profiling tool capable of monitoring all function calls, returns, and exception events (64). It also provides precise timings (including total time, per-call time, and cumulative time) and number of calls for each function. *cProfile* also offers reasonable overhead (64). We selected this tool since it shows measurements of each line of code, providing sufficient information about the overhead sources and allowing us to organize the function calls into different categories. Furthermore, the *cProfile* profiling statistics format is supported by many visualizing tools that help understand the profiling results.

- **pprofile:** This tool is another deterministic profiler that provides line-granular measurements for the profiled applications (65). Our tests and evaluation showed that *pprofile* satisfied our requirements. However, *pprofile*'s documentation explains that the deterministic profiling mode has a large overhead (65), with a slowdown rate of 36.8 times compared to 1.7 times compared to cProfile (67). Thus, we decided to opt for *cProfile* instead since it also fulfilled all the requirements but was significantly less intrusive.

- **py-spy:** The *py-spy* tool is a sampling profiler, which offers low overhead and does not run in the same process as the profiled Python application (66). The tool also has built-in visualizing capabilities to analyze the application calls. The tool also offers a live view of the functions and their details during the application runtime. However, at the time of the research, *py-spy* has issues with the compatibility with the OSX operating system, which arised difficulties with the evaluation. In addition, the initial assessment of the tool concluded that it might be challenging to implement *py-spy* into our framework since it runs as a different process and does not have a Python library, making it difficult to record the measurements during the experiment. Therefore, we decided not to use the *py-spy* tool for our use case.

- **Scalene:** A high-performance profiler capable of measuring CPU, memory, and GPU usage with low overhead. Our brief evaluation showed that Scalene did not satisfy the requirements we set for the profiler. First, Scalene shows the profiling results in several formats but does not provide a programmatic or convenient way to save the measurements for each iteration. Secondly, the profiling output of our

test program did not show the measurements for all of our code lines but only for a few (even when modifying the CPU percent threshold to a very low number).

### 5.2.3 Visualizing the Profiled Application

Visualizing the profiled application helps us ensure the accuracy of our measurements and the correct categorization of the sources of overhead. In addition, the profiled data helps us confirm that we collect the correct measurements and classify the appropriate function calls into the respective categories of configuration, instrumentation, export, and task execution. Since *cProfile* reports the execution time in cumulative time, which contains the times for each subsequent function call, we must ensure we do not count the distributed tracing functions several times. We used two types of visualization methods: flame graphs and DOT graphs.

#### 5.2.3.1 Flame graphs

The flame graphs visually represent the hierarchical structure of function calls within an application (69). An example of a flame graph is displayed in Figure 5.1, which shows the profiling information of an OpenTelemetry instrumented *dynamic-html* application. The stacked bars represent the function calls, where the width of each bar corresponds to the time spent in that function. We generate flame graphs from the *cProfile* generated profiling data using the *flameprof* (70) tool. We use flame graphs to visually identify the most time-consuming processes and validate whether the measurements match the flame graph. We also use the graphs to understand the call stack and identify the function calls related to OpenTelemetry.

#### 5.2.3.2 DOT graph

DOT graphs represent the application's call structure as a directed acyclic graph (DAG) (71). The nodes in the DOT graph represent a function, and the edges represent the calls and the relationship between them. This visualization method helps understand the flow of requests, their relations and the order of the calls. We use the DOT graph to determine which functions and their corresponding cumulative call times we record for the measurements. In addition, we analyze if some function calls have sub-calls that fall under a different category. We use a Python tool named *gprof2dot* (1) to generate

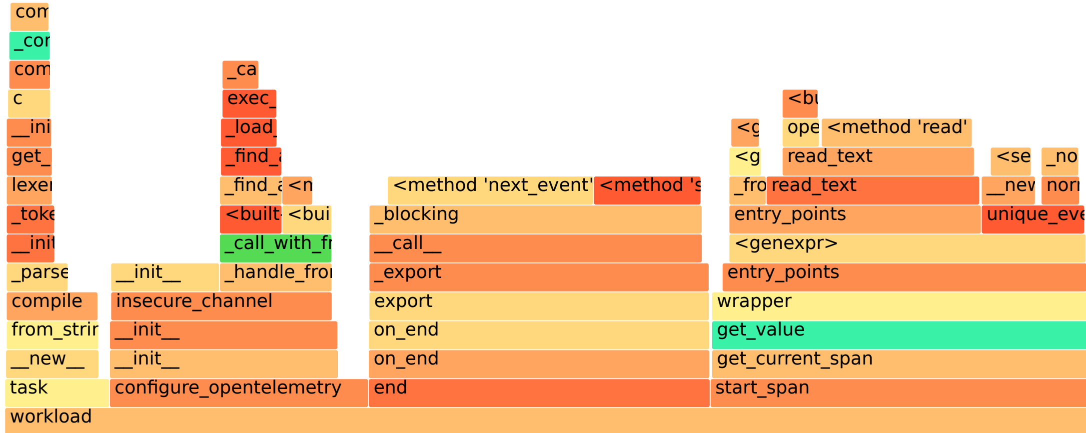


**Figure 5.1:** Flame graph of the *dynamic-html* task-based application with OpenTelemetry instrumentation. The CPU threshold for displaying function calls is set to 2.5%.

the DOT graph based on the *cProfile* data output. Figure 5.2 displays a DOT graph generated with *gprof2dot* based on the *cProfile* generated from a task-based application.

## 5.3 Experiment Setup

In this chapter, we describe the setup used for the sources of overhead experiment. The experiment reuses a subset of the applications from Chapter 3 and Chapter 4, focusing on both request-based and task-based applications. This selection of applications enables us to explore tracing impacts across different workload types, including both short- and long-duration microservices and serverless workloads.

### 5.3.1 Experiment Applications

The experiment reuses the applications from Chapter 3 and Chapter 4: request-based and task-based applications. We use a subset of the applications used in the previous experiments: */db* and */updates* from the microservice experiment, and *dynamic-html* and *graph-pagerank* from the serverless experiment. A smaller group of applications allows us to maintain the research focus on a small but varied group of workloads. The group contains applications from different architecture types with short-duration and long-duration workloads.

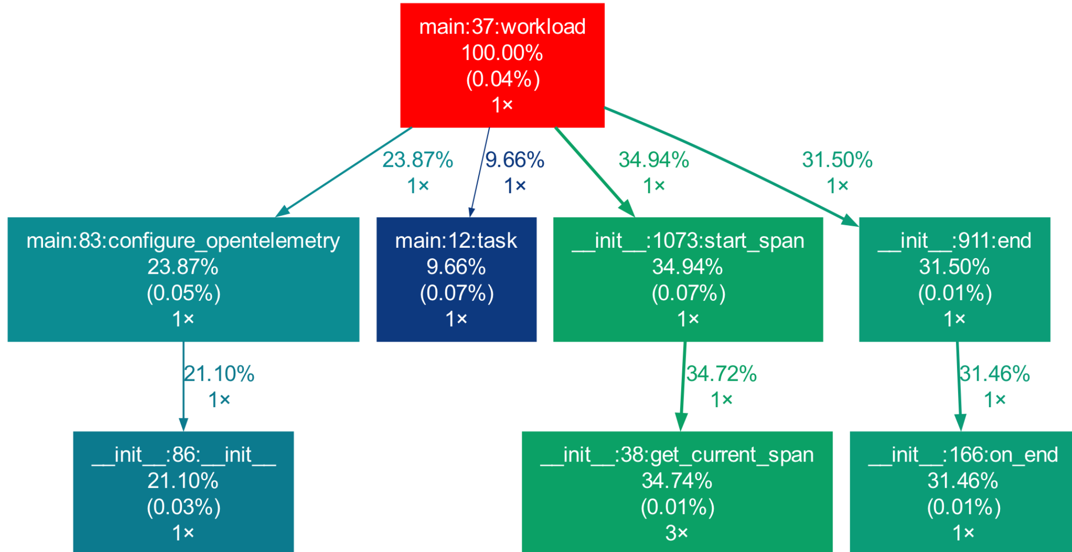


**Figure 5.2:** DOT graph of a task-based application with OpenTelemetry instrumentation. The graph is generated with gprof2dot (1). The edge and node CPU threshold is set to 9.5%, and the depth of the graph is set to two for readability.

#### 5.3.1.1 Request-Based Applications

The request-based application is a Python Flask-based web service implementing endpoints from the TechEmpower web benchmarks (72). The two selected endpoints represent typical database-driven microservices involving interaction with a PostgreSQL database. Our application consists of the following two endpoints:

- **/db:** This endpoint simulates the database interaction, involving reading a random entry from the PostgreSQL database.
- **/updates:** This endpoint handles a database update request, which updates a number of database entries based on the provided parameter.

Both endpoints are instrumented automatically with OpenTelemetry to trace the lifecycle of each request, including database interactions.

#### 5.3.1.2 Task-Based Applications

The task-based applications represent workloads used in serverless environments. We use two applications in this experiment that originate from the serverless benchmarking

suite (SeBS) (28, 37) and are also used in Chapter 4:

- **dynamic-html:** This application generates dynamic HTML content based on input parameters. It serves as a short-duration workload in the serverless domain for this experiment.
- **graph-pagerank:** This application runs the PageRank algorithm, which processes a large graph dataset to compute node ranks. This workload serves as a long-duration, compute-intensive task to evaluate the impact of tracing on prolonged operations.

Both applications are manually instrumented using OpenTelemetry to capture trace data. The instrumentation is identical to the versions used in the serverless experiment of Chapter 4. The applications and the instrumentation process is described more in-depth in Chapter 4.

### 5.3.2 Environment

The experiment environment uses Docker and Docker Compose to manage and execute the applications in isolated containers. The experiment framework runs within a Docker container, executing both the request-based and task-based applications. This setup ensures consistency and reproducibility across runs, as well as isolation between the application workloads and the underlying host system.

Unlike the previous experiments in Chapters 3 and 4, the frameworks from those chapters cannot be reused here due to the need to incorporate cProfile into the tracing instrumentation. As a result, the applications are run in separate processes to facilitate the integration of profiling tools, allowing for a more detailed examination of tracing overhead.

## 5.4 Experiment Configuration

This section describes the configurations and parameters used for the benchmark applications. Section 5.4.1 gives an overview of the cold start and warm start mode for the serverless applications. Section 5.4.2 describes the parameters used for all the applications used in this experiment.

### 5.4.1 Cold Start and Warm Start

For the task-based applications, we evaluate two modes of operation: cold start and warm start.

- Cold start refers to starting the application from a completely fresh state with no preloaded resources or configurations. Profiling begins from the initialization phase, which includes setting up the task and configuring tracing (OpenTelemetry).
- Warm start simulates a scenario where the application has already been initialized, with its state or configuration loaded, reducing the overhead typically associated with setup.

In this experiment, we simulate cold starts by profiling the initialization, configuration, and task execution phases. For warm starts, we first initialize OpenTelemetry and other configurations and then profile only the task workload, focusing on the instrumentation and export phases.

For request-based applications, only the warm start mode is evaluated. These applications are designed to handle multiple requests, and it is uncommon for them to be started fresh for a single request.

In summary, we run the experiment with six configurations: (1) *dynamic-html-cold*, (2) *dynamic-html-warm*, (3) *graph-pagerank-cold*, (4) *graph-pagerank-warm*, (5) */db*, and (6) */updates*.

### 5.4.2 Application Parameters

Table 5.1 displays the parameters used for the request-based and task-based applications used in the experiments. We describe the parameters used for each benchmark and provide the value used.

## 5.5 Experiment Deployment

This section discusses the deployment process for the experiment, which is conducted within Docker containers and orchestrated with Docker Compose to ensure consistency, isolation, and reproducibility. The deployment strategy is designed to run each application or endpoint experiment separately to ensure that the experiments do not interfere.

**Table 5.1:** Parameters used for the request-based and task-based benchmarks for evaluating the sources of overhead.

| **Benchmark** | **Parameter** | **Description** | **Value** |
|---|---|---|---|
| dynamic-html | random_len | Number of items in the unordered list inside the HTML template | 10 |
| graph-pagerank | size | Number of vertices in the graph | 10000 |
| /db | – | – | – |
| /updates | queries | Number of database entries to update | 10 |

### 5.5.1 Docker Containerization

The experiment is executed within Docker containers to ensure consistent and isolated environments. Figure 5.3 shows the experiment's deployment architecture. We run each component in a separate Docker container and orchestrate the components with Docker Compose (73, 74). The deployment consists of the following key components:

- **Application Containers:** Both the task-based and request-based applications are deployed within an individual Docker container. This container includes the necessary dependencies, application code, and the experiment framework responsible for executing the tasks and collecting profiling data. All applications are deployed with a limited amount of resources: 1 CPU unit.

- **PostgreSQL:** The request-based application requires a PostgreSQL database for both endpoints used in this experiment. The database is deployed with three replicas to ensure the database is not a bottleneck. The applications connect to the Docker service endpoint, and the request is forwarded to one of the replicas.

- **OpenTelemetry Collector:** The OpenTelemetry Collector is deployed in a separate Docker container in the same Docker network. This service is configured to accept traces on both HTTP and GRPC endpoints. The collector is set up to forward the data to an Elastic APM server so we can validate and visualize the traces in Kibana.

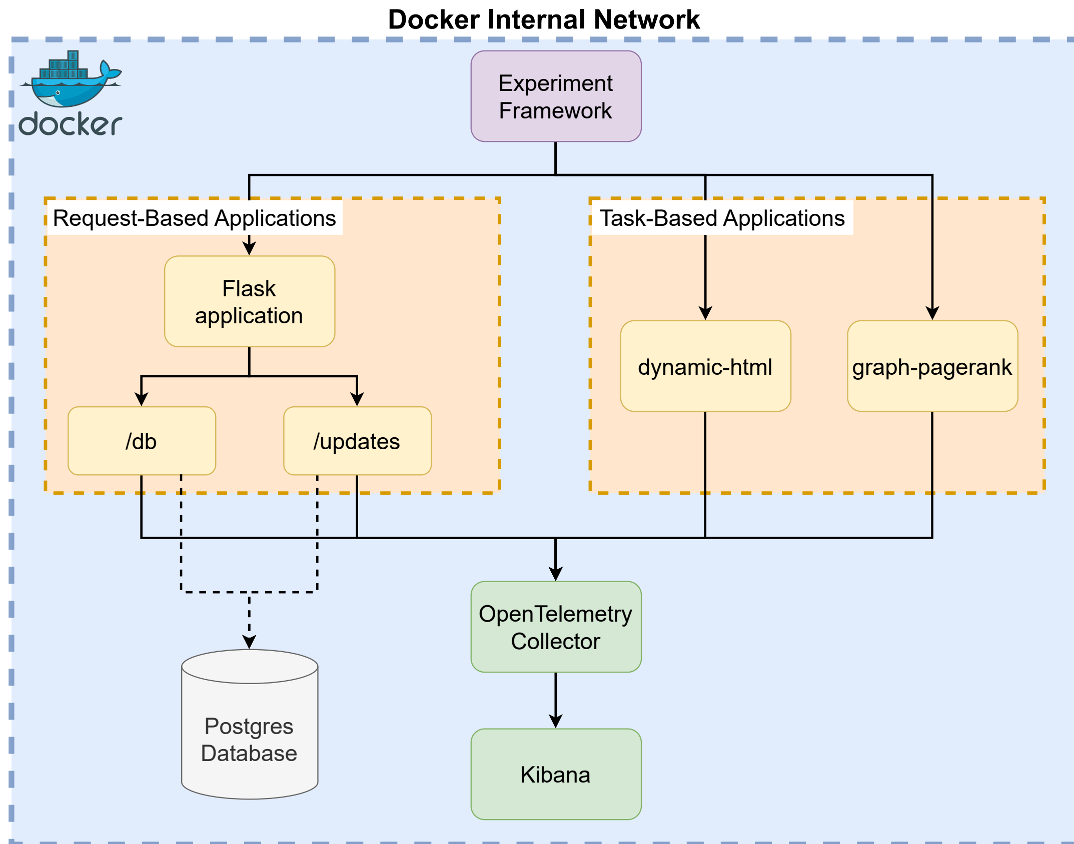


**Figure 5.3:** Experiment architecture for evaluating the performance impact of distributed tracing processes.

- **Docker Network:** All containers are connected via an internal user-defined Docker network (75, 76), allowing communication between the application containers, database, and the OpenTelemetry Collector.

- **Experiment and Service Orchestration:** The orchestration of the experiments and its services are managed using Docker Compose. We set up the prerequisite components (PostgreSQL, OpenTelemetry Collector, and Network) and then run each application experiment one-by-one. The Docker Compose manifests declare the parameters of the experiments, such as the number of iterations to run, the application name, and the task-based application start modes (cold or warm).

## 5.6 Experiment Results

In this section, we discuss the results of the distributed tracing overhead experiment, focusing on the main contributors to the overhead, the comparison between cold and warm starts, and the differences in overhead between microservices and serverless applications. The results are based on profiling data collected from request-based and task-based applications.

The main findings of this experiment are:

**MF5.1** Configuration overhead is a major contributor in cold-start scenarios for serverless applications.

**MF5.2** Instrumentation overhead remains minimal for serverless applications but is relatively high in request-based applications.

**MF5.3** Exporting the trace data consistently incurs the highest overhead in request-based applications.

**MF5.4** Exporting causes a considerable overhead in short-duration serverless applications.

### 5.6.1 Main Contributors of Overhead

Table 5.2 shows the overview of distributed tracing overhead in different categories (Configuration, Instrumentation, Export, Task) for task-based and request-based applications. We analyze the average and 99th percentile latencies for the total duration and also observe the percentage of execution time for each of the categories. Task-based applications are further divided into cold and warm start scenarios.

**Configuration** We observe that the configuration overhead is very significant in short-duration task-based applications with cold starts, with *dynamic-html-cold* having configuration account for 52.63% of the total execution time. For the *graph-pagerank-cold* benchmark with a longer run duration, the configuration only made up for 0.85% of the total execution time. The configuration is already done in warm start scenarios and is, therefore, 0% For the request-based applications */db* and */updates*, the configuration contributed only a small percentage of 1.30% and 0.25%, respectively. For the

**Table 5.2:** Sources of distributed tracing overhead in request-based and task-based applications based on profiling data. The percentages for each category represent the average contribution of the operations.

| **Application** | **Avg. Time (ms)** | **P99 (ms)** | **Config.** | **Instr.** | **Export** | **Task** |
|---|---|---|---|---|---|---|
| **Task-Based Applications** | | | | | | |
| dynamic-html-cold | 38.83 | 52.41 | 52.63% | 1.36% | 13.96% | 31.95% |
| dynamic-html-warm | 17.27 | 36.96 | 0.0% | 2.93% | 28.32% | 68.60% |
| graph-pagerank-cold | 2731.21 | 3484.64 | 0.85% | 0.03% | 1.79% | 97.33% |
| graph-pagerank-warm | 2658.97 | 4866.62 | 0.0% | 0.04% | 1.64% | 98.32% |
| **Request-Based Applications** | | | | | | |
| /db | 9.66 | 12.39 | 1.30% | 8.02% | 52.43% | 38.25% |
| /updates | 51.83 | 58.89 | 0.25% | 11.97% | 39.57% | 48.21% |

request-based applications, a small contribution came from fetching the tracer to start the spans.

**Instrumentation** The results show that the instrumentation part of tracing generally has the lowest impact. Instrumentation overhead is relatively minor in task-based applications across cold and warm scenarios. This indicates that the tracing operations have a minimal effect on the overall performance of task-based applications. However, with the request-based applications, the contribution of the instrumentation was significantly higher, with 8.02% for */db* benchmark and 11.97% for */updates* benchmark. The considerably higher contribution compared to task-based applications is likely because of additional instrumentation operations done by the SQL and Flask instrumentor.

**Export** The export is consistently the most substantial contributor to the performance impact caused by tracing. In request-based applications, export accounts for 52.43% for */db* and 39.57% for */updates*; this makes export the most significant contributor to the microservice scenarios. Similarly, in the task-based applications bench-

marks, the export overhead is significant in the *dynamic-html* benchmark, with 13.96% contribution for the cold scenario and 28.32% for the warm scenario. For the *graph-pagerank*, the relative impact of export is significantly lower at around 1% for both warm and cold starts.

### 5.6.2 Comparing Cold Start and Warm Start

Figure 5.4 presents the distribution of tracing overhead across different categories–configuration, instrumentation, export, and task execution–over multiple runs in cold and warm start scenarios. The top row box plots show the *dynamic-html* benchmark and the bottom row shows the distribution for the *graph-pagerank* cold and warm start scenarios.

**dynamic-html** For the *dynamic-html-cold*, configuration occupies a significant portion of the execution time, covering approximately 40-60% each run. The main contributor is exporting the trace data in the warm start scenario for *dynamic-html-warm* benchmark. The export also causes a significant impact on the performance of the cold scenario. Instrumentation takes up only a small but consistent portion of the whole execution time in both warm and cold scenarios for the *dynamic-html* benchmark. Table 5.2 also shows that the cold-start scenario's average and 99th percentile latency are considerably higher than the warm-start scenario.

**graph-pagerank** In *graph-pagerank*, the configuration's impact on the performance is considerably lower than in *dynamic-html-cold* benchmark. The tracing impact on performance is slightly higher with the cold start than with the warm start, but the difference is barely visible. The exporting cost is also similar in both scenarios, as shown in Table 5.2. The impact of instrumentation is insignificant for both warm and cold starts.

### 5.6.3 Comparison of Microservices and Serverless Applications Overhead

Figure 5.5 illustrates the distribution of time (milliseconds) spent in each of the overhead categories (configuration, instrumentation, and export) for serverless applications

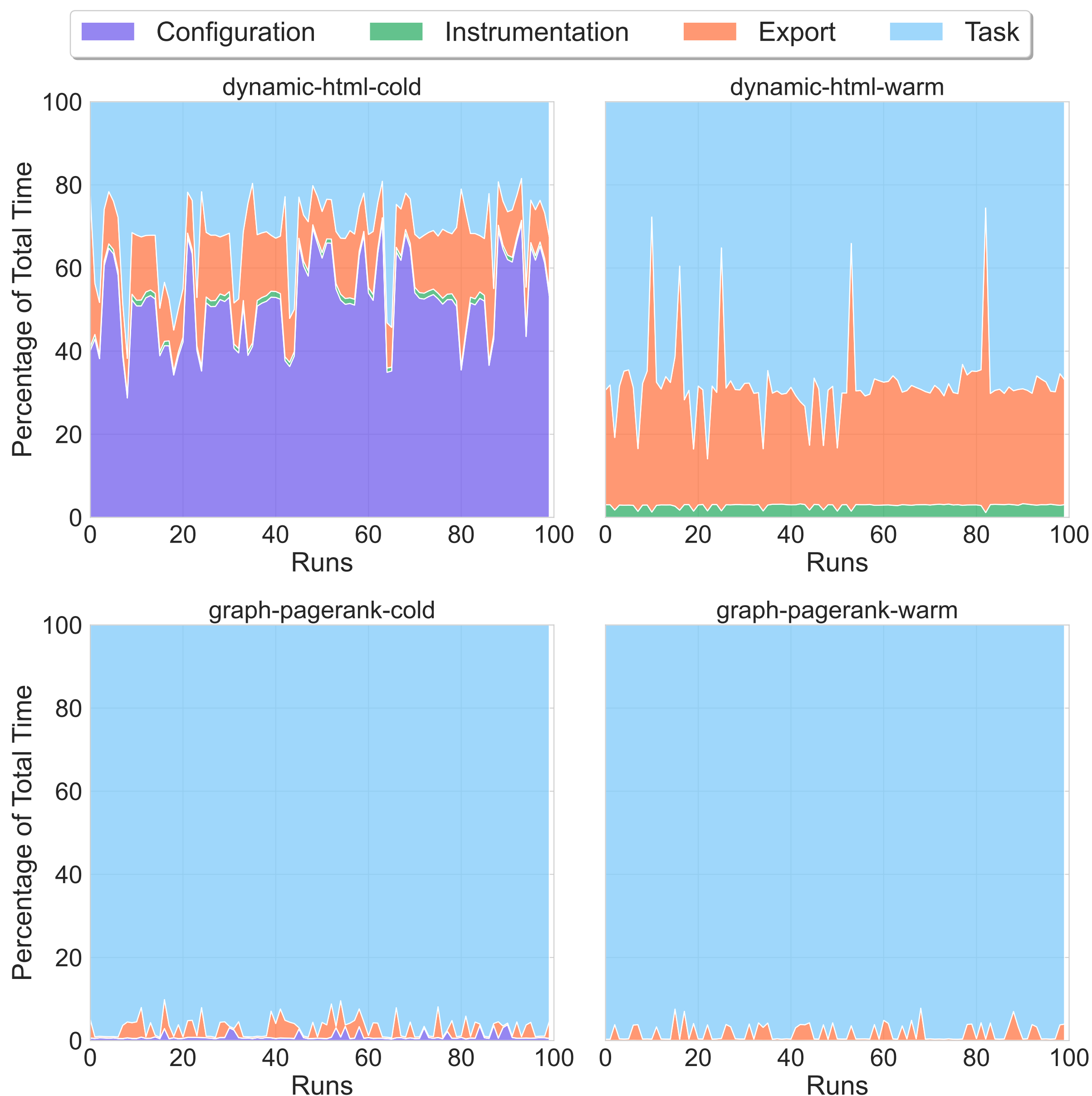


**Figure 5.4:** Distribution of tracing overhead categories (Configuration, Instrumentation, Export, Task) across 100 runs for task-based applications under cold start and warm start scenarios.

and microservices. The results for each group are combined from all the evaluated benchmarks.

The results show that the overhead across the categories varies significantly between

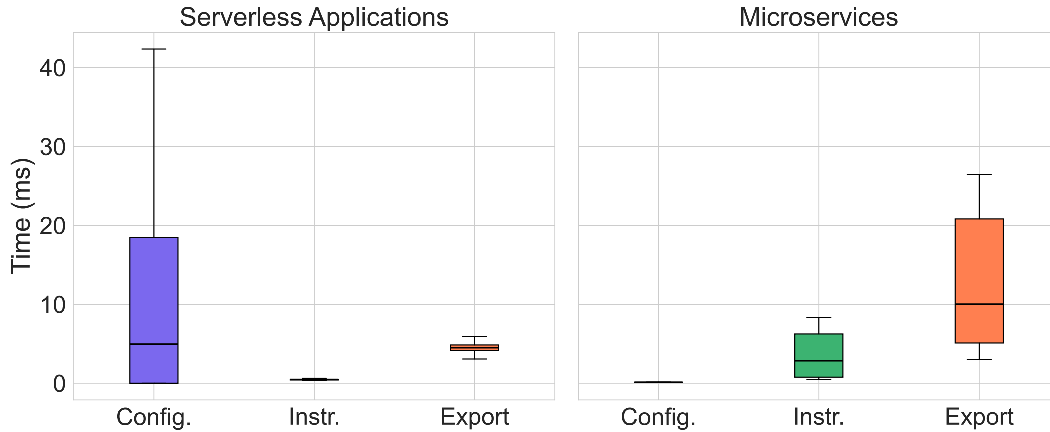


**Figure 5.5:** Comparison of distributed tracing overhead across configuration, instrumentation, and export phases for serverless applications and microservices.

the serverless applications and microservices. Configuration is the main contributor to serverless applications' overhead due to the cold starts. Configuration also has significant variability compared to other categories. Export is the second largest contributor to the overhead for serverless applications but has significantly less variability than configuration. Instrumentation has the least impact on the performance of serverless applications.

For microservices, the export overhead dominates with the highest median and variability. Instrumentation has moderate overhead and lower variability, likely due to the additional instrumentation operations from SQL and Flask instrumentors. Configuration overhead is minimal in terms of both time and variability because the configuration part was minimal for microservices.

Overall, this comparison highlights that serverless applications face considerable configuration overhead due to cold starts. The main contributor for microservices is exporting, with the instrumentation operations also causing considerable impact.

### 5.6.4 Results Overview

**RQ4.1: What are the main contributors to overhead in distributed tracing?** The results confirm that in most cases, exporting the trace data is indeed the primary contributor to the overhead; this aligns with the hypothesis H4.3. The only exception

is the cold-start scenario for short-duration benchmark *dynamic-html-cold*, where the main contributor is configuration. Overall, export operations consistently show the most significant impact on performance, especially in request-based applications, where export times can reach up to 52.43% of the total execution time. Configuration and instrumentation contribute much smaller proportions, with instrumentation significantly impacting request-based applications.

**RQ4.2: How does the overhead of distributed tracing vary between cold start and warm start scenarios?** We observe that hypothesis H4.1 holds, meaning that cold start scenarios incur significantly higher overhead than warm start scenarios. In the *dynamic-html-cold* benchmark, the configuration contributes over 50% of the total execution time in cold starts, and the latency impact is also significant. The average latency is considerably higher for *graph-pagerank* benchmarks, causing the configuration overhead to have a relatively low impact. This pattern highlights the performance impact of cold starts in short-duration serverless applications.

**RQ4.3: How does the overhead in our specified categories of distributed tracing vary between request-based and task-based applications?** The results show that our hypothesis H4.3 does not hold. While both serverless applications and microservices face significant export overhead, the impact of serverless applications' configuration is significantly higher than that of other categories due to cold starts. Exports are the main contributor to microservice applications. Furthermore, instrumentation overhead is insignificant in both *dynamic-html* and *graph-pagerank* serverless applications but is significant with the microservices by contributing 8.02% and 11.97% of overhead. Overall, we observe that the sources of overhead are different for microservices and serverless applications.

The experiment results confirm hypotheses H4.1 and H4.2, with exporting being the main contributor to tracing overhead and cold starts incurring significantly higher overhead than warm starts. Meanwhile, hypothesis H4.3 does not hold as serverless and microservice applications experience different overhead distributions.

# 6

# Related Work

This chapter examines the current state of research on distributed tracing, focusing on performance overhead and the comparative evaluation of tracing tools. After outlining the key research gaps identified in the literature, the chapter reviews studies that address tracing overhead or research instrumentation tools.

## 6.1 Research Gaps

Despite the amount of research on distributed tracing, several key gaps exist in the current research. These gaps include a limited understanding of tracing overhead across its distinct phases, the absence of detailed and comprehensive performance analyses, and the lack of studies spanning multiple architectural paradigms. We identified the following gaps in existing research during the literature review:

1. **Limited Analysis of Tracing Phases:** Existing research generally measures the overall performance impact of tracing but does not break down the overhead into specific categories. Some studies address computational and storage overheads (47), but none looked into the export, configuration, or instrumentation part of distributed tracing. Such a breakdown would provide valuable insights into where the most significant sources of overhead lie, allowing for more targeted optimizations.

2. **Limited Comprehensive Analysis of Performance Overhead:** Another gap lies in the limited scope of distributed tracing's performance analysis in existing research. Many studies fail to describe the system under test, often not reporting

details such as application architecture, instrumentation method, absolute values of metrics, or even not reporting the observed overhead metric at all (13, 15, 47). Therefore, comparing the distributed tracing tools, methods, and overhead across research is difficult since there is a lack of a unified benchmarking method and structure.

3. **Lack of Analysis Across Multiple Architectures and Runtime Lengths:** Tracing has been studied in isolated architectural settings, but there is a gap in research comparing tracing overhead across diverse application architecture types, such as microservices and serverless. Most studies focus on one type of application or barely describe the benchmarks (13, 15, 45). Furthermore, most research focuses only on a single application time, overlooking how tracing affects workloads with varying execution times, such as those with 10 milliseconds compared to 10 seconds.

## 6.2 Performance Overhead of Tracing

Distributed tracing offers visibility into complex systems but can impose a significant performance cost. This overhead is a critical concern for systems where performance is paramount. Studies show that the overhead caused by tracing can vary widely, depending on factors such as application type, workload, and the tracing tools used. Table 6.1 summarizes the overhead reported in the literature, typically measured either throughput or latency. However, many studies do not report the absolute performance values, making it difficult to perform direct comparisons between tools. We also add the results from the experiments conducted in this thesis for comparison.

Google Dapper, designed for large-scale infrastructures, minimizes overhead through sampling, often tracing only 1 in every 1,000 requests (15). This strategy allows Dapper to maintain low performance impact while providing valuable insights into system behavior. Dapper's reported latency overhead ranges from 0% to 16.3%, depending on the sampling rate, while its impact on throughput remains minimal, with a maximum reduction of 1.5%.

At Facebook, the Canopy tracing system traces billions of requests while maintaining low overhead through sampling techniques (12). Canopy's evaluation of two services (ServiceA and ServiceB) shows that instrumentation can result in an overhead of 8.15%

for ServiceA (which has shorter requests and more detailed instrumentation), while ServiceB, with longer request durations and default instrumentation, experiences only 0.76% overhead.

NanoLog, a high-performance logging system optimized for low-latency environments, achieves log invocation latencies of 8 to 18 nanoseconds (43). However, enabling instrumentation introduces a 3-4% increase in latency and a 19% reduction in throughput, highlighting the trade-off between performance and detailed observability in time-sensitive systems.

## 6.3 Comparative Analysis of Tracing Tools

Numerous distributed tracing tools exist, each with different capabilities, integration options, and performance characteristics. Popular tools such as OpenTelemetry, Jaeger, Zipkin, and Elastic APM vary significantly in the overhead they introduce and in the integration complexity with different systems (9, 30, 80).

Bento et al. (2021) compared Jaeger and Zipkin and found that while both are effective for visualizing traces and diagnosing issues, they require significant manual effort to detect performance bottlenecks (80). The study highlighted key differences between the tools, such as Zipkin's broader support for span transport technologies and Jaeger's dynamic sampling capabilities, which allows for more flexible performance tuning. The authors proposed using machine learning to automate anomaly detection, improving the usability of these tracing systems.

Janes et al. (2023) conducted a comprehensive analysis of 30 tracing tools, including Jaeger, Zipkin, Elastic APM, and OpenTelemetry (30). The study evaluated these tools based on features, language support, and popularity, emphasizing that each tool offers a unique combination of characteristics. The study concludes that the choice of a tracing tool depends heavily on the specific architecture and needs of the system, with no single tool being universally optimal.

Hindsight, introduced by Zhang et al. (2023) addresses limitations in traditional tracing approaches, such as head and tail sampling, which often misses critical edge cases or incur significant overhead (44). Hindsight's retroactive sampling records all trace data locally but only processes it when an issue is detected. This method significantly reduces the overhead of constant tracing while ensuring that rare and complex system failures

are captured efficiently. The paper compares Hindsight with OpenTelemetry and Jaeger, which both introduced significantly higher overhead in terms of throughput.

**Table 6.1:** Performance overhead comparison of various tracing tools across frameworks and metrics.

| Tracing Tool | Framework | Metric | Overhead |
|---|---|---|---|
| OpenTelemetry | Experiment 1 (Chapter 3) | Latency | 33-192% |
| Elastic APM | Experiment 1 (Chapter 3) | Latency | 17-50% |
| OpenTelemetry | Experiment 2 (Chapter 4) | Latency | 6-175% |
| Dapper (15) | Web search cluster | Latency | 16.30% |
| Canopy (12) | Unspecified ServiceA and ServiceB | Wallclock time | 0.76%, 8.15% |
| Kernel tracing (18) | Python Django | Latency | 5.10% |
| NanoLog (43) | C++ | Latency | 3-4% |
| OpenTelemetry | Experiment 1 (Chapter 3) | Throughput | 19-52% |
| Elastic APM | Experiment 1 (Chapter 3) | Throughput | 22-80% |
| OpenTelemetry (44) | Benchmark A[1] | Throughput | 42% |
| Jaeger (44) | Benchmark A[1] | Throughput | 41.70% |
| Hindsight (44) | Benchmark A[1] | Throughput | 3.50% |
| NanoLog (43) | C++ | Throughput | 19% |
| X-Trace (46) | Apache website | Throughput | 15% |

[1] Benchmark A includes DeathStar Microservices Benchmark (77), Hadoop Distributed File System (78), and Alibaba benchmark (79).

# 7

# Conclusion

In this thesis, we conducted three experiments: (1) evaluation of tracing's performance impact on request-based applications (microservices), (2) evaluation of tracing's performance impact on task-based applications (serverless) and (3) categorizing and quantifying the sources of overhead in distributed tracing. We designed and implemented the frameworks for each experiment, conducted the evaluations, and analyzed the results.

We present the following main findings from the three experiments:

**MF3.1** Distributed tracing reduces throughput across all frameworks, with declines ranging from 19.55% to 80.18%.

**MF3.2** Java Spring exhibits the lowest overhead with microservice applications, while Node.js shows the most significant impact on throughput.

**MF3.3** OpenTelemetry generally delivers higher throughput than Elastic APM across most frameworks.

**MF3.4** Distributed tracing increases median request latency for all evaluated microservices, with increases ranging from 7% to 42%.

**MF3.5** OpenTelemetry and Elastic APM introduce comparable median latency overhead, but OpenTelemetry generally results in slightly higher median latency across most microservices.

**MF3.6** The performance impact on median latency varies significantly across frameworks, with increases ranging from around 10% in some cases to as much as 179% in others.

**MF4.1** Tracing consistently increases latency across all task-based workloads, with low-latency tasks experiencing higher relative overhead percentages.

**MF4.2** Compute-intensive task-based applications, such as *graph-pagerank*, show lower percentage overhead (6.69%) despite substantial absolute latency increases.

**MF4.3** As baseline non-instrumented request duration increases, the overhead percentage generally decreases, indicating a relatively reduced impact.

**MF4.4** With serverless applications, Node.js framework generally shows a higher percentage overhead than Python, especially for low-latency tasks.

**MF4.5** Despite higher overhead percentages, the absolute overhead values for serverless applications in Python and Node.js are comparable.

**MF5.1** Configuration overhead is a major contributor in cold-start scenarios for serverless applications.

**MF5.2** Instrumentation overhead remains minimal for serverless applications but is relatively high in request-based applications.

**MF5.3** Exporting the trace data consistently incurs the highest overhead in request-based applications.

**MF5.4** Exporting causes a considerable overhead in short-duration serverless applications.

## 7.1 Research Questions

**RQ1: How does the implementation of distributed tracing affect the throughput and latency of request-based applications?** Our experiment shows that distributed tracing significantly impacts throughput and latency in request-based applications. We observe that the throughput can decrease from 19.55% to 80.18%. Regarding latency, tracing consistently adds considerable delays by increasing the median request duration by 7% to 42%. These findings highlight the trade-off between observability and the performance cost of distributed tracing.

**RQ2: What are the effects of distributed tracing instrumentation on the performance of task-based applications?** The serverless evaluation experiment reveals that enabling tracing adds measurable latency across all workloads. With short-duration benchmarks (2.53 ms), we observe latency increases up to 175%. For the benchmarks with a moderate duration ( 25 ms and 112 ms), distributed tracing added approximately 8-13% of additional latency. Longer-running benchmarks ( 4500 ms) exhibit a lower relative increase in latency of around 6.7%, but the absolute latency increase is significantly higher. These results confirm that distributed tracing introduces a considerable overhead, with more pronounced effects on short-duration tasks than benchmarks with higher baseline latency.

**RQ3: What are the primary sources of overhead introduced by distributed tracing, and which of them are the main contributors to overhead?** We first identified three main categories in distributed tracing: configuration, instrumentation, and export. Configuration includes the initial setup of the tracing system. The experiment reveals that configuration overhead significantly contributes to short-duration cold-start serverless applications, which consume a significant proportion of the execution time. The instrumentation category consists of operations, which insert trace points and enrich the data. We observe that instrumentation provides minimal impact on serverless applications but a considerable amount ( 8-12%) on request-based applications. Exporting involves transmitting the data to external storage or backend. We found that exporting is consistently one of the largest sources of overhead, especially in request-based applications.

# Appendix

## A Results Data

The datasets used in this thesis, including performance metrics, profiling data, and experiment configurations, are available for access and download. The data is organized and described in a GitHub repository, which contains links to the files hosted on Google Drive. The download links are described in the GitHub page:
https://github.com/andersnou/msc-thesis/blob/master/ResultsData.md.

## B Task-based application experiment benchmark execution record


```
{
  "billing": {
    "_billed_time": null,
    "_gb_seconds": 0,
    "_memory": null
  },
  "output": {
    "begin": "1720219857.361634",
    "end": "1720219857.367960",
    "is_cold": false,
    "request_id": "00142209237343789422092373037861",
    "result": {
      "measurement": {
        "compute_time": 57,
        "graph_generating_time": 469
```

```
      },
      "result": 0.1
    },
    "results_time": 6326
  },
  "provider_times": {
    "execution": 0,
    "initialization": 0
  },
  "request_id": "00142209237343789422092373037861",
  "stats": {
    "cold_start": false,
    "failure": false,
    "memory_used": null
  },
  "times": {
    "benchmark": 6326,
    "client": 69912,
    "client_begin": "2024-07-06 00:50:57.309980",
    "client_end": "2024-07-06 00:50:57.379892",
    "http_first_byte_return": 0.069829,
    "http_startup": 0.031384,
    "initialization": 0
  }
}
```


## C Task-based application *graph-pagerank* manual instrumentation code example

```
import igraph
import os

from opentelemetry import trace
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanExport
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import SimpleSpanProcessor
```

```
from opentelemetry.sdk.resources import Resource

otlp_exporter = OTLPSpanExporter(
    endpoint=os.getenv("OTLP_EXPORT_ENDPOINT", "http://localhost:4317"),
    insecure=True
)

span_processor = SimpleSpanProcessor(otlp_exporter)

resource = Resource(attributes={
    "service.name": "660.graph-pagerank-opentelemetry"
})
trace.set_tracer_provider(TracerProvider(resource=resource))
trace.get_tracer_provider().add_span_processor(span_processor)

tracer = trace.get_tracer("handler")


def handler(event):
    span = tracer.start_span("handler")
    ctx = trace.set_span_in_context(span)

    size = event.get('size')
    span.set_attribute("size", size)

    generate_graph_span = tracer.start_span("generate_graph", context=ctx)
    graph = igraph.Graph.Barabasi(size, 10)
    generate_graph_span.end()

    pagerank_span = tracer.start_span("pagerank", context=ctx)
    result = graph.pagerank()
    pagerank_span.end()

    first_node_rank = result[0]
    span.set_attribute("first_node_rank", first_node_rank)

    span.end()
    return {
        'result': result[0]
    }
```

# References

[1] José Fonseca. **gprof2dot: Python script to convert the output from many profilers into a dot graph**. `https://github.com/jrfonseca/gprof2dot`. Accessed: 2024-08-28. iv, 75, 77

[2] Garrett McGrath and Paul R Brenner. **Serverless computing: Design, implementation, and performance**. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. IEEE, 2017. 1

[3] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. **Cloud programming simplified: A berkeley view on serverless computing**. *arXiv preprint arXiv:1902.03383*, 2019. 1

[4] Ioana Baldini, Paul C. Castro, K. Chang, P. Cheng, Stephen J. Fink, Vatche Isahagian, N. Mitchell, Vinod Muthusamy, R. Rabbah, Aleksander Slominski, and Philippe Suter. **Serverless Computing: Current Trends and Open Problems**. *ArXiv*, **abs/1706.03178**, 2017. 1

[5] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. **Serverless computing: a survey of opportunities, challenges, and applications**. *ACM Computing Surveys*, **54**(11s):1–32, 2022. 1

[6] Gojko Adzic and Robert Chatley. **Serverless computing: economic and architectural impact**. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 884–889, 2017. 1

[7] Raja R Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R Ganger. **So, you want to trace your distributed system? Key design insights from years of practical experience**. *Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-PDL*, **14**, 2014. 1

[8] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices.* O'Reilly Media, 2020. 1

[9] Jonathan Mace. **End-to-End Tracing: Adoption and Use Cases**. Survey, Brown University, 2017. 1, 11, 91

[10] Eric Anderson, Christopher Hoover, Xiaozhou Li, and Joseph Tucek. **Efficient tracing and performance analysis for large distributed systems**. In *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 1–10. IEEE, 2009. 1, 70, 72

[11] Bo Sang, Jianfeng Zhan, Gang Lu, Haining Wang, Dongyan Xu, Lei Wang, Zhihong Zhang, and Zhen Jia. **Precise, Scalable, and Online Request Tracing for Multitier Services of Black Boxes**. *IEEE Transactions on Parallel and Distributed Systems*, **23**:1159–1167, 2010. 1

[12] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. **Canopy: An end-to-end performance tracing and analysis system**. In *Proceedings of the 26th symposium on operating systems principles*, pages 34–50, 2017. 1, 6, 12, 22, 23, 29, 49, 90, 93

[13] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. **Fay: Extensible distributed tracing from kernels to clusters**. *ACM Transactions on Computer Systems (TOCS)*, **30**(4):1–35, 2012. 1, 6, 12, 48, 72, 90

[14] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, et al. **Network-centric distributed tracing with deepflow: Troubleshooting**

**your microservices in zero code**. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 420–437, 2023. 1, 49, 72

[15] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. **Dapper, a large-scale distributed systems tracing infrastructure**. 2010. 1, 2, 4, 12, 22, 23, 29, 48, 49, 71, 90, 93

[16] Loïc Gelle, Naser Ezzati-Jivan, and Michel R Dagenais. **Combining distributed and kernel tracing for performance analysis of cloud applications**. *Electronics*, **10**(21):2610, 2021. 1, 49

[17] Zoltán Zvara, Péter GN Szabó, Barnabás Balázs, and András Benczúr. **Optimizing distributed data stream processing by tracing**. *Future Generation Computer Systems*, **90**:578–591, 2019. 2

[18] Francis Giraldeau and Michel Dagenais. **Wait analysis of distributed systems using kernel tracing**. *IEEE Transactions on Parallel and Distributed Systems*, **27**(8):2450–2461, 2015. 2, 29, 93

[19] Betsy Beyer, Niall Richard Murphy, David K Rensin, Kent Kawahara, and Stephen Thorne. *The site reliability workbook: practical ways to implement SRE*. " O'Reilly Media, Inc.", 2018. 2

[20] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: How Google runs production systems*. " O'Reilly Media, Inc.", 2016. 2

[21] Pedro Las-Casas, Jonathan Mace, Dorgival Guedes, and Rodrigo Fonseca. **Weighted sampling of execution traces: Capturing more needles and less hay**. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 326–332, 2018. 4, 12

[22] Joel Scheuner, Simon Eismann, Sacheendra Talluri, Erwin Van Eyk, Cristina Abad, Philipp Leitner, and Alexandru Iosup. **Let's Trace It: Fine-Grained Serverless Benchmarking using Synchronous and Asynchronous Orchestrated Applications**. *arXiv preprint arXiv:2205.07696*, 2022. 6, 29

[23] Jörn Kuhlenkamp, Sebastian Werner, Maria C. Borges, Dominik Ernst, and Daniel Wenzel. **Benchmarking elasticity of FaaS platforms as a foundation for objective-driven design of serverless applications**. *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020. 6

[24] Ron Kohavi, R. Longbotham, D. Sommerfield, and Randal M. Henne. **Controlled experiments on the web: survey and practical guide**. *Data Mining and Knowledge Discovery*, **18**:140–181, 2009. 6

[25] S. Barker and Prashant J. Shenoy. **Empirical evaluation of latency-sensitive application performance in the cloud**. pages 35–46, 2010. 6

[26] Maria C Borges, Sebastian Werner, and Ahmet Kilic. **Faaster troubleshooting-evaluating distributed tracing approaches for serverless applications**. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 83–90. IEEE, 2021. 6, 48

[27] **Open Source Serverless Cloud Platform**. `https://openwhisk.apache.org/`. Accessed: 2024-07-29. 6

[28] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. **Sebs: A serverless benchmark suite for function-as-a-service computing**. In *Proceedings of the 22nd International Middleware Conference*, pages 64–78, 2021. 8, 16, 50, 52, 78

[29] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. **Fear and logging in the internet of things**. In *Network and Distributed Systems Symposium*, 2018. 12

[30] Andrea Janes, Xiaozhou Li, and Valentina Lenarduzzi. **Open tracing tools: Overview and critical comparison**. *Journal of Systems and Software*, page 111793, 2023. 12, 91

[31] **What is OpenTelemetry?** `https://opentelemetry.io/docs/what-is-opentelemetry/`. Accessed: 2024-09-21. 14

[32] **OpenTelemetry Collector**. `https://opentelemetry.io/docs/collector/`. Accessed: 2024-09-21. 14

[33] **Application Performance Monitoring (APM)**. https://www.elastic.co/observability/application-performance-monitoring. Accessed: 2024-05-20. 14

[34] **Elastic Stack**. https://www.elastic.co/elastic-stack. Accessed: 2024-09-22. 15

[35] Karim Djemame, Matthew Parker, and Daniel Datsev. **Open-source serverless architectures: an evaluation of apache openwhisk**. In *2020 ieee/acm 13th international conference on utility and cloud computing (ucc)*, pages 329–335. IEEE, 2020. 15, 16

[36] Nima Kaviani, Dmitriy Kalinin, and Michael Maximilien. **Towards serverless as commodity: A case of knative**. In *Proceedings of the 5th International Workshop on Serverless Computing*, pages 13–18, 2019. 15, 16

[37] **SeBS: Serverless Benchmark Suite**. https://github.com/spcl/serverless-benchmarks/tree/master. Accessed: 2024-06-15. 16, 50, 60, 78

[38] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. *Kubernetes: up and running.* " O'Reilly Media, Inc.", 2022. 17

[39] Kai Lei, Yining Ma, and Zhi Tan. **Performance comparison and evaluation of web development technologies in php, python, and node. js**. In *2014 IEEE 17th international conference on computational science and engineering*, pages 661–668. IEEE, 2014. 22, 49

[40] Christina Eder, Stefan Winzinger, and Robin Lichtenthäler. **A Comparison of Distributed Tracing Tools in Serverless Applications**. In *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 98–105. IEEE, 2023. 23

[41] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. **Overhead comparison of opentelemetry, inspectit and kieker**. 2021. 23

[42] **Discover, iterate, and resolve with ES|QL on Kibana**. https://www.elastic.co/kibana. Accessed:2024-07-22. 24

[43] Stephen Yang, Seo Jin Park, and John Ousterhout. **{NanoLog}: A Nanosecond Scale Logging System**. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 335–350, 2018. 29, 91, 93

[44] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. **The Benefit of Hindsight: Tracing {Edge-Cases} in Distributed Systems**. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 321–339, 2023. 29, 91, 93

[45] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. **Stardust: tracking activity in a distributed storage system**. *ACM SIGMETRICS Performance Evaluation Review*, **34**(1):3–14, 2006. 29, 90

[46] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. **{X-Trace}: A pervasive network tracing framework**. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, 2007. 29, 93

[47] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. **Sifter: Scalable sampling for distributed traces, without feature engineering**. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 312–324, 2019. 29, 70, 71, 89, 90

[48] Fahd Al-Haidari, Mohammed Sqalli, and Khaled Salah. **Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources**. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, **2**, pages 256–261. IEEE, 2013. 29

[49] Marjan Gusev, Sasko Ristov, Monika Simjanoska, and Goran Velkoski. **Cpu utilization while scaling resources in the cloud**. *Cloud Computing*, pages 131–137, 2013. 29

[50] **Project Information Framework Tests Overview**. `https://github.com/TechEmpower/FrameworkBenchmarks/wiki/Project-Information-Framework-Tests-Overview`. Accessed: 2024-05-05. 31, 37

[51] **[flask] Flask is a web application framework for python.** https://stackoverflow.com/questions/tagged/flask. Accessed: 2024-09-28. 32

[52] **[spring] The Spring Framework is an open-source application framework for the Java platform.** https://stackoverflow.com/questions/tagged/spring. Accessed: 2024-09-28. 32

[53] **[node.js] Node.js is an event-based, non-blocking, asynchronous I/O (input/output) runtime that uses Google's V8 JavaScript Engine.** https://stackoverflow.com/questions/tagged/node.js. Accessed: 2024-09-28. 32

[54] Ioannis K Chaniotis, Kyriakos-Ioannis D Kyriakou, and Nikolaos D Tselikas. **Is Node. js a viable option for building modern web applications? A performance evaluation study**. *Computing*, **97**:1023–1044, 2015. 32

[55] **OpenWhisk System Overview**. https://github.com/apache/openwhisk/blob/master/docs/about.md. Accessed: 2024-06-13. 54

[56] Sergey Brin and Lawrence Page. **The anatomy of a large-scale hypertextual web search engine**. *Computer networks and ISDN systems*, **30**(1-7):107–117, 1998. 55

[57] **OpenWhisk Helm Chart**. https://github.com/apache/openwhisk-deploy-kube/tree/master/helm/openwhisk. Accessed: 2024-06-14. 60

[58] Nicolae Marian Popa and Ana Oprescu. **A data-centric approach to distributed tracing**. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 209–216. IEEE, 2019. 70

[59] Ping-Min Lin and Alex Glikson. **Mitigating cold starts in serverless platforms: A pool-based approach**. *arXiv preprint arXiv:1903.12221*, 2019. 71

[60] Alexander Fuerst and Prateek Sharma. **FaasCache: keeping serverless computing alive with greedy-dual caching**. *Proceedings of the 26th ACM*

*International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021. 71

[61] **OpenTelemetry Tracing SDK**. https://opentelemetry.io/docs/specs/otel/trace/sdk/. Accessed: 2024-09-01. 71

[62] Matheus Santana, Adalberto Sampaio Jr, Marcos Andrade, and Nelson S Rosa. **Transparent tracing of microservice-based applications**. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1252–1259, 2019. 72

[63] **Python interface for the 'lsprof' profiler. Compatible with the 'profile' module.** https://github.com/python/cpython/blob/main/Lib/cProfile.py. Accessed:2024-08-26. 73

[64] **The Python Profilers.** https://docs.python.org/3/library/profile.html#module-cProfile. Accessed:2024-08-26. 73, 74

[65] **Line-granularity, thread-aware deterministic and statistic pure-python profiler.** https://github.com/vpelletier/pprofile/tree/master. Accessed:2024-08-26. 73, 74

[66] **py-spy: Sampling profiler for Python programs.** https://github.com/benfred/py-spy. Accessed:2024-08-26. 73, 74

[67] Emery D. Berger, Sam Stern, and Juan Altmayer Pizzorno. **Triangulating Python Performance Issues with Scalene**. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 51–64, Boston, MA, July 2023. USENIX Association. 73, 74

[68] **Scalene: a high-performance, high-precision CPU, GPU, and memory profiler for Python with AI-powered optimization proposals.** https://github.com/plasma-umass/scalene. Accessed:2024-08-26. 73

[69] Brendan Gregg. **The flame graph**. *Communications of the ACM*, **59**(6):48–57, 2016. 75

[70] Anton Bobrov. **Flamegraph generator for cProfile stats**. `https://github.com/baverman/flameprof`. Accessed: 2024-08-28. 75

[71] Emden Gansner, Eleftherios Koutsofios, and Stephen North. **Drawing graphs with dot**, 2006. 75

[72] **Web Framework Benchmarks**. `https://www.techempower.com/benchmarks`. Accessed: 2024-09-05. 77

[73] David Reis, Bruno Piedade, Filipe F Correia, João Pedro Dias, and Ademar Aguiar. **Developing docker and docker-compose specifications: A developers' survey**. *Ieee Access*, **10**:2318–2329, 2021. 80

[74] **Docker Compose overview**. `https://docs.docker.com/compose/`. Accessed: 2024-09-04. 80

[75] **Network Overview, User-defined networks**. `https://docs.docker.com/engine/network/#user-defined-networks`. Accessed: 2024-09-02. 81

[76] Ian Miell and Aidan Sayers. *Docker in practice*. Simon and Schuster, 2019. 81

[77] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. **An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems**. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019. 93

[78] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. **The hadoop distributed file system**. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010. 93

[79] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. **Characterizing microservice dependency and performance: Alibaba trace analysis**. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021. 93

[80] Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardoso. **Automated analysis of distributed tracing: Challenges and research directions**. *Journal of Grid Computing*, **19**(1):9, 2021. 91