# OpenTelemetry Tracing Overhead in a Single Go Service on Kubernetes

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Telecommunications and Cybersecurity Technology
May 2026
Ville-Pekka Hoikka

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Computing

Ville-Pekka Hoikka: OpenTelemetry Tracing Overhead in a Single Go Service on Kubernetes

Master of Science (Tech) Thesis, 68 p., 1 app. p.
Telecommunications and Cybersecurity Technology
May 2026

---

This thesis measures the computational resource cost of OpenTelemetry distributed tracing when running a single instrumented Go service on Kubernetes. The focus is on CPU, memory, latency and network overhead and on how different sampling strategies change those costs. The diagnostic value obtained from each strategy and the privacy and security trade-offs of different configurations are also examined.
A Go service with five different workload endpoints was built and experiments were run on a managed Kubernetes cluster at three load levels. Prometheus was used to collect resource and performance metrics during the test runs. Trace data was exported to Grafana Tempo for span count validation and sampling rate confirmation.
The results show that tracing overhead is predictable and scales with the sampling rate. Even at 100% sampling the system stayed stable. In this single-service setup, head-based and parent-based sampling performed the same across all workloads. In multi-service deployments the difference moves more to architectural questions since parent-based sampling keeps traces consistent across services without noticeable resource overhead. A custom adaptive sampler was also implemented in the Go service that adjusts its rate based on observed error counts and latency. It did not add measurable overhead to the service and captured a larger batch of relevant requests and traces. Overhead from this strategy was stable across all configurations.
The privacy and security analysis shows that at low overhead levels, sampling rate becomes more of a data governance decision: what accumulates in the tracing backend and who can access it depends on configuration choices rather than technical constraints.

Keywords: OpenTelemetry, distributed tracing, Go, Kubernetes, sampling strategies, performance overhead, observability

# Contents

**1 Introduction** **1**

1.1 Background . . . 1

1.2 Problem Statement . . . 2

1.3 Scope and Objectives . . . 2

1.4 Thesis Structure . . . 3

1.5 Use of AI . . . 3

**2 Observability and tracing in microservices** **4**

2.1 Microservice Architecture and Kubernetes . . . 4

2.1.1 Kubernetes orchestrator & microservices . . . 5

2.1.2 System Complexity and Observability . . . 5

2.2 General items of Observability & Monitoring . . . 6

2.2.1 Logs, Metrics and Traces . . . 7

2.2.2 Observability Stack . . . 9

2.3 Tracing and Span Structure . . . 10

2.3.1 Span and Trace Structure . . . 10

2.3.2 Context Propagation . . . 11

2.4 Sampling Strategies . . . 12

2.4.1 Choosing a Strategy . . . 14

2.5 Privacy and Security Considerations in Telemetry . . . 15

2.5.1 Data Privacy . . . 15
2.5.2 Security . . . 16
2.6 Related Work . . . 17
2.7 Closing summary . . . 20

**3 Research Design** **21**
3.1 Research Questions . . . 21
3.2 Experimental Factors and Variables . . . 22
3.2.1 Factors in the experiment . . . 22
3.2.2 Measured outcomes . . . 23
3.2.3 Conditions . . . 23
3.2.4 Experiment Matrix and Run Order . . . 24
3.3 Metrics and Data Collection . . . 24
3.3.1 Service-Level Metrics . . . 25
3.3.2 System-Level Metrics . . . 25
3.3.3 Load Generator Metrics . . . 26
3.3.4 Trace Backend Validation . . . 26
3.3.5 Data Processing and Quality Control . . . 26
3.3.6 Summary of Metrics . . . 27

**4 Experimental Setup** **28**
4.1 System Under Test . . . 28
4.1.1 Service Endpoints . . . 28
4.1.2 Tracing Configuration . . . 29
4.1.3 Deployment Configuration . . . 30
4.2 Environment and Infrastructure . . . 31
4.2.1 Kubernetes Cluster . . . 31
4.2.2 Observability Stack . . . 32

4.2.3 Version Control and Deployment . . . . . . . . . . . . . . . . . 33
4.3 Workload Generation and Test Execution . . . . . . . . . . . . . . . . . 34
4.3.1 Load Generator Configuration . . . . . . . . . . . . . . . . . . . . 34
4.3.2 Test Execution Timeline . . . . . . . . . . . . . . . . . . . . . . . 35
4.3.3 Request Patterns and Endpoint Isolation . . . . . . . . . . . . 35
4.3.4 Load Generator Deployment . . . . . . . . . . . . . . . . . . . . 36
4.4 Sampling Strategy Implementation . . . . . . . . . . . . . . . . . . . . 36
4.4.1 Baseline Configuration (No Tracing) . . . . . . . . . . . . . . . 37
4.4.2 Head Sampling Configurations . . . . . . . . . . . . . . . . . . . 37
4.4.3 Parent-Based Sampling Configuration . . . . . . . . . . . . . . . 38
4.4.4 Adaptive Sampling Configuration . . . . . . . . . . . . . . . . . 39
4.4.5 Configuration Matrix and Test Combinations . . . . . . . . . 40

**5 Results and Analysis** **42**
5.1 Research Question 1 . . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.1.1 Endpoint Selection . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.1.2 CPU Overhead . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.1.3 Request Latency . . . . . . . . . . . . . . . . . . . . . . . . . . 45
5.2 Research Question 2 . . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.2.1 Scope and Context . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.2.2 Experiment Overview . . . . . . . . . . . . . . . . . . . . . . . . 47
5.2.3 CPU Usage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.2.4 Request Latency . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.2.5 Memory Usage . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.2.6 Network Egress . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.2.7 Throughput . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.2.8 Trace Completeness . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.2.9 Summary and Implications . . . . . . . . . . . . . . . . . . . . . 52

5.3 Research Question 3 . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.3.1 Effective Sampling Behavior . . . . . . . . . . . . . . . . . . . . . 52
5.3.2 Span Volume and Backend Impact . . . . . . . . . . . . . . . . 53
5.3.3 Application-Level Performance . . . . . . . . . . . . . . . . . . . . 54
5.3.4 Diagnostic Coverage . . . . . . . . . . . . . . . . . . . . . . . . . 56
5.3.5 Feasibility Assessment . . . . . . . . . . . . . . . . . . . . . . . . 57
5.4 Research Question 4 . . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.4.1 Sampling Rate and Risk of Data Exposure . . . . . . . . . . . 59
5.4.2 Adaptive Sampling and Sensitive Events in Service . . . . . . 59
5.4.3 Centralized Trace Storage and Multi-Tenant Risk . . . . . . . 60
5.4.4 Context Propagation and Topology Exposure . . . . . . . . . 60
5.4.5 Mitigation Mechanisms . . . . . . . . . . . . . . . . . . . . . . . 61
5.4.6 Overall Trade-off Assessment . . . . . . . . . . . . . . . . . . . 62

**6 Discussion and Conclusions** **63**
6.1 Summary of Findings . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.2 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
6.3 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
6.4 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
6.5 Concluding Remarks . . . . . . . . . . . . . . . . . . . . . . . . . . 68

**References** **69**

**Appendices**

**A Technical Material** **A-1**

# List of Figures

5.1 CPU overhead for api-basic endpoint with configured sampling rates. Figure shows percentage increase in CPU usage relative to baseline (0% sampling) at different load levels (50, 500, 1000 RPS). . . . . . . 43

5.2 CPU overhead for api-cpu-intensive endpoint with configured sampling rates. Despite higher CPU usage, overhead percentage follows the same pattern as api-basic endpoint. . . . . . . . . . . . . . . . . . . 44

5.3 p95 and p99 request latency for *api-basic* from all configured sampling rates and load levels. Both percentiles stay close to the no-tracing baseline regardless of configurations. . . . . . . . . . . . . . . . . . . . 45

5.4 p95 and p99 request latency for *api-cpu-intensive* all configured sampling rates and load levels. At the highest load and sampling rate combinations, p99 rises slightly as the service approaches CPU-usage limitation but the increase stays consistent between test runs. . . . . 46

5.5 CPU usage as a function of sampling rate for the *api-basic* workload at 50, 500 and 1000 requests per second, comparing head-based and parent-based sampling. . . . . . . . . . . . . . . . . . . . . . . . . . 48

5.6 CPU usage as a function of sampling rate for the *api-cpu-intensive* workload at 50, 500 and 1000 requests per second. Application CPU dominates over any sampling strategy effects. . . . . . . . . . . . . . 49

5.7 p95 and p99 request latency for the *api-cpu-intensive* workload across sampling rates and load levels. Both strategies show the same tail-latency behavior. . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.8 Achieved throughput for the *api-basic* workload under increasing sampling rates and load levels. Head-based and parent-based sampling show comparable capacity. . . . . . . . . . . . . . . . . . . . . . . . . 51
5.9 Spans ingested per second for adaptive and head-based sampling at 500 requests per second. Each panel shows a separate endpoint workload. Adaptive sampling produces more spans at every rate above 1%, with the gap widening consistently as the base rate increases. . . 53
5.10 CPU usage and span yield for adaptive versus head-based sampling at 500 requests per second, averaged across all endpoints. The left panel shows that CPU cost in cores stays nearly identical between strategies at every sampling rate. The right panel shows the span yield advantage of adaptive sampling, which reaches approximately 50% at rates of 10% and above. . . . . . . . . . . . . . . . . . . . . . . 54
5.11 CPU usage in cores for adaptive and head-based sampling at 500 requests per second, split by endpoint. Both strategies follow each other closely within each workload group across all tested sampling rates. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
5.12 p95 request latency for adaptive and head-based sampling at 30% sampling rate and 500 requests per second. The bars are equal across all four endpoints, confirming that adaptive sampling adds no measurable latency overhead compared to head-based sampling. . . . . . 56

5.13 CPU usage in cores averaged across endpoints as a function of sampling rate at 50, 500 and 1000 requests per second. Adaptive and head-based sampling remain comparable across all load levels, showing that overhead does not change radically with the applied request rates. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

# List of Tables

3.1 Summary of metrics collected during experiments. . . . . . . . . . . . . 27

4.1 Experiment configuration matrix. . . . . . . . . . . . . . . . . . . . . . 41

# 1 Introduction

## 1.1 Background

Modern software is increasingly built using microservice architectures where applications are split into small services that communicate over a network. Platforms such as Kubernetes handle the orchestration of these services, meaning managing deployment, scaling, and operation. This approach gives developers flexibility, but it also makes it harder to see what is happening inside the system. A single user request can travel through multiple services on different machines and when failures occur it is not always easy to identify where the problem originated.

Distributed tracing addresses this challenge. It tracks requests as they move through services and records timing and metadata at each step. This gives developers and system administrators a way to see the full path of a request and identify where requests slow down or fail. OpenTelemetry is a popular vendor-neutral framework for implementing this kind of tracing, supported by a variety of backends and tools without locking users into a single platform.

## 1.2 Problem Statement

Despite the growing adoption of distributed tracing, its actual resource cost in production-like environments is rarely measured in a systematic way. Tracing is commonly enabled because it is considered best practice, but the overhead that tracing introduces on CPU, memory, latency and network usage is not well documented for single-service Go deployments on Kubernetes. Sampling strategies are used to control how much trace data is collected but their performance differences are not well documented. Available documentation is either general or based on benchmarks that do not reflect single service configurations.

It is also worth noting that higher sampling rates and centralized trace storage introduce privacy and security risks that are easily overlooked when talking about observability. It is also hard to find any adaptive sampling strategies that can adjust their sampling rate based on runtime conditions such as error rates or latency. It should be evaluated whether such strategies are feasible without adding measurable overhead to a service.

## 1.3 Scope and Objectives

This thesis measures the performance overhead of OpenTelemetry tracing in a single Go service deployed on Kubernetes. Three sampling strategies are compared: head-based, parent-based and a custom adaptive sampler built for this thesis. The adaptive sampler adjusts its rate based on observed errors and latency. The experiments evaluate if dynamic rate adjustment is feasible without considerable overhead. The privacy and security aspects of different tracing configurations are also studied based on experimental results and existing literature.

## 1.4 Thesis Structure

The thesis is structured as follows: Chapter 2 covers the background on microservices and observability in systems. Chapter 3 defines the research questions and experiment design. Chapter 4 describes the test service, the infrastructure, the system and the measurement setup. Chapter 5 presents the results for each research question. Chapter 6 presents the discussion and conclusions.

## 1.5 Use of AI

The Claude Sonnet 4.6 language model was used during the writing of this thesis for proofreading and checking language correctness. This included fixing grammar and improving the sentence structure for better readability. The technical implementations were developed by the author but were reviewed by the same tool for any errors. The research design, experimental setup, infrastructure configuration and interpretation of the experiment results were made by the author.

# 2 Observability and tracing in microservices

Modern software systems are increasingly built with microservice architectures in mind. Microservice architecture structures applications as small, independently deployable services that communicate over a network. Microservices divide an application into smaller, autonomous units. Each unit is responsible for a specific function or a transaction between said units. Microservices communicate with each other using mechanisms such as RESTful or RPC-based APIs [1]. This design allows the services to be developed, deployed and scaled independently within the overall system. It also enhances modularity, flexibility and resilience, making it a suitable choice for delivering and maintaining modern software. However, it also introduces new challenges in system management, monitoring and debugging by adding extra layers of complexity. [2]–[4].

## 2.1 Microservice Architecture and Kubernetes

To orchestrate these distributed services, teams use container orchestrators. One of the most widely adopted is Kubernetes. Kubernetes is a container orchestration platform originally developed by Google in 2014. It was later released as an open-source project and is now developed and maintained by a large active community under the Cloud Native Computing Foundation (CNCF). [5]–[7]

### 2.1.1 Kubernetes orchestrator & microservices

The main function of Kubernetes is to automate deployment, scaling and keeping containerized applications running. It handles multiple aspects within a cluster, including service discovery, load balancing, resource allocation and automatic restarts of failed components. This ensures that the system maintains its desired state. These capabilities allow Kubernetes to maintain high availability in environments where services are constantly changing. It is also flexible and can be adapted to specific workloads. Since Kubernetes is an open-source project, it has active development and maintenance from a large group of maintainers. This means extensive support, documentation and a broad ecosystem of supporting tools [8].

### 2.1.2 System Complexity and Observability

The benefits of microservices and platforms like Kubernetes come at the cost of increased system complexity. Since most tasks in the application are split into individual services, the execution path of a single request often spans multiple containers, hosts, or even cloud regions. This makes it harder to trace failures within the architecture and complicates understanding system behavior without the help of dedicated observability and monitoring tooling. Before the widespread adoption of distributed architectures, software systems were commonly implemented as monolithic applications.

In a monolithic design, the entire application is built as a single, unified codebase that runs as one process. All core components such as business logic, data access and user interfaces are tightly coupled within the same process. This centralized structure simplifies tasks like debugging and performance monitoring, as developers can trace the execution of a request from start to finish using conventional logging or profiling tools within a single runtime environment. However, monolithic systems are cumbersome to scale and are prone to full application crashes if a single component

fails.

Microservice architectures improve scalability and flexibility by separating application components but it also makes it harder to trace requests, detect performance bottlenecks or identify failure points in applications. A single user interaction might involve dozens of services operating across different machines or cloud regions. This complicates visibility on how the system behaves under load or during failures. [2], [9]

This complexity makes the ability to observe and understand system behavior a critical requirement, leading to the need for dedicated observability and monitoring tooling. In Kubernetes-based environments, services often start and stop quickly and the system scales workloads up or down based on demand and resource usage. This means that parts of the application may only run for a short time or move between servers, depending on how the system is being used. Because of this it can be difficult to monitor what is happening in the system at any given time, especially when trying to find the root cause of a problem. Tracing makes it possible to follow individual requests across multiple services and find performance or reliability issues. [10]

## 2.2 General items of Observability & Monitoring

Understanding what is happening inside an application can be difficult without proper visibility and tools. This is where the concept of observability becomes useful. Observability refers to the ability to understand what is happening inside a system by analyzing the data it produces during operation. This includes information such as logs, metrics and traces. Together they help reveal the system's internal state, especially when something goes wrong. Unlike simple monitoring that typically focuses on predefined alerts and metrics, observability gives engineers enough data to diagnose unexpected problems based on what the application produces. [11], [12]

### 2.2.1 Logs, Metrics and Traces

Observability has three main components, each giving a different view of the system: logs, metrics and traces. These help developers and operators understand how the system is functioning, identify problems and respond to incidents. [11]

**Logs** Logs are timestamped records of events that occur within an application. They often include contextual information useful for diagnosing errors or understanding behavior during specific points in time. Developers can add their own data to logs to help find and determine errors in services. In microservice architectures, logs are usually collected from each container and centralized using tools like Loki, Fluent Bit, or the ELK stack, enabling system-wide querying and correlation. Logs are typically shipped to a central location using lightweight agents or log forwarders, which format, buffer and transmit the logs over the network to ensure reliable delivery even under high load or temporary network disruptions. [11]

**Metrics** Metrics are numeric representations of system performance and resource usage over time. Metrics are usually collected from the service itself or from the system running and supporting the service. Examples include CPU utilization, memory consumption, response times, request counts and error rates. Metrics are typically exposed as time-series data and are well suited for dashboards, trend analysis and alerting. Tools like Prometheus and Grafana are widely used for collecting and visualizing these metrics. [13]

**Traces** Traces follow the lifecycle of individual requests as they travel through multiple services in a distributed system. Each segment of a trace is called a span, which contains metadata such as duration, parent-child relationships and annotations. Like with logs, developers can configure what is exported in traces. Tracing helps uncover latency bottlenecks, failed service calls, or de-

pendencies that contribute to system instability. Tracing is especially valuable in Kubernetes-based microservices, where a single request might pass through several short-lived services that can run on different servers or even in different cloud locations. [10], [14]

While each of the components offers unique insights, incorporating all three into the same stack will enable sufficient observability of services. For example, a trace might reveal slow response times between services, a metric could show high CPU usage on a specific server and logs might explain what caused the spike. Together they provide causality and help identify the root cause of the problem. This combination is particularly important in large-scale tracing deployments where correlation across services is required [15].

Monitoring complements observability by defining known failure conditions or performance thresholds, often through static alerts. These can be defined by system operators and alerts can be triggered based on logs, metrics, or traces. For example, an alert could be fired when a specific log line appears, when CPU usage remains high or when a trace shows an error in a request flow. Compared to monitoring, observability helps explore unexpected or new issues, especially in systems where it is difficult to predict how failures might occur. It also helps investigate the root cause behind alerts triggered by monitoring. [11], [14]

### 2.2.2 Observability Stack

Observability is usually integrated into microservice-based systems to support monitoring, debugging and performance optimization in multi-service and complex environments. Modern practice in software development utilizes instrumentation to give developers and operators access to data for maintaining and securing complex applications. [11]

Observability stacks typically consist of a set of specialized components. Each component is designed to handle a particular type of telemetry data. They collect and send observability data to consumers such as dashboards, alerting systems and incident response tools. [13], [16]

**Metrics collector** gathers numerical time-series data from services and infrastructure. It scrapes or receives data at regular intervals and stores it in a format optimized for querying and alerting. [13]

**Log aggregator** collects log data from multiple sources, centralizes it for indexing and search and may also parse, enrich or route it to storage.

**Trace collector** receives trace data from instrumented applications and assembles spans into end-to-end traces for analysis and visualization. [16]

**Visualization** layer provides dashboards and query interfaces for exploring the collected data which helps operators and administrators respond to events in the system. [13], [17]

These components are typically deployed as services within the observability infrastructure and work together to give visibility into distributed systems. In addition to operational benefits, observability can also support security by helping detect anomalies, unauthorized access patterns, or abnormal service behaviors. When telemetry is tagged with context and monitored in real-time, it becomes easier to investigate incidents and trace potential vulnerabilities across services. The workload design draws on prior microservice benchmarking work on how instrumentation

interacts with load [11], [18].

## 2.3 Tracing and Span Structure

As explained in section 2.2, traces are an essential part of observability in distributed systems. They offer insight into how individual requests move through the various components of an application and the system. This is particularly valuable in microservice environments, where a single request might involve multiple services and layers of infrastructure. Without tracing, it becomes difficult to pinpoint where bottlenecks, service failures or unexpected behavior happen. [11], [14]

### 2.3.1 Span and Trace Structure

At its core, tracing involves capturing the flow of a request across services and representing it as a series of connected operations, called spans. A span corresponds to a single unit of work, such as an HTTP request, database query or an internal computation. Spans typically include metadata like start time, duration, service name but they can be modified with additional attributes. They may also carry contextual information such as error indicators, HTTP status codes or custom attributes. This lets developers control what each span records. [10], [16]

All spans belonging to the same request are grouped under a trace. The relationships between spans are defined using parent-child references. For example, a span containing an HTTP request that was received by a frontend service might be the parent of another span that captures a database query made by that service. If the frontend calls another backend service, a new child span would be created from that request. [10], [19]

This structure creates a tree-like trace that shows the full path and timing of a request across the system, which helps developers understand the order and structure of a request. By analyzing the trace, it becomes easier to detect where latency accumulates, which services contribute to errors, or which dependencies slow down request handling. The trace-span model and its use for end-to-end request correlation were used in early tracing systems such as Dapper [10], [14], [20]

To avoid overloading the network or trace collectors, spans are usually batched before export. Rather than sending each span individually, instrumentation groups spans together and transmits them at regular intervals or once a buffer is full. This reduces overhead and helps ensure more efficient resource usage, especially in systems where there is high throughput. Batching improves performance but it also introduces a trade-off in observability latency. Some trace data may be delayed or even lost if the system fails before exporting the spans. [10], [16]

### 2.3.2 Context Propagation

The structure and terminology used in tracing systems tend to follow similar patterns. A trace is identified by a unique trace ID while each span has its own span ID along with a reference to its parent span ID. These identifiers let services link spans together even across different hosts and technologies. They can also link to corresponding log lines and metrics. In distributed environments like Kubernetes this matters since services are often short-lived and spread across multiple nodes or clusters. Trace context propagation across service boundaries is standardized through the W3C Trace Context specification [5], [10], [21]

Tracing tools also allow attaching custom data to spans. These could be business-specific tags or security-related attributes. This makes it easier to search for traces related to a particular user, endpoint, or anomaly. Additionally, spans can be sampled, meaning that only a portion of all traces is collected and analyzed, to reduce

overhead. Different sampling strategies have a direct impact on the visibility and reliability of tracing data. Sampling strategies will be discussed in more detail in section 2.4. Actual instrumentation will be covered in Chapter 4. [14], [16], [22]

## 2.4 Sampling Strategies

**Always-On Sampling** All traces are collected within the service. This provides full visibility of the service but can lead to high resource and network usage. It is generally only used in development or debugging environments since it will use more resources and it also produces redundant traces [14], [16].

**Always-Off Sampling** No traces are collected within the service. This can be used when no tracing data from a service is needed or used. [16]

**Probabilistic Sampling** A percentage of traces are collected at random, for example 10% of all traces. This helps reduce load while still providing insight into typical system behavior. However, rare or critical events may be missed unless specifically configured to be included. Best used when a service generates a huge amount of tracing data, for example high request HTTP services. [10], [16]

**Rate-Limiting Sampling** Limits the number of traces per time unit (e.g., 5 traces per second). This provides predictable resource usage regardless of traffic spikes but can result in sampling bias. [14], [16]

**Tail-Based Sampling** The sampling decision is made after a trace is collected, which allows smarter filtering (e.g., keeping only failed or slow requests). Tail-based sampling requires buffering entire traces before deciding what to keep and is often implemented by the collector rather than the client itself. [15], [22]

**Parent-Based Sampling** This strategy ensures that all spans within a distributed trace share the same sampling decision, meaning that if a parent span is sampled, its children are also sampled and vice versa. For root spans, another strategy, such as probabilistic or always-on, is applied. Parent-based sampling is useful for maintaining trace consistency across services, avoiding fragmented or incomplete traces, but it also means that individual traces and spans cannot be tweaked independently. [16], [21]

**Head-Based Sampling** The sampling decision is made at the very beginning of a request, before the trace has completed. Most tracing SDKs (including OpenTelemetry) use head-based sampling by default, as it minimizes overhead in client libraries and avoids buffering. However, it may miss valuable information since the decision is made without knowing whether the service runs will have errors or slowdowns. [10], [14], [23]

**Context-Aware Sampling** This strategy uses attributes of the request or runtime context to decide whether a trace should be collected. For example, sampling requests from specific endpoints, users, or error codes more aggressively than normal traffic. Context-aware sampling helps capture critical traces that may otherwise be dropped. It is often combined with probabilistic or tail-based strategies for finer control over resource use and observability. [22], [24]

### 2.4.1 Choosing a Strategy

Choosing the right sampling strategy depends on the system resource capacity, volume of the wanted data and observability needs within the system. For example, high-throughput systems might combine probabilistic sampling with tail-based rules to capture only performance outliers or errors, while in development always-on sampling can be used for full observability. [14], [22]

Sampling decisions can also be context-aware, such as retaining traces for a specific user, endpoint, or transaction type. While sampling reduces overhead, it introduces limitations in root-cause analysis and statistical accuracy. This trade-off must be managed carefully to avoid blind spots in system monitoring. Finding a balance between resource use and observability is critical. [14], [15]

Sampling has historically been used to control telemetry volume in production tracing systems and it directly shapes the cost and visibility trade-off of tracing [10]. More advanced approaches extend uniform random sampling by prioritizing unusual or diagnostically valuable traces, which supports strategies that increase sampling under abnormal conditions [22]. The OpenTelemetry specification defines the sampling interfaces used in this thesis and the placement of sampling decisions in the SDK [16].

## 2.5 Privacy and Security Considerations in Telemetry

Telemetry data can include sensitive information since logs, traces and metrics capture user inputs, internal system behavior and network interactions. This makes telemetry a potential target since it records what happened inside the system. It is similar to an audit log [25]. If this data is centralized, it should be considered who can access the data and whether they can tamper with it. [26]

### 2.5.1 Data Privacy

An important starting point is collecting only what is needed. Authentication endpoints are a common source of leaking sensitive data, since request bodies may contain tokens or credentials. Sensitive payloads such as passwords or personal tokens and identifiers should not appear in spans or logs. Filtering can be applied at instrumentation level or at the collector backend to force this. Most observation tools support regex-based redaction which makes it easier to apply consistent policies generally. [16], [27], [28]

Telemetry also carries risks related to how context travels across services. W3C `traceparent` headers propagate trace context across service boundaries and if these are forwarded to external or untrusted services, they can expose internal system topology. A single log line or metric may not identify anyone but combining traces, logs and metrics across services and systems can reveal patterns. This may be sufficient to profile a specific user or service. For this reason, both the scope of collected telemetry and its retention period should be considered. [21], [29], [30]

### 2.5.2 Security

On the backend side, observability infrastructure needs the same protections as other production services. The collector backend should require authentication and authorization. Telemetry should be encrypted in transit and at rest. mTLS between agents and collectors also protects against data injection into the tracing pipeline. For cloud-hosted deployments it is worth checking where collector backends store data by default since some services route to regions that may not comply with GDPR. Access to observability data should be logged since telemetry backends can be considered useful targets for anyone trying to map how a system works. [21], [25], [27], [31]–[33]

Sampling decisions also affect what ends up stored. Head-based sampling captures the full request before knowing its content which means sensitive data may be included in the payload. Tail-based and filtered sampling give more control since the decision can be based on what the trace actually contains. In systems with authentication flows it is often safer to exclude those spans from tracing entirely. [10], [22], [28]

These concerns are easy to miss when instrumentation is added gradually alongside development. In practice, data minimization and access controls can be considered high priority since they limit exposure even if other measures are not in place. Telemetry design is worth including in security reviews alongside other infrastructure components. [11], [25]

## 2.6 Related Work

To identify related work for this thesis, a targeted literature search was conducted using IEEE Xplore, ACM Digital Library, ScienceDirect/Elsevier and Springer-Link. Search terms included combinations of "OpenTelemetry", "distributed tracing", "sampling strategy", "microservice", "Kubernetes", "tracing overhead", "adaptive sampling", "telemetry privacy", "telemetry security" and "observability data governance". The search was narrowed to studies addressing tracing overhead, sampling behavior, Kubernetes-based microservices, adaptive and runtime sampling and telemetry data security. The purpose was to identify literature relevant to the thesis experiments and results.

The performance overhead of distributed tracing instrumentation in microservice systems has attracted research interest, particularly as OpenTelemetry has become a widely adopted standard for vendor-neutral telemetry. Related work covers empirical overhead measurements, sampling strategy design and evaluation, observability frameworks and the privacy implications of telemetry.

### Empirical Overhead Measurements

Several studies have measured the resource overhead introduced by OpenTelemetry instrumentation in Kubernetes-based environments. Nõu et al. [34] measured performance overhead of distributed tracing using both OpenTelemetry and Elastic APM across microservice and serverless deployments on Kubernetes. The study compares the effect of different sampling configurations on CPU utilization and request latency. Their results show that overhead varies with sampling rate and workload type. Higher sampling rates correlate with increased resource consumption.

Karkan [35] examined OpenTelemetry sampling methods in a cloud infrastructure and reported CPU overhead of 71.33% under always-on sampling relative to a no-tracing baseline. Sandberg [36] evaluated the impact of OpenTelemetry on CPU

and latency in a microservice architecture and reported CPU overhead reaching 42% at maximum sampling rates. Both of these studies confirm that sampling rate is the primary driver of tracing overhead and that reducing the rate brings resource use close to the no-tracing baseline. These study results provide reference points for the measurements presented in this thesis.

Based on the searches, no study was found that simultaneously compares head-based, parent-based and a custom adaptive sampling strategy in a single Go microservice deployed on Kubernetes using OpenTelemetry.

### Sampling Strategy Design and Evaluation

Beyond head-based and probabilistic sampling, research has explored more adaptive approaches. Huang et al. [37] proposed Trastrainer, an adaptive sampling system that adjusts sampling rates based on system runtime state. The system dynamically raises or lowers sampling depending on observed anomalies and resource conditions. The aim is to maintain trace visibility while limiting overhead. This approach is related to the adaptive sampler implemented in this thesis.

Chen et al. [38] address the practical limitations of head-based sampling by proposing TraceMesh, a scalable streaming sampler for distributed traces. TraceMesh uses locality-sensitivity hashing to cluster traces and dynamically adjusts sampling decisions to avoid over-capturing common recurring patterns. This demonstrates the trade-off between trace completeness and storage costs in high-throughput environments.

### Observability Frameworks and Instrumentation Overhead

OpenTelemetry has displaced earlier framework solutions such as Jaeger and Zipkin by providing a vendor-neutral, CNCF-standardized instrumentation layer that decouples application code from the backend that is used to store and visualize

traces [39]. Comparable instrumentation overheads have been reported, suggesting that the overhead findings extend to distributed tracing in general rather than specific to OpenTelemetry [39].

Usman et al. [40] conducted a survey of observability approaches in distributed edge and container-based microservices. This survey covers tracing, metrics and logging in Kubernetes environments. The survey establishes OpenTelemetry as the dominant instrumentation standard and identifies overhead management and sampling strategy selection as key open problems in the field.

Reichelt et al. [39] benchmarked the instrumentation overhead of multiple frameworks using the MooBench benchmark. This shows that the framework choice measurably affects CPU and latency regardless of sampling configuration. This provides broader context for interpreting the findings of this thesis: overhead comes from both the instrumentation layer itself and the sampling behavior on top of it.

### Privacy and Security of Telemetry Data

Privacy implications of observability data have received less attention than performance concerns in the literature. Usman et al. [40] note that data governance (what is collected, how long it is retained, and who can access it) remains an open problem in observability deployments. OpenTelemetry span attributes routinely include HTTP routes, service names, operation names and user-agent strings, meaning that trace data can carry sensitive information about user interactions even when that is not the intent. This is the context for the privacy and security discussion in RQ4.

## 2.7 Closing summary

Chapter 2 explains how logs, metrics and traces give visibility in modern microservice systems. Span structure and context propagation show how requests move across services. The chapter covers sampling strategies and where in the pipeline they can be applied and the trade-off between resource use and observability. It also lists privacy and security practices for telemetry, such as collecting only what is needed, filtering out sensitive fields, controlling and auditing access and setting retention for stored data. Related work gives context to the thesis through existing empirical overhead measurements, sampling strategy research and observability surveys.

Observability is a set of design choices such as what to instrument, how to structure spans, how to sample and how to protect telemetry through its lifecycle. Observability should be measured, developed and revisited as systems and data regulations evolve. These concepts are explored more in later chapters.

Chapter 3 defines the research design used to measure tracing overhead and evaluate sampling strategies in Kubernetes-hosted Go services. It specifies the research questions and the methods used to design the experiments. Chapter 4 then describes the experimental setup in detail, including the system under test, infrastructure environment, workload generation and sampling strategy implementations.

# 3 Research Design

## 3.1 Research Questions

This chapter describes the research approach used to measure the performance overhead of distributed tracing and to compare different sampling strategies in Go-based microservices running on Kubernetes. The goal is to provide a clear and repeatable process that builds on the concepts from Chapter 2 and produces interpretable results. The experimental design is based on established guidelines for conducting and reporting controlled software engineering experiments [41], [42].

**RQ1.** What overhead does distributed tracing introduce compared to a no-tracing baseline for a Go microservice on Kubernetes?

**RQ2.** How do sampling strategies change overhead and diagnostic value under comparable load?

**RQ3.** Is it feasible to have adaptable sampling strategies?

**RQ4.** What privacy and security trade-offs follow from these configurations, based on the experimental results and existing literature?

**Operational notes.** Overhead refers to the increase in response time, reduction in throughput and higher CPU and memory usage compared to running the same services without tracing enabled. Because diagnostic usefulness is difficult to measure

directly, practical checks are used instead: whether traces are complete, whether errors have traces attached and whether key attributes are present. Section 3.3 lists the specific metrics used to measure both overhead and diagnostic value.

Measuring a system can affect its behavior, so the influence of instrumentation is accounted for when interpreting the results [43]. Chapter 4 describes the system under test, the test environment, the workloads used, and how each sampling strategy was implemented.

## 3.2 Experimental Factors and Variables

This section lists what is tested during the research, what tools are used and what measurements are expected. The aim is to keep the factors small and clear so results are easy to compare with each other.

### 3.2.1 Factors in the experiment

**Sampling strategy** None (baseline measurement), head sampling, parent-based sampling and adaptive sampling. The baseline shows the natural performance of the service without tracing. Head sampling keeps a fixed percentage of requests based on trace ID. Parent-based sampling follows parent span decisions when available. Adaptive sampling adjusts the sampling rate based on events in the service (e.g., increased rate in sampling when errors occur). The adaptive logic was developed as part of this thesis.

**Sampling rate** The sampling rate is set to one of six fixed values: 1%, 5%, 10%, 30%, 50% and 100%. These are configured using the OpenTelemetry Go SDK's built-in samplers.

**Load level** Low, medium and high request rates with fixed patterns per level.

**Payload size** Small vs. large to test how message size interacts with tracing overhead.

Each run changes one of the above where possible. When two factors change together (e.g., strategy and rate), the combination is treated as one configuration.

### 3.2.2 Measured outcomes

**Service performance** Latency (p50, p95, p99), throughput, HTTP error rate, CPU and memory.

**Observability pipeline** Collector resource usage available from the observability stack.

**Tracing backend** Ingest rate, storage writes and dropped span counts.

**Network** Egress bytes and overall network usage

### 3.2.3 Conditions

- Fixed Kubernetes version, node type, kernel and container images.
- Resource requests/limits set and unchanged and autoscaling disabled.
- Time sync enabled on all nodes, meaning identical deployment workflow for all runs.
- Load profiles taken from the middle of a test run.
- One code branch and image tag for the service, meaning configuration via values files only.

### 3.2.4 Experiment Matrix and Run Order

The experiment matrix is built from combinations of sampling strategy × load level × workload endpoint.

**Baseline** 1 sampling strategy × 3 load levels × 5 endpoints = 15 configurations.

**Head sampling** 6 rates (1%, 5%, 10%, 30%, 50%, 100%) × 3 load levels × 5 endpoints = 90 configurations.

**Parent-based sampling** 5 rates (1%, 5%, 10%, 50%, 100%) × 3 load levels × 5 endpoints = 75 configurations.

**Adaptive sampling** 4 base rates (1%, 10%, 30%, 50%) × 3 load levels × 5 endpoints = 60 configurations.

Total: 240 unique configurations.

The adaptive configurations use different base rates but the rate can shift up during a run when errors or latency thresholds are hit. Each configuration was run twice to reduce noise in recorded values. Adaptive configurations were run three times since they showed more variation between runs than the static strategies.

## 3.3 Metrics and Data Collection

This section describes the metrics collected during the experiments and how they were processed for analysis. Measurements covered the service under test, the Kubernetes environment, the load generator and the trace backend. To ensure consistency, only data from the 10-minute measurement window was included in the analysis. Prometheus scraped metrics every 15 seconds and all raw data was saved for analysis. [13]

### 3.3.1 Service-Level Metrics

Service-level metrics capture the direct impact of tracing on request handling and instrumentation:

**Latency and throughput** Request duration (p50, p95, p99) and response rates, measured via server-side Prometheus histogram metrics and cross-checked against load generator throughput counters.

**Error rate** Proportion of HTTP 5xx and timeout responses, also used to confirm adaptive sampling triggers.

**Trace volume** Number of spans ingested by the trace backend, verifying that the intended sampling rates were applied.

### 3.3.2 System-Level Metrics

System metrics were collected from Kubernetes to quantify resource overhead:

**CPU utilization** Per pod and per node, with focus on the instrumented service pods.

**Memory usage** Container working set memory of service pods, as reported by cAdvisor via the Kubernetes metrics API, reflecting the additional memory footprint of instrumentation and span buffering.

**Network usage** Outbound traffic from service pods to the collector, measuring export overhead.

### 3.3.3 Load Generator Metrics

The load generator (`hey`) was used to send requests to the service endpoints. Achieved throughput was confirmed through the `rps_total` and `rps_2xx` Prometheus metrics to verify that the generator sustained the configured RPS levels. Request latency was measured server-side via Prometheus histogram percentiles (p50, p95, p99) rather than from client-side generator summaries [43].

### 3.3.4 Trace Backend Validation

The trace backend (Tempo) was monitored to validate end-to-end correctness:

**Sampling confirmation** Comparison of expected vs. observed sampling rates via ingested span counts (`tempo_ingest_spans`, `tempo_traces_created`).

**Export reliability** Detection of dropped spans at the backend (`tempo_spans_dropped`), ensuring overhead results reflect actual tracing activity. Collector-internal metrics such as queue depth and exporter error counters were not available since the collector was not configured to expose them to Prometheus, and are not included in this analysis.

### 3.3.5 Data Processing and Quality Control

All metrics were aggregated over the 10-minute measurement window, using medians and 95th percentiles for latency and throughput and mean values for CPU, memory and network utilization [44].

After collecting the data, runs were checked for large variances between runs of the same configuration. Runs with incomplete data (e.g., missed scrapes) or unusually high variance were excluded. Only data from the middle of the runtime was used to cut out any startup or shutdown period data that would not represent normal workload. The data points from all runs of the same configuration were then

averaged into a single mean value per metric, which was used as the comparison value for that configuration. These averages were then compared against each other to evaluate differences in overhead for each sampling strategy.

### 3.3.6 Summary of Metrics

Table 3.1: Summary of metrics collected during experiments.

| **Metric** | **Source** | **Purpose** |
|---|---|---|
| Latency (p50, p95, p99) | Service (Prometheus histograms) | Measure request latency and baseline overhead |
| Throughput | Service (Prometheus), load generator | Measure sustained request rate under load |
| Error rate | Service logs, load generator | Validate adaptive sampling triggers and stability |
| Trace volume | Trace backend (Tempo) | Confirm sampling configuration and volume reduction |
| CPU utilization | Kubernetes (per pod/node) | Quantify processing overhead from instrumentation |
| Network usage | Kubernetes (service pod) | Measure export overhead to collector |
| Achieved request rate | Load generator, Prometheus | Verify load generator maintained target RPS |
| Dropped spans | Trace backend (Tempo) | Detect span loss and validate pipeline reliability |

# 4 Experimental Setup

This chapter describes the technical components and configurations used to execute the experiments defined in Chapter 3. It covers the test service implementation, infrastructure environment, workload generation procedures and the specific sampling strategy implementations. Together, these elements form the experimental apparatus that enables controlled measurement of tracing overhead.

## 4.1 System Under Test

The test service is a Go HTTP application with OpenTelemetry instrumentation, designed to represent typical microservice workloads. The service provides multiple endpoints that simulate different workload types and resource usage patterns. It also includes custom adaptive sampling logic to implement dynamic sampling ratios based on different statuses of the query endpoints.

### 4.1.1 Service Endpoints

The service exposes five primary endpoints for testing load:

**/api/basic** Lightweight computation representing standard business logic. It counts the sum of squares from 0 to 999 to add calculation processes.

**/api/large-payload** Configurable payload generation to test memory and network impact. The default is 100 KB.

**/api/long-running** Simulated slow operations for timeout and span duration testing. Applied 1 to 5 second timeout.

**/api/deep-spans** Nested span creation to evaluate tracing overhead with call tree. Set span level 5 in the service.

**/api/cpu-intensive** CPU-intensive workload that uses prime calculation to test resource utilization.

The service also comes with additional endpoints that provide service status and monitoring:

**/health** Service health status and tracing configuration.

**/config** Current configuration in JSON format.

**/adaptive-metrics** Real-time adaptive sampling metrics (only when using the adaptive strategy).

### 4.1.2 Tracing Configuration

The service supports runtime configuration via environment variables that map to the experimental factors. These are set before each test run:

**TRACING_ENABLED** Boolean to enable/disable all tracing from the service. Baseline resource usage is tested with tracing disabled.

**SAMPLING_STRATEGY** Options: `always_on`, `always_off`, `traceidratiobased`, `parentbased_traceidratiobased`, `adaptive`.

**SAMPLING_RATE** Float64 value (0.0–1.0) for probabilistic sampling rates.

**COLLECTOR_ENDPOINT** OTLP HTTP endpoint for trace export.

The service uses OpenTelemetry Go SDK v1.19.0 with HTTP instrumentation via `otelhttp`. Spans include standard semantic conventions and custom attributes to classify each operation.

The adaptive sampler was built to test whether runtime logic for adjusting the sampling rate affects overhead in Go services. For this, additional configuration parameters control the adaptation behavior:

**ADAPTIVE_BASE_RATE** Starting sampling rate (default: 0.05).

**ADAPTIVE_ERROR_BOOST** Multiplier for error conditions (default: 3.0).

**ADAPTIVE_LATENCY_THRESHOLD** Latency threshold in milliseconds (default: 1000).

**ADAPTIVE_LATENCY_BOOST** Multiplier for high latency (default: 2.0).

**ADAPTIVE_UPDATE_INTERVAL** Rate recalculation interval in seconds (default: 30).

**ADAPTIVE_MAX_RATE** Maximum allowed sampling rate (default: 1.0).

Note: adaptive sampling does not allow the sampling rate to drop below 0.01, ensuring that some traces are always collected.

### 4.1.3 Deployment Configuration

Container images are built from a fixed Dockerfile and deployed to a Kubernetes cluster using manifests. The Dockerfile and deployment manifests are provided in the appendix of this research. The service runs with fixed resource requests and limits to ensure consistent baseline performance. These limits are set to avoid "out of memory" errors and prevent worker nodes from evicting services. No horizontal or vertical autoscaling is enabled during experiments. The worker nodes host other services in parallel with this test service.

## 4.2 Environment and Infrastructure

The test environment uses a managed Kubernetes cluster to provide realistic deployment conditions while maintaining experimental control. This cluster hosts other services outside of this research, but the experiments have dedicated compute resources inside the cluster. Supporting services such as the observability stack will not be changed during the experiments to keep conditions consistent.

### 4.2.1 Kubernetes Cluster

The cluster uses a Rancher-managed Kubernetes setup running `v1.33.3+rke2r1` with one worker and one master. The cluster infrastructure consists of virtual machines hosted in VMware vSphere. Cluster scaling is managed through Rancher which is a centralized Kubernetes management platform [45].

The nodes are uniform: 4 vCPU, 8 GB RAM, Ubuntu 24.04 LTS. The kernel version is `6.8.0-35-generic` and `containerd v2.0.5-k3s2` is used for container runtime [46].

The cluster uses Cilium as the Container Network Interface (CNI). Cilium provides eBPF-based networking and also provides good observation tools and security policies [47]. The cluster also uses Traefik v2.2 as the ingress controller and load balancer [48].

This infrastructure was set up to mimic a production-like environment with realistic networking and load balancing while maintaining experimental control through virtualization (vSphere) and centralized management (Rancher).

### 4.2.2 Observability Stack

Grafana Alloy `v1.10.1` [49] is deployed as a DaemonSet for service log collection from worker node. Alloy is a telemetry collection agent that aggregates container logs from all pods on the worker and ships them to Loki `v3.5.5` for log storage [50]. Loki is a log aggregation system that stores log data in object storage while only indexing metadata. This reduces storage costs.

Application traces are sent from the Go service to Grafana Tempo [17] via OTLP HTTP endpoints. Grafana Tempo `v2.8.2` works as the tracing receiver. The service exports traces directly to Tempo without intermediate processing, keeping the pipeline simple. Tempo stores the traces in a dedicated volume.

This architecture separates infrastructure monitoring from application tracing, so that tracing overhead can be measured without interference from metric collection and processing. This separation is also used in modern observability stacks where logs, metrics and traces have their own pipelines.

Metrics are collected from the Go service with the use of Prometheus [13] `v2.45.3`. Prometheus is a time-series monitoring system that uses a pull-based model to collect metrics from services and infrastructure components. It stores metrics in a local time-series database and provides PromQL for querying and alerting based on collected metrics. Retention is set to 7 days during experiments, with automated cleanup afterward to free up space while keeping enough data for analysis. Prometheus is configured to scrape metrics from all components every 15 seconds.

Grafana `v10.2.1` serves as the visualization and dashboard platform [51]. Grafana provides visualizations and data source integrations for metrics, logs and traces. It connects to Prometheus for metrics visualization, Loki for log exploration and Tempo for trace analysis. Grafana can also be used for alerting but that is not relevant for the experiments. Custom dashboards are configured to monitor experimental metrics, service performance and infrastructure health during test execution. In this thesis, Grafana is used to verify that the experiment's core services are stable.

### 4.2.3 Version Control and Deployment

All infrastructure services are version-controlled in GitHub repositories with GitOps deployment handled through pipelines. GitHub serves as the centralized version control system where infrastructure configurations are maintained and tracked. Any changes to infrastructure components are handled through GitHub.

The Go service used in this thesis is maintained in a separate repository with independent versioning. Service code changes are pushed to GitHub, after which a Docker image is built and tagged based on the service iteration. Other versioning schemes are not needed.

This separation allows independent versioning on the test service while maintaining stable infrastructure versions throughout the experiment. All configuration changes and experiment parameters are tracked through Git commits. This ensures reproducibility of the experiments and keeps track of any changes made.

## 4.3 Workload Generation and Test Execution

Load generation with Hey simulates realistic HTTP traffic while keeping runs repeatable. The goal is to apply controlled load to the service endpoints to see how different request volumes and patterns affect tracing overhead.

The following subsections describe the load generator setup, execution timeline, request patterns and deployment considerations in more detail.

### 4.3.1 Load Generator Configuration

The experiment uses `hey v0.1.4` as the HTTP load generator [52]. `Hey` is a lightweight Go-based tool capable of sustaining fixed request rates with minimal overhead on the client side. This makes it ideal for this experiment as it generates stable request patterns without significant variance from the client side.

Three load levels used in the experiment:

**Low** 50 requests per second (RPS) in total.

**Medium** 500 RPS in total.

**High** 1000 RPS in total.

These levels target different operational conditions: low load establishes baseline overhead, medium load reflects typical service behavior and high load pushes the system toward saturation, where the impact of tracing overhead is most pronounced.

### 4.3.2 Test Execution Timeline

Each run follows a timeline to ensure consistency across configurations:

- 0–2 minutes: service deployment, runtime start and health checks for the service.
- 2–4 minutes: warm-up period to allow the service to stabilize.
- 4–14 minutes: measurement window of 10 minutes.
- 14–16 minutes: metrics export.

The total duration of a single run is 16 minutes. Focusing on the 10-minute window after warm-up avoids startup effects and produces data that is reliable for analysis.

### 4.3.3 Request Patterns and Endpoint Isolation

Traffic is directed to one endpoint at a time so the tracing overhead of each operation can be measured in isolation and without interference. This avoids overlap between workloads and makes it possible to directly link performance differences to endpoint behavior.

Endpoints used in testing:

**/api/basic** Lightweight computation with very small JSON payloads (<1 KB).

**/api/large-payload** Configurable payload generation, defaulting to 100 KB responses.

**/api/long-running** Simulated slow operations with fixed delays (1–5 seconds).

**/api/deep-spans** Requests generating nested spans (five levels deep).

**/api/cpu-intensive** CPU-bound operations such as prime calculation.

Request design characteristics:

- Deterministic payloads: fixed JSON structures remove variance from serialization.
- Consistent timing: open-loop generation prevents latency feedback.
- Reproducible content: identical request bodies across test iterations.
- Minimal client overhead: simple HTTP requests without authentication or session state.

This design makes it possible to observe how different workload types interact with tracing instrumentation and to identify which patterns are most sensitive to sampling strategies and overhead.

### 4.3.4 Load Generator Deployment

To run `hey` next to the Go service, a test pod is deployed in the same namespace and worker node. This pod image is a minimal Alpine Linux environment with `hey` installed in it. All requests to the experimental service are run through this test pod over Cilium CNI network.

The location of the test pod removes any variables from external networking and ensures that latency between the client and service remains consistent across experiment runs. To prevent overuse of resources on the worker, the load generator pod is given fixed resource allocations.

## 4.4 Sampling Strategy Implementation

This section defines the specific sampling configurations tested in the experiment. The theoretical sampling approaches that were introduced in Chapter 2 are implemented in OpenTelemetry so that their overhead can be measured and compared.

Each strategy has a trade-off between observability and system resource usage. This makes it possible to compare their effects side by side.

### 4.4.1 Baseline Configuration (No Tracing)

The baseline configuration disables all tracing instrumentation to give a benchmark on the service performance. No spans are created and telemetry is not exported to Tempo. This captures the natural performance characteristics of the service without interference from instrumentation.

This configuration is applied by disabling tracing in the application configuration, relying on no-operation (noop) implementations that OpenTelemetry provides. The noop tracer creates placeholder objects that perform no actual work, ensuring that instrumentation code paths remain unchanged while eliminating tracing overhead.

### 4.4.2 Head Sampling Configurations

Head sampling decides whether to record a trace when a request first enters the system. This is done using OpenTelemetry's `TraceIDRatioBased` sampler, which makes a decision based on the trace ID. The sampler takes the 64-bit trace ID and applies a hash to it. It then compares the trace ID against the configured sampling strategy. This ensures that the same trace ID always produces the same decision.

Head sampling works like a reproducible "coin flip" controlled by the trace ID. The configured sampling rate sets how often that "flip" comes up "trace this." For example, at a 1% rate, roughly one out of every hundred requests will be traced. Because the decision is made only once at the start of the trace, every span belonging to that trace either gets recorded or not. This makes overhead predictable. Instrumentation costs remain constant since spans are always created, while export and storage costs scale with the number of traces sampled and transmitted.

The configurations used in experiments are:

**1%** `SAMPLING_STRATEGY=traceidratiobased`, `SAMPLING_RATE=0.01`

**5%** `SAMPLING_STRATEGY=traceidratiobased`, `SAMPLING_RATE=0.05`

**10%** `SAMPLING_STRATEGY=traceidratiobased`, `SAMPLING_RATE=0.10`

**30%** `SAMPLING_STRATEGY=traceidratiobased`, `SAMPLING_RATE=0.30`

**50%** `SAMPLING_STRATEGY=traceidratiobased`, `SAMPLING_RATE=0.50`

**100%** `SAMPLING_STRATEGY=always_on`, `SAMPLING_RATE=1.0`

### 4.4.3 Parent-Based Sampling Configuration

Parent-based sampling makes its decision by looking at the parent span if one is present. When a request enters the service with an existing trace context, the sampler inherits the parent's decision. If the parent was sampled, all child spans are sampled. And if the parent was dropped, the children are also dropped. This ensures that the trace remains consistent through services.

For requests that start a new trace (root spans), there is no parent decision to follow. In this case, OpenTelemetry's `ParentBased` sampler falls back to a `TraceIDRatioBased` policy, using the same hash-based comparison described in Section 4.4.2.

Tested rates for parent-based: 1%, 5%, 10%, 50% and 100%. The 100% configuration was included to compare parent-based behavior against always-on head sampling, even though for single-service entry points the two strategies produce the same sampling decisions.

This strategy captures how parent–child relationships affect the sampling outcomes. Compared to pure head sampling, it avoids inconsistent recording of spans across services. The experiments measure whether this has any additional overhead compared to independent head-based decisions.

### 4.4.4 Adaptive Sampling Configuration

Adaptive sampling uses a custom sampler that dynamically adjusts rates based on real-time service conditions. The sampler recalculates the sampling value every 30 seconds and applies multipliers when error rates or latencies exceed values set in the configuration. The maximum sampling rate is 100%. This allows the system to capture more traces during conditions that are not normal to the service while keeping overhead low on normal runtime.

Key configuration parameters:

**Base rates** 1%, 10%, 30% and 50%.

**Update interval** 30 seconds.

**Error multiplier** $\times 3$ when HTTP errors are detected in the service.

**Latency threshold** 1000 ms, with a $\times 2$ multiplier applied when exceeded.

Adaptive sampling changes behavior during the service runtime. It adds some constant cost for monitoring conditions and overhead can increase temporarily when errors or slow requests trigger higher sampling rates.

### 4.4.5 Configuration Matrix and Test Combinations

The experiment design combines sampling strategies with workload types and load levels to give ample data on overhead comparison.

`Workload endpoints in the experiment`:

- `/api/basic` (lightweight requests)
- `/api/large-payload` (network and memory heavy)
- `/api/cpu-intensive` (compute heavy)
- `/api/long-running` (latency sensitive)
- `/api/deep-spans` (nested span hierarchy)

`Load levels`:

**Low** 50 requests per second (RPS)

**Medium** 500 RPS

**High** 1000 RPS

`Sampling strategies`:

**Baseline** No tracing (1 configuration).

**Head sampling** 1%, 5%, 10%, 30%, 50% and 100% rates (6 configurations).

**Parent-based sampling** 1%, 5%, 10%, 50% and 100% rates (5 configurations).

**Adaptive sampling** Four base rates (1%, 10%, 30%, 50%) with dynamic boosting (4 configurations).

Below is the configuration matrix:

Table 4.1: Experiment configuration matrix.

| **Strategy** | **Rates** | **Loads** | **Endpoints** | **Total** |
|---|---|---|---|---|
| Baseline | 1 | 3 | 5 | 15 |
| Head sampling | 6 | 3 | 5 | 90 |
| Parent-based | 5 | 3 | 5 | 75 |
| Adaptive | 4 | 3 | 5 | 60 |
| **Total** | | | | **240** |

Each configuration was run twice to capture any variation between experiment runs. Adaptive configurations were run three times since the dynamic capturing & value changes can have more variation between runs than the static strategies. This gives approximately 240 experiment runs in total. After the data has been captured, any large variances between runs were identified, and only data from the middle of the runtime was used to cut any warm-up or shutdown period data that did not represent the experiment.

# 5 Results and Analysis

This chapter goes through the results from the experiments described in Chapter 4. The analysis follows the research questions from Chapter 3. For RQ1 to RQ3 it means analysis on measured data. RQ4 on the other hand is structured a bit differently since the experiment results are used to analyze the privacy and security aspects of different tracing configurations.

Load was generated with fixed configurations using `hey`. The actual numbers reflect what the service could sustain during runtime rather than the configured target request rate. All overhead percentages are calculated relative to the baseline achieved requests per second (RPS). This means that if the service could not keep up with the target RPS, that shows up as a capacity limit on the service side and not as a data quality issue.

## 5.1 Research Question 1

### 5.1.1 Endpoint Selection

Two endpoints were selected for this research question: `api-basic` and `api-cpu-intensive`. These endpoints can be considered as the lightest and heaviest workloads in the test suite in terms of resource usage. The resource usage of the other endpoints (large-payload, deep-spans, long-running) fell between `api-basic` and `api-cpu-intensive`. This is a simplification but it keeps the results subsection more manageable and does not affect the analysis of the gathered data.

### 5.1.2 CPU Overhead

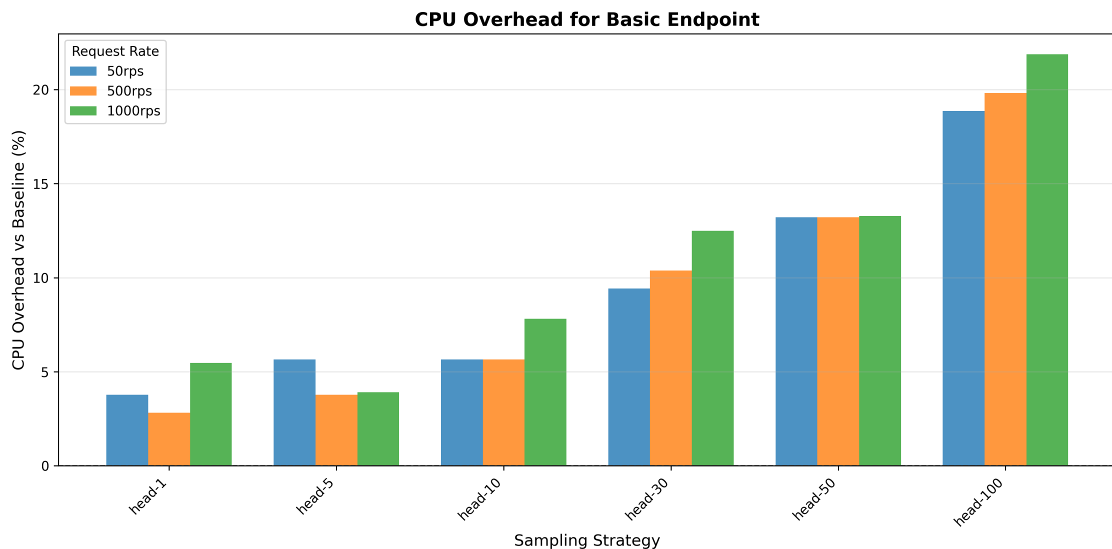


Figure 5.1: CPU overhead for api-basic endpoint with configured sampling rates. Figure shows percentage increase in CPU usage relative to baseline (0% sampling) at different load levels (50, 500, 1000 RPS).

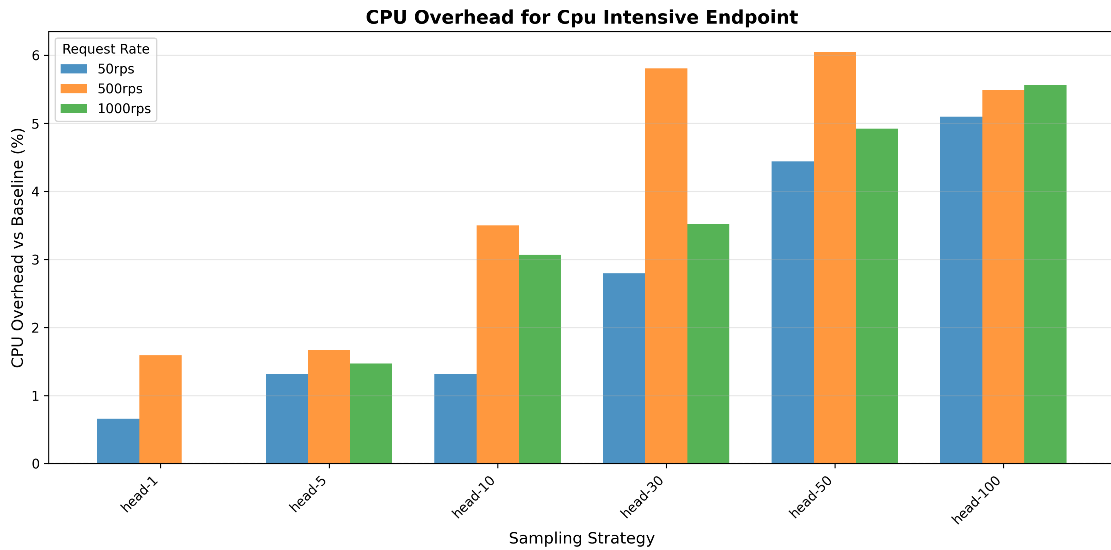


Figure 5.2: CPU overhead for api-cpu-intensive endpoint with configured sampling rates. Despite higher CPU usage, overhead percentage follows the same pattern as api-basic endpoint.

Figures 1 and 2 show how CPU overhead from tracing scales with the sampling rate for both lightweight and CPU-heavy workloads. CPU usage in terms of total usage is very different between the two endpoints but the relative overhead that tracing adds follow a pattern in both cases.

For api-basic (Figure 1), CPU overhead goes up roughly linearly with the sampling rate. At low rates (1–10%), the overhead stays below 10% across all load levels. At 100% sampling it climbs to around 18–22% depending on the request rate. This makes sense because the endpoint itself does little computational work so the resource cost of creating and exporting spans becomes more visible relative to the baseline.

The api-cpu-intensive endpoint (Figure 2) shows a different picture. The relative overhead stays below about 6% even at full sampling. The reason is that the endpoint already uses a lot of CPU for its computation, so the tracing cost gets

mitigated by the larger baseline CPU usage. This indicates that tracing overhead becomes less significant as the workload itself gets heavier.

Across both endpoints and all load levels, CPU overhead scales smoothly with the sampling rate. There are no sudden spikes. This shows that tracing does add resource cost but it stays predictable. Lightweight endpoints feel it more at high sampling rates while compute-heavy workloads mitigate the resource usage.

No configuration caused the service to become unstable or run out of resources. Memory usage stayed flat across all sampling rates. It shows that in Go services, tracing overhead can be controlled with sampling without affecting core performance.

### 5.1.3 Request Latency

Request latency (p95 and p99, the 95th and 99th response time percentiles) was also measured for all configurations. Figures 5.3 and 5.4 show the results for the two representative endpoints.

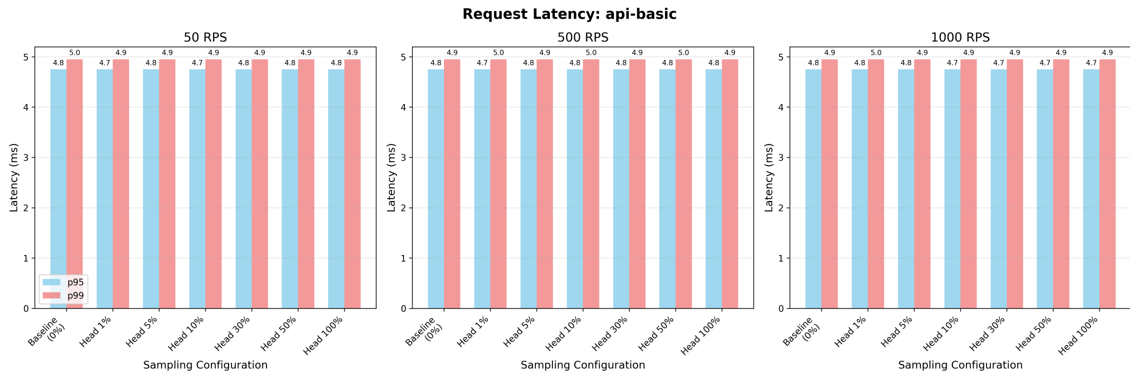


Figure 5.3: p95 and p99 request latency for *api-basic* from all configured sampling rates and load levels. Both percentiles stay close to the no-tracing baseline regardless of configurations.

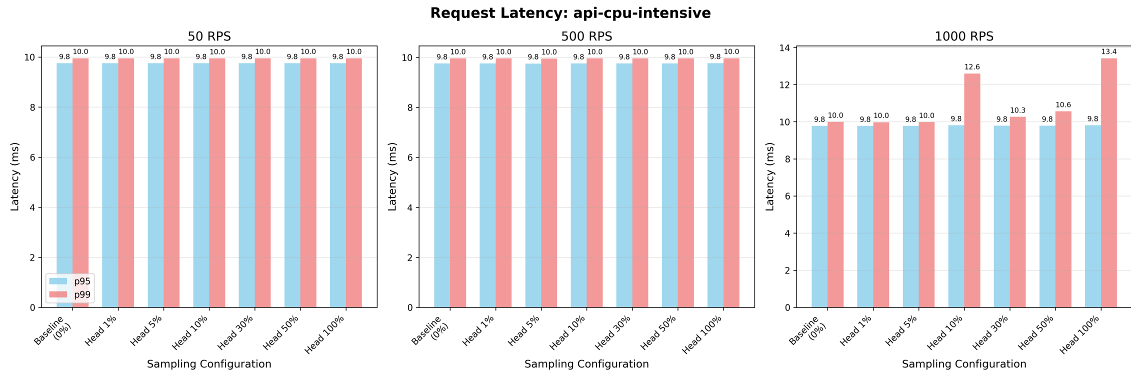


Figure 5.4: p95 and p99 request latency for *api-cpu-intensive* all configured sampling rates and load levels. At the highest load and sampling rate combinations, p99 rises slightly as the service approaches CPU-usage limitation but the increase stays consistent between test runs.

For api-basic, p95 and p99 stay within a small margin across all configured sampling rates and load levels. Tracing does not seem to produce any measurable latency increase for this lightweight workload. For api-cpu-intensive, p99 rises slightly at the highest load and sampling configuration combinations as the service approaches high CPU-usage but there is no evidence of anything being unstable. None of the configurations produced anomalous latency, so the analysis remains valid.

## 5.2 Research Question 2

### 5.2.1 Scope and Context

This subsection builds on the baseline overhead analysis from RQ1. RQ2 is based on the question of whether the sampling strategy used in tracing affects resource usage overhead. The comparison used in this subsection is between head-based and parent-based sampling.

The similar results between the two strategies are partly explained by the single-

service setup. As discussed in chapter 2, parent-based sampling works by inheriting the sampling decision from an incoming trace header. In a multi-service system this is what keeps a trace consistent through its lifecycle, but in this experiment there are no incoming parent spans. This means that every request starts a new root span. When that happens, parent-based sampling falls back to the same TraceIDRatioBased decision as head-based sampling. The architectural difference only shows up when sampling decisions need to carry over from one service to another.

The following subsections compare the resource overhead of head-based and parent-based sampling across CPU usage, memory consumption, request latency and network egress. The goal is to determine whether the choice of sampling strategy produces any measurable difference in overhead, independently of the sampling rate applied.

### 5.2.2 Experiment Overview

The same five workload endpoints as in RQ1 were tested:

- *api-basic*: lightweight request processing
- *api-cpu-intensive*: CPU-heavy workload
- *api-deep-spans*: deep span hierarchy
- *api-large-payload*: increased network usage
- *api-long-running*: requests with longer processing time

Each workload was tested at three load levels (50, 500 and 1000 requests per second) and six sampling rates (1%, 5%, 10%, 30%, 50% and 100%).

### 5.2.3 CPU Usage

From all of the configured workloads and load levels, CPU usage does not show much difference between head-based and parent-based sampling. Both strategies have a similar usage pattern. CPU goes up with the request rate and sampling rate as expected from the RQ1 findings, but the strategy itself does not seem to matter.

There are some small differences in the high-load and high-sampling runs, but the differences flip between strategies and do not grow consistently with load or sampling rate. This looks like normal variance between runs.

For CPU-heavy workloads the workload itself uses most of the CPU, and for slow workloads like *api-long-running* the CPU usage is low by design. In both cases, changing the sampling strategy has little noticeable effect on CPU usage.

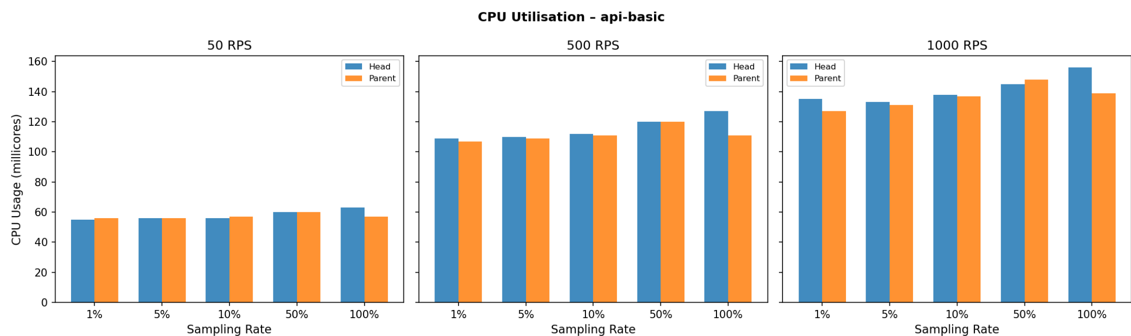


Figure 5.5: CPU usage as a function of sampling rate for the *api-basic* workload at 50, 500 and 1000 requests per second, comparing head-based and parent-based sampling.

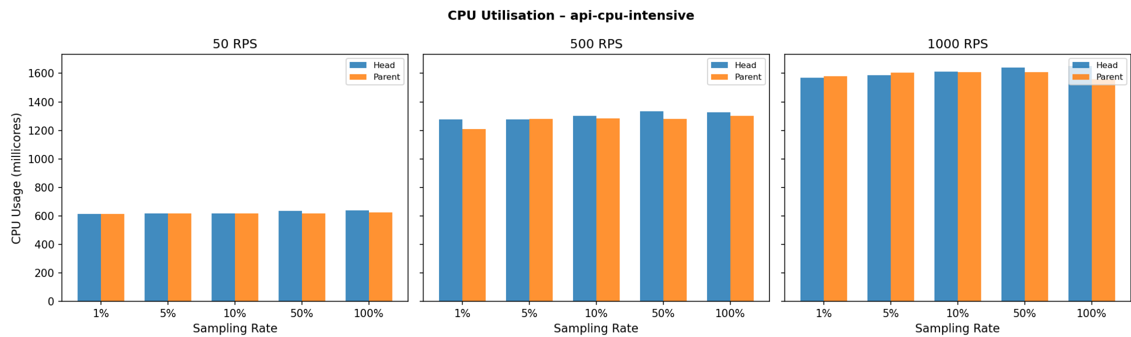


Figure 5.6: CPU usage as a function of sampling rate for the *api-cpu-intensive* workload at 50, 500 and 1000 requests per second. Application CPU dominates over any sampling strategy effects.

### 5.2.4 Request Latency

Request latency stays the same between the two strategies. For fast endpoints, p95 and p99 are stable across all sampling rates. The sampling decision does not affect the request path in any measurable way.

At high request rates on the CPU-intensive workload, there is some minor p99 variability as the service gets close to saturation. But this hits both strategies equally and without a consistent direction. For delay-dominated workloads like *api-deep-spans* and *api-long-running*, the injected delays dominate the latency numbers completely, so any sampling-related effect is invisible.

In short, parent-based context propagation does not add any latency compared to independent head-based decisions.

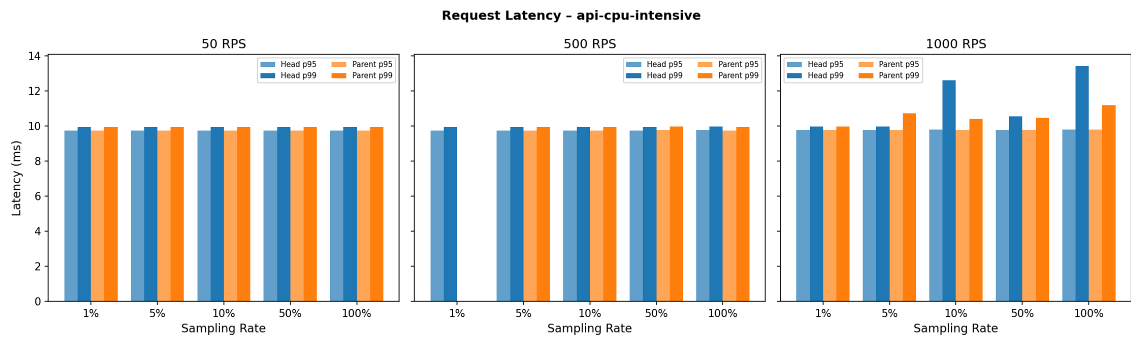


Figure 5.7: p95 and p99 request latency for the *api-cpu-intensive* workload across sampling rates and load levels. Both strategies show the same tail-latency behavior.

### 5.2.5 Memory Usage

Memory usage stays flat across sampling rates and load levels. Memory use stays within roughly ±0.5 MiB across all experiment configurations.

### 5.2.6 Network Egress

Network egress scales with request rate and sampling rate, which makes sense since more sampled spans means more data exported to the backend. Higher sampling rates produce more network traffic.

Head-based and parent-based sampling have nearly the same network egress usage. The occasional small deviations can be attributed to normal variance in the data. This confirms that the cost of exporting traces depends on how many are sampled and not on which strategy is used to make the decision.

### 5.2.7 Throughput

Throughput is the number of successful requests per second the service actually handled. Both strategies achieve the same throughput at every load level and sampling

rate. Small differences at high load reflect the workload itself and not the sampler. Compute-heavy endpoints are affected by CPU limitations and long-running ones are bound by their artificial delays.

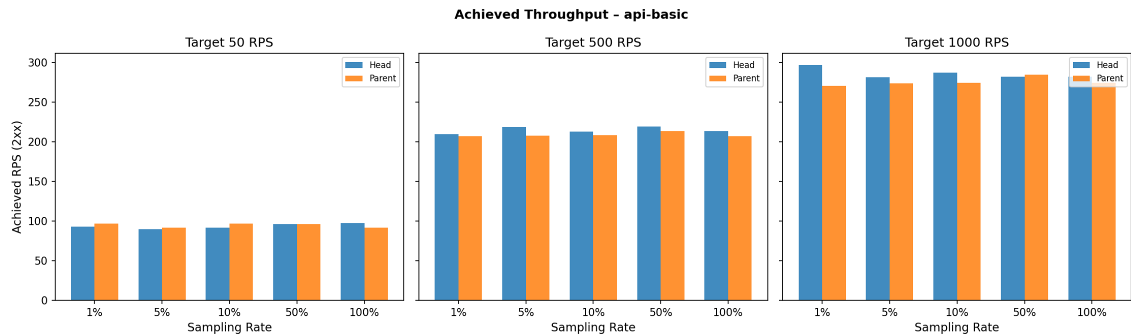


Figure 5.8: Achieved throughput for the *api-basic* workload under increasing sampling rates and load levels. Head-based and parent-based sampling show comparable capacity.

### 5.2.8 Trace Completeness

Both sampling strategies produced complete traces in every run. No spans were dropped and trace volume matched the configured sampling rates during experiment runs. This confirms both samplers work as intended.

As stated in other chapters, the sampling decision only matters when services make calls to other services. Head-based sampling lets each service decide independently, which can leave traces incomplete in a multi-service setup. Parent-based sampling inherits the parent's decision so the full trace is either sampled or not.

This experiment did not include calls to other services but the propagation behavior is the main reason to prefer parent-based sampling in a distributed system. The key finding here is that this consistency comes without any visible resource overhead.

### 5.2.9 Summary and Implications

The results from RQ2 show that head-based and parent-based sampling perform the same across all measured metrics which were CPU, latency, memory, network and throughput. There seems to be no penalty for using parent-based sampling.

This means that the choice between the two sampling strategies should be based on architectural needs rather than performance concerns. For multi-service platforms where visibility into calls between services is needed, parent-based sampling gives that without noticeable overhead in resource usage.

This also sets up the comparison for RQ3, which looks at whether adaptive sampling is feasible without breaking this predictable overhead profile. Adaptive sampling will be using logic where sampling rate changes based on runtime conditions.

## 5.3 Research Question 3

This subsection looks at whether adaptive sampling can be implemented and run in a Kubernetes-hosted Go microservice. The idea is to see if the span rate can be adjusted by checking if there are any error events inside the service.

Usability is evaluated from three angles: if the sampling behavior stays stable, if the service performance is affected and whether adaptive sampling adds observability when the service encounters error events. The data captured from adaptive sampling is compared with data from head-based sampling at the same configured experiment rates.

### 5.3.1 Effective Sampling Behavior

The adaptive sampler built into the Go service dynamically adjusts its rate based on observed error rates inside the service. It recalculates the sampling value every

30 seconds. To check whether this produces predictable behavior, the actual span ingestion rate was compared against the configured base sampling rate from head-based strategy.

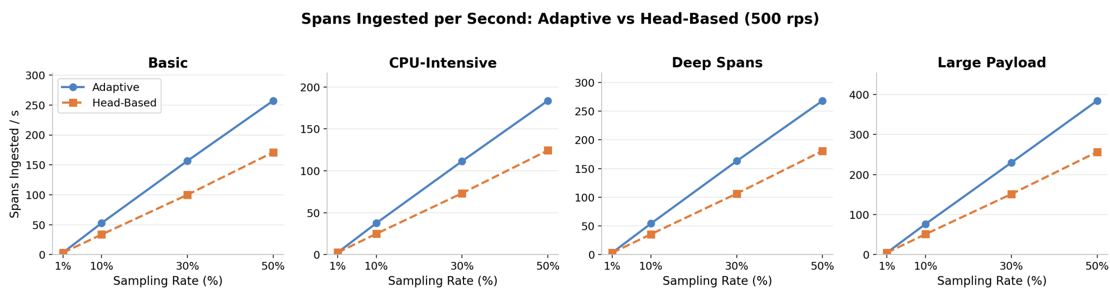


Figure 5.9: Spans ingested per second for adaptive and head-based sampling at 500 requests per second. Each panel shows a separate endpoint workload. Adaptive sampling produces more spans at every rate above 1%, with the gap widening consistently as the base rate increases.

At low base rates (1%), adaptive and head-based sampling produce almost the same span volumes. As the base rate goes up, adaptive sampling starts to produce a higher sampling rate compared to head-based. This is expected since the adaptive span multipliers kick in when errors are crossed, pushing the span & trace rate above the configured base.

The increase stays consistent across endpoints and load levels. No variations or sudden jumps in the rate were observed during the measurement window.

### 5.3.2 Span Volume and Backend Impact

Since adaptive sampling ends up with a higher effective rate, it also sends more spans to the tracing backend.

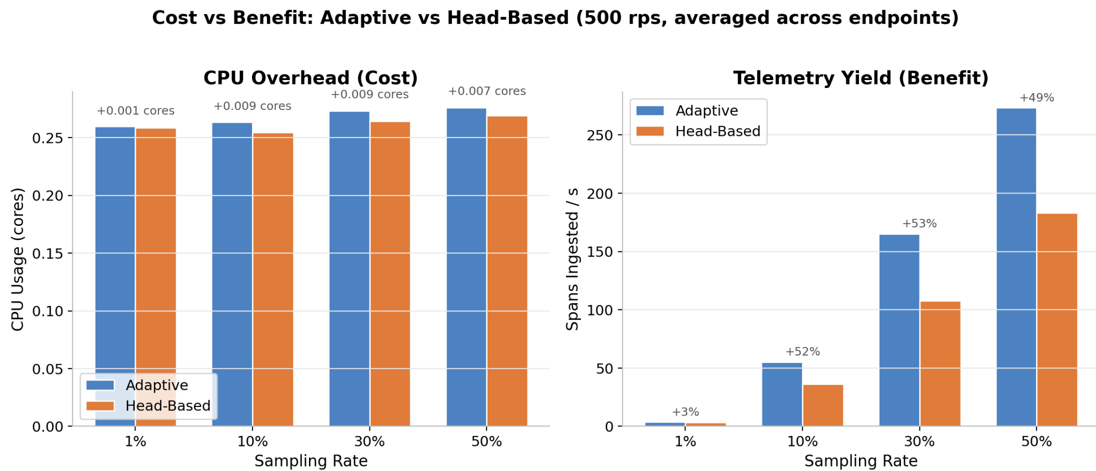


Figure 5.10: CPU usage and span yield for adaptive versus head-based sampling at 500 requests per second, averaged across all endpoints. The left panel shows that CPU cost in cores stays nearly identical between strategies at every sampling rate. The right panel shows the span yield advantage of adaptive sampling, which reaches approximately 50% at rates of 10% and above.

The difference is most visible at higher base rates (30% and 50%). Head-based sampling scales proportionally with the configured rate as expected. Adaptive sampling produces a steeper increase because the multipliers push the effective rate higher.

No span drops or exporter errors were observed under any of the experiment configurations. Tempo as the tracing backend handled all the span volumes without issues during the experimental runs.

### 5.3.3 Application-Level Performance

CPU usage was compared across adaptive and head-based configurations to see whether the dynamic rate adjustments add overhead to the service itself.

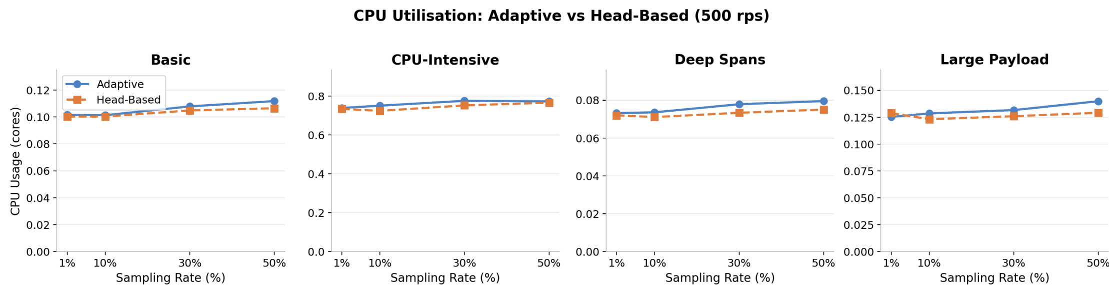


Figure 5.11: CPU usage in cores for adaptive and head-based sampling at 500 requests per second, split by endpoint. Both strategies follow each other closely within each workload group across all tested sampling rates.

Across all workloads and request rates, CPU usage is driven by the endpoint characteristics rather than the sampling strategy. Within each workload group, adaptive and head-based sampling show comparable CPU numbers. The differences fall within normal run-to-run variance and do not point in a consistent direction.

Latency percentiles (p95 and p99) also stay stable across strategies. No latency increases were observed that would come from just adaptive sampling. Memory usage stayed around the normal ranges across all experiment configurations.

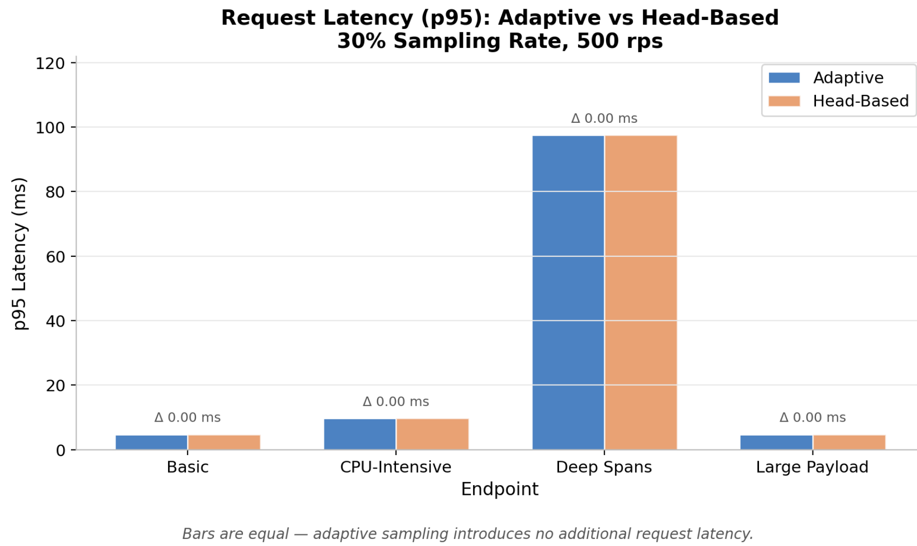


Figure 5.12: p95 request latency for adaptive and head-based sampling at 30% sampling rate and 500 requests per second. The bars are equal across all four endpoints, confirming that adaptive sampling adds no measurable latency overhead compared to head-based sampling.

The results confirm that the adaptive sampler does not add measurable overhead to the application compared to static head-based sampling. The computation it does to check the service condition and recalculate the sampling rate every 30 seconds is not noticeable.

### 5.3.4 Diagnostic Coverage

To check whether adaptive sampling actually helps with diagnostics, the span amount delivered was compared against head-based sampling at the same configured base rate. Figure 5.10 shows this directly: at rates of 10% and above, adaptive sampling captures approximately 50% more spans per second without adding CPU cost. More spans means a higher chance that the requests worth investigating actually have a trace recorded against them.

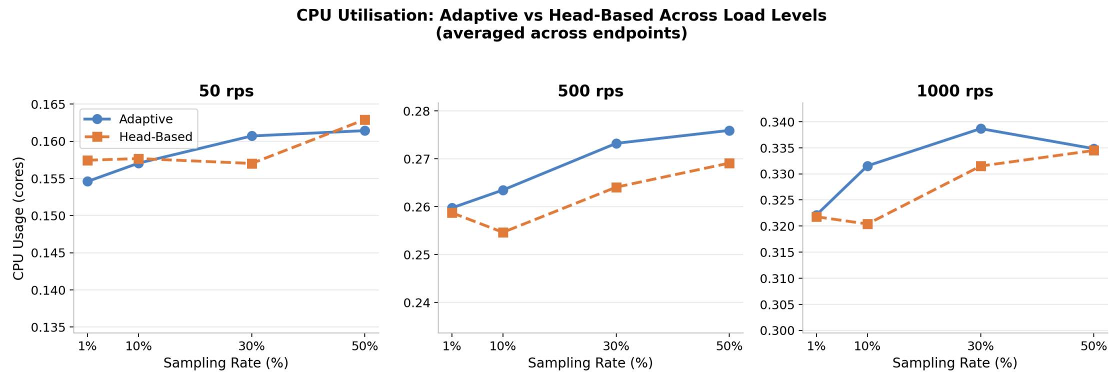


Figure 5.13: CPU usage in cores averaged across endpoints as a function of sampling rate at 50, 500 and 1000 requests per second. Adaptive and head-based sampling remain comparable across all load levels, showing that overhead does not change radically with the applied request rates.

When error conditions triggered adaptive multipliers, the effective sampling rate increased above the configured base. This means that those events were more likely to have a trace recorded against them compared to head-based sampling at the same rate. The adaptive design targets exactly the requests where visibility matters.

At higher base rates, adaptive sampling approaches the coverage of the 100% configuration during abnormal conditions while using a lower overall average rate. This is the main practical benefit since better visibility into problems is available without sampling everything all the time.

### 5.3.5 Feasibility Assessment

In this experiment, feasibility means that adaptive sampling can be deployed without causing unstable behavior to the service or adding unexpected performance costs. Based on the experiments:

- The adaptive sampler does not add measurable CPU, latency or memory over-

head compared to head-based sampling.

- The rate adjustments do not spike and stay within reasonable limits.
- Span volume does increase compared to static sampling at the same nominal rate, which is expected.
- More visibility into error requests when adaptive triggers are activated.

Within the scope of this single-service experiment, adaptive sampling works and is feasible to deploy. It helps in gathering spans during error states in the service. The trade-off is that the backend gets more spans than it would with static sampling at the same base rate, but the service itself does not suffer from it.

It should be noted that the adaptive sampling rate is done with only the test service in mind. Different adaptations in other languages might bring different results.

## 5.4 Research Question 4

The results from RQ1 through RQ3 showed that tracing overhead stays stable throughout the different configurations and workloads. Memory usage and latency did not show noticeable increases regardless of which sampling strategy was used or how high the sampling rate was set. CPU overhead did increase with the sampling rate, reaching up to about 20% for lightweight workloads at full sampling.

No spans were dropped and throughput held even at 100% sampling. This is good from a performance standpoint but it also means that nothing on the system side stops someone from collecting a large amount of trace data. Running at full sampling is feasible but what gets stored in the backend becomes a data governance question rather than a performance issue. Centralized telemetry backends hold similar kinds of data to audit log and should be treated with the same care [25].

This subsection goes through the security aspects of different sampling strategies and tracing in general.

### 5.4.1 Sampling Rate and Risk of Data Exposure

Telemetry volume scales linearly with the sampling rate, meaning that more sampling means more data sitting in the backend and its storage. Even when nobody intentionally records payloads, traces carry context such as routes that the service called, service names and versions, time data, attributes in span and error messages.

When considering tracing in a service, performance should be one factor and the data gathered another. At low sample rates, the odds of capturing something sensitive on any given request or workload are small. At 100%, everything is kept and traced.

### 5.4.2 Adaptive Sampling and Sensitive Events in Service

RQ3 showed that the adaptive sampler raises the effective sampling rate when it detects abnormal or error conditions in the service. From a security perspective this is useful since anomalies are captured in more detail which makes debugging the service easier.

The problem is that out of the ordinary conditions may often include sensitive information. Things like failed authentication attempts, authorization errors, input validation failures and internal exception traces can show up when microservices talk to each other.

Since everything the sampler decides to capture actually gets stored, boosting the rate during those events increases the chance of retaining error messages or contextual identifiers in the tracing backend.

The idea for the adaptive sampler was to capture unexpected behavior. This unexpected behavior is usually where sensitive data tends to come from. This means

that adaptive sampling should be paired with attribute filtering. This way the amount of sensitive data stored would be minimized.

### 5.4.3 Centralized Trace Storage and Multi-Tenant Risk

All experiments were run against a centralized Grafana Tempo backend deployed in the same Kubernetes cluster as the test service. Centralized storage makes correlating traces with the service easy but it also creates a problem with data sitting in one place. In a system accessed by multiple people, unrestricted access to the tracing backend could expose sensitive information about the system's internals.

Even if individual traces would not contain personal data, the combined dataset reveals information about the system architecture and its operations. This kind of information could be used to map out the system without the need to search or analyze the actual services. Causal datasets like provenance records are known to reveal detailed system relationships and need confidentiality and integrity controls [26].

Since no spans were dropped during experiments, the backend stored everything it received. Combined with centralized storage and higher sampling, this increases how much data could be exposed if the backend is compromised. Retention and access policies are the main controls here, meaning that those limits have to come from policy decisions rather than from constraints on the system.

### 5.4.4 Context Propagation and Topology Exposure

The sampling strategies tested relied on context propagation through `traceparent` headers. The overhead was barely noticeable which means that it is possible to propagate trace context across every service in a cluster. But universal propagation also means that service relationships become visible throughout the system.

Trace context links requests across microservices, exposes internal service rela-

tionships and makes it possible to reconstruct full call graphs inside the system. Trace IDs themselves are not sensitive but predictable or improperly validated context propagation could enable correlation attacks or let an attacker inject trace context. On top of the identifiers, trace metadata reveals topology. Timing relationships and span names expose how services are connected even without looking at payloads. Research has shown that metadata alone can leak structural information about systems and their communication patterns [21], [30].

Since context propagation costs almost nothing computation wise, it can be enabled across every service. But doing so means that the internal structure of the system becomes visible in the traces, including from externally accessible endpoints if they are connected to the services or system.

### 5.4.5 Mitigation Mechanisms

Based on the experimental results and the trade-offs described above, there are five areas where adding control might benefit the security posture of the service:

- **Minimizing data:** Only record attributes that are actually useful for diagnosing problems. The experiments ran fine with a minimal attribute set, which shows that capturing everything is not necessary to get useful traces.
- **Redaction and filtering:** Strip or hash sensitive fields before spans leave the service. This is more of a factor in adaptive sampling since higher error rates push up the volume of captured traces and those traces are more likely to contain error details.
- **Access control:** Tracing backends need access controls so teams can only query traces from their own services.
- **Retention policies:** Shorter retention windows would reduce long-term exposure without affecting how useful traces are day to day. Since span volume

scales predictably with sampling rate, cutting retention is one of the more straightforward ways to limit stored data.

- **Sampling as a data control:** Sampling rate is also a decision on data collection. Lower rates reduce how much gets recorded and adaptive sampling can be scoped to specific endpoints rather than applied to the whole service.

### 5.4.6 Overall Trade-off Assessment

Higher sampling rates improve observability but increase the amount of metadata stored in the backend. Adaptive sampling improves visibility into anomalies and error conditions but also concentrates data capture around sensitive events. Centralized trace storage simplifies analysis by consolidating data in one location but it also increases the risk of data exposure. Universal context propagation improves correlation but reveals service relationships inside the system. These limits are not enforced automatically by the system. Sampling, redaction, filtering, access controls and data retention must all be explicitly configured. When controls and filters are in place, the risks described above are more manageable.

# 6 Discussion and Conclusions

This chapter summarizes the main findings of the thesis. The chapter also discusses the limitations encountered and directions for future work. The goal is to put the experiment results into a broader context and to reflect on what they mean for practitioners deploying distributed tracing in production environments.

## 6.1 Summary of Findings

The evaluation of RQ1 showed that tracing overhead is predictable in basic sampling strategies. CPU usage increases with the sampling rate and reaches around 20% for lightweight workloads at full sampling. For compute-heavy workloads the relative overhead is much smaller because the application itself already uses most of the CPU. Memory usage and latency remained close to the no-tracing baseline across all configurations. This is notably lower than the 42% CPU overhead reported by Sandberg [36] and the 71.33% reported by Karkan [35] under always-on sampling. The difference is likely attributable to the single-service scope and Go's low-overhead concurrency model, but the directional finding is consistent: sampling rate is the primary driver of tracing overhead.

RQ2 compared head-based and parent-based sampling. Both strategies performed similarly across all measured metrics: CPU usage, latency, memory usage, network egress and throughput. The difference between the strategies is therefore architectural rather than performance-related. Parent-based sampling keeps traces

consistent across service boundaries by inheriting the sampling decision from the parent span. This consistency does not cause measurable overhead. This is consistent with Nõu et al. [34], who similarly found that overhead differences between tracing configurations are driven by sampling rate rather than by the specific sampling strategy used.

For RQ3, a custom adaptive sampler was tested that adjusts the sampling rate based on error rates and latency inside the service during runtime. It did not add measurable overhead compared to head-based sampling. When errors in the service triggered the multipliers, it captured a higher share of those requests than a static sampler would at the same base rate. The tracing backend handled all the extra spans without dropping any of them.

RQ4 goes over the privacy and security side of these results. Because tracing overhead is low, the sampling rate becomes more of a data governance decision rather than a performance issue. Centralized trace storage holds a lot of metadata. Context propagation exposes service relationships. The system does not enforce any limits on its own, so sampling rate, redaction, access control and retention all have to be configured.

## 6.2 Discussion

The main finding of these experiments is that tracing overhead in a Go microservice on Kubernetes is predictable enough to plan around. The tested sampling strategies all performed at the same service level. This shifts the question away from overhead and toward which strategy best fits the observability requirements. The decision comes down to what kind of trace consistency and diagnostic coverage is needed.

Parent-based sampling keeps full traces together when calls go through multiple services inside the system. It inherits the sampling decision from the parent span, so a trace is either kept completely or dropped completely. The results show this does

not introduce measurable additional overhead compared to head-based sampling. For any multi-service deployment that makes it the more practical strategy.

The adaptive sampler built for this thesis is a simple one. It checks error rates and latency every 30 seconds and raises the sampling rate when errors occur. Overhead of running that check was not visible in the CPU or latency measurements. More advanced systems such as Trastrainer [37] use runtime anomaly signals to adjust sampling budgets dynamically across multi-service environments. The sampler in this thesis follows the same principle at a smaller scale: adjusting the rate based on observed errors and latency without measurable overhead cost. More advanced versions could use more signals or adjust rates per endpoint, but the basic approach works and the resource cost was low in the experiments.

The privacy and security side is where risks may be overlooked. Tracing is cheap to run, which makes it easy to leave at a high sampling rate without considering what accumulates in the backend. Nothing stops the backend from gathering a large amount of trace data by default. That means the controls and filters have to come from policy decisions: what gets collected, how long it is kept and who can query it. Data governance considerations are often addressed later than the technical aspects of tracing, and this is worth keeping in mind. [27], [28].

Compared with previous OpenTelemetry overhead studies, the results of this thesis support the general conclusion that sampling rate is the primary factor controlling tracing overhead. The measured overhead is lower than in several earlier studies, which is likely attributable to the narrower single-service scope, the Go runtime's low-overhead concurrency model, and the absence of cross-service trace propagation chains. The contribution of this thesis is therefore not to contradict earlier work, but to provide a controlled single-service Kubernetes baseline and to show that adaptive sampling can improve diagnostic coverage without measurable additional application-level overhead in this setting. The adaptive sampler is con-

siderably simpler than systems such as Trastrainer [37] and TraceMesh [38], which target multi-service environments and use richer runtime signals, but the results point in the same direction: selective sampling decisions add little overhead while improving the relevance of collected traces.

## 6.3 Limitations

The experiments ran on a single Go microservice. This makes it easy to isolate the tracing overhead, but it does not show how things change when spans reach other services or when sampling decisions propagate through multiple services. In a real multi-service setup the aggregate overhead and the behavior of parent-based sampling could look different from what was measured.

The test environment used a dedicated cluster with fixed resource limits and no autoscaling. Production environments typically carry additional workloads and share resources across services. These conditions could interact with tracing overhead in ways that the controlled setup did not have. It is also worth mentioning that the environment ran on virtualized platforms, which may also be a factor.

The adaptive sampler is a custom implementation made for this thesis. It uses a simple interval-based check with fixed multipliers. A production version would likely need more thought. This could be for example per-endpoint settings and some way to read feedback from the tracing backend.

The tracing pipeline was validated through span counts and drop metrics. Collector-internal metrics were not available because the collector was not configured to expose them to Prometheus. Pipeline reliability was assessed based on end-to-end span delivery rather than internal collector metrics.

The privacy and security analysis in RQ4 is based on the experimental setup and existing literature. It was not tested against a real multi-tenant production environment. The trade-offs described are grounded in the data but how much they

matter in practice depends on the deployment context.

## 6.4 Future Work

The most obvious next step is running the same experiments across multiple services. That would show how overhead adds up along a call chain and how parent-based sampling behaves when decisions have to carry over service boundaries. It would also be beneficial to see if the adaptive sampler holds up in multiple services.

Tail-based sampling was not part of these experiments because it needs to buffer full traces at the collector before making a sampling decision. Adding it would be useful, especially for setups where the sampling decision depends on what happened at the end of the request.

The adaptive sampler could be extended to use more signals. Possibilities would be for example per-endpoint sampling policies, feedback from the tracing backend on queue or tying the rate adjustment to incident severity from an alerting system rather than just reading errors and latency from the service itself.

On the privacy side, it would be useful to measure the overhead of specific redaction and filtering implementations in the OpenTelemetry pipeline. Understanding what attribute processing costs at the SDK or collector level would support the sampling-focused results in this thesis.

Running the same experiments in other languages and runtimes would also show how much of what was found is specific to Go and even Kubernetes. Go's concurrency model and garbage collector behavior may contribute heavily to the stability observed in experiments. Other runtimes could behave differently under the same conditions.

## 6.5 Concluding Remarks

Distributed tracing in Go-based microservices on Kubernetes adds stable and predictable overhead. The choice of sampling strategy does not really matter for service performance in Go. The decision should come down to what trace consistency and observability coverage is needed for the service and the system.

The adaptive sampler worked well and captured more of the requests worth investigating. The more significant trade-off is not about CPU or latency. It is about how much data accumulates in the backend and who has access to it. If sampling, filtering, access control and retention are set, tracing is a useful and manageable addition to any Go microservice on Kubernetes.

# References

[1] R. T. Fielding, “Architectural styles and the design of network-based software architectures”, Ph.D. dissertation, University of California, Irvine, 2000.

[2] N. Dragoni et al., “Microservices: Yesterday, today, and tomorrow”, in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds., Springer, 2017, pp. 195–216.

[3] A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices architecture enables devops”, *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[4] P. Di Francesco, I. Malavolta, and P. Lago, “Research on architecting microservices: Trends and focus”, in *Proceedings of the IEEE International Conference on Software Architecture (ICSA)*, Gothenburg, Sweden: IEEE, 2017, pp. 21–30.

[5] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, omega, and kubernetes”, *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[6] A. Verma et al., “Large-scale cluster management at google with borg”, in *Proceedings of EuroSys*, Bordeaux, France: ACM, 2015.

[7] Cloud Native Computing Foundation, *CNCF cloud native definition v1.0*, GitHub, 2018. [Online]. Available: `https://github.com/cncf/toc/blob/main/DEFINITION.md`.

[8] Kubernetes Authors, *Kubernetes documentation*, Cloud Native Computing Foundation, 2024. Accessed: Feb. 19, 2026. [Online]. Available: `https://kubernetes.io/docs/`.

[9] M. Fowler and J. Lewis, *Microservices: A definition of this new architectural term*, martinfowler.com, 2014. [Online]. Available: `https://martinfowler.com/articles/microservices.html`.

[10] B. H. Sigelman, L. A. Barroso, M. Burrows, et al., "Dapper, a large-scale distributed systems tracing infrastructure", Google, Tech. Rep., 2010.

[11] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review", *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018.

[12] C. Sridharan, *Distributed Systems Observability*. O'Reilly Media, 2018.

[13] Prometheus Authors. "Prometheus documentation", Accessed: Feb. 19, 2026. [Online]. Available: `https://prometheus.io/docs/`.

[14] R. R. Sambasivan, R. Fonseca, I. Shafer, and G. R. Ganger, "So, you want to trace your distributed system? Key design insights from years of practical experience", Carnegie Mellon University, Technical Report CMU-PDL-14-102, 2014.

[15] J. Kaldor, J. Mace, et al., "Canopy: An end-to-end performance tracing and analysis system", in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, Shanghai, China: ACM, 2017, pp. 34–50.

[16] OpenTelemetry Authors, *OpenTelemetry specification*, Cloud Native Computing Foundation, 2024. [Online]. Available: `https://opentelemetry.io/docs/specs/otel/`.

[17] Grafana Labs, *Grafana Tempo — distributed tracing backend*, Grafana Labs, 2024. [Online]. Available: `https://grafana.com/docs/tempo/`.

[18] Y. Gan et al., "An open-source benchmark suite for microservices", in *Proceedings of ASPLOS*, Providence, RI, USA: ACM, 2019, pp. 3–18.

[19] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework", in *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, USA: USENIX Association, 2007, pp. 271–284.

[20] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling", in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, USA: USENIX Association, 2004, pp. 259–272.

[21] W3C, *Trace context — W3C recommendation*, World Wide Web Consortium, 2021. [Online]. Available: `https://www.w3.org/TR/trace-context/`.

[22] P. Las-Casas, G. Papakerashvili, J. Mace, and R. Fonseca, "Sifter: Scalable sampling for distributed traces, without feature engineering", in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, Santa Cruz, CA, USA: ACM, 2019, pp. 312–324.

[23] Y. Shkuro, *Mastering Distributed Tracing*. Packt Publishing, 2019.

[24] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems", in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, USA: ACM, 2015, pp. 378–393.

[25] B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics", *ACM Transactions on Information and System Security*, vol. 2, no. 2, pp. 159–176, 1999.

[26] A. Bates, D. ( Tian, K. R. B. Butler, and T. Moyer, "Trustworthy whole-system provenance for the Linux kernel", in *Proceedings of the 24th USENIX Security Symposium*, Washington, D.C., USA: USENIX Association, 2015, pp. 989–1004.

[27] *Regulation (eu) 2016/679 (general data protection regulation)*, Official Journal of the European Union, 2016.

[28] A. Cavoukian, *Privacy by design: The 7 foundational principles*, Information and Privacy Commissioner, Ontario, Canada, 2011.

[29] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds", in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, USA: ACM, 2009, pp. 199–212.

[30] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr, "Secure network provenance", in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal: ACM, 2011, pp. 215–230.

[31] K. Hashizume, D. G. Rosado, E. Fernández-Medina, and E. B. Fernandez, "An analysis of security issues for cloud computing", *Journal of Internet Services and Applications*, vol. 4, no. 1, p. 5, 2013.

[32] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead", *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019.

[33] E. Politou, E. Alepis, and C. Patsakis, "Forgetting personal data and revoking consent under the gdpr", *Computers & Security*, vol. 79, pp. 220–231, 2018.

[34] A. Nõu, S. Talluri, A. Iosup, and D. Bonetta, "Investigating performance overhead of distributed tracing in microservices and serverless systems", in *Companion of the 16th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2025. DOI: 10.1145/3680256.3721316.

[35] T. Karkan, "Performance overhead of OpenTelemetry sampling methods in a cloud infrastructure", M.S. thesis, DiVA portal, 2024. [Online]. Available: `https://www.diva-portal.org/smash/get/diva2:1867120/FULLTEXT01.pdf`.

[36] F. Sandberg, "Evaluating OpenTelemetry's impact on performance in microservice architectures", M.S. thesis, DiVA portal, 2024. [Online]. Available: `https://www.diva-portal.org/smash/get/diva2:1877027/FULLTEXT01.pdf`.

[37] H. Huang, X. Zhang, P. Chen, Z. He, and Z. Chen, "Trastrainer: Adaptive sampling for distributed traces with system runtime state", *Proceedings of the ACM on Management of Data*, 2024. DOI: `10.1145/3643748`.

[38] Z. Chen, Z. Jiang, Y. Su, M. Lyu, and Z. Zheng, "TraceMesh: Scalable and streaming sampling for distributed traces", in *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, 2024. arXiv: `2406.06975`. [Online]. Available: `https://arxiv.org/abs/2406.06975`.

[39] D. Reichelt, L. Bulej, R. Jung, and A. Van Hoorn, "Overhead comparison of instrumentation frameworks", in *Companion of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2024. DOI: `10.1145/3629527.3652269`.

[40] M. Usman, S. Ferlin, A. Brunstrom, and J. Taheri, "A survey on observability of distributed edge & container-based microservices", *IEEE Access*, vol. 10, 2022. DOI: `10.1109/ACCESS.2022.3193102`.

[41] B. A. Kitchenham et al., "Preliminary guidelines for empirical research in software engineering", *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721–734, 2002.

[42] C. Wohlin et al., *Experimentation in Software Engineering*. Springer, 2012.

[43] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, “Evaluating the accuracy of Java profilers”, in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Ontario, Canada: ACM, 2010, pp. 187–197.

[44] R. Jain, *The Art of Computer Systems Performance Analysis*. Wiley, 1991.

[45] SUSE, *Rancher by SUSE — kubernetes management platform*, SUSE, 2024. Accessed: Feb. 19, 2026. [Online]. Available: `https://ranchermanager.docs.rancher.com/`.

[46] containerd Authors, *Containerd — an industry-standard container runtime*, Cloud Native Computing Foundation, 2024. Accessed: Feb. 19, 2026. [Online]. Available: `https://containerd.io/docs/`.

[47] Cilium Authors, *Cilium — eBPF-based networking, observability and security*, Cloud Native Computing Foundation, 2024. Accessed: Feb. 19, 2026. [Online]. Available: `https://docs.cilium.io/`.

[48] Traefik Labs, *Traefik proxy documentation*, Traefik Labs, 2024. Accessed: Feb. 19, 2026. [Online]. Available: `https://doc.traefik.io/traefik/`.

[49] Grafana Labs, *Grafana alloy documentation*, Grafana Labs, 2024. Accessed: Feb. 19, 2026. [Online]. Available: `https://grafana.com/docs/alloy/`.

[50] Grafana Labs, *Grafana loki documentation*, Grafana Labs, 2024. Accessed: Feb. 19, 2026. [Online]. Available: `https://grafana.com/docs/loki/`.

[51] Grafana Labs, *Grafana documentation*, Grafana Labs, 2024. Accessed: Feb. 19, 2026. [Online]. Available: `https://grafana.com/docs/grafana/`.

[52] J. Dogan, *Hey – http load generator, apachebench (ab) replacement*, `https://github.com/rakyll/hey`, Version 0.1.4, 2024.

# Appendix A Technical Material

Technical material used in this thesis is publicly available in the following GitHub repository:

`https://github.com/vphoikka/thesis-material-ville-pekka-hoikka`

The repository contains:

- Go service used in the experiment
- Kubernetes manifests for deploying the test environment
- Shell scripts used for experiment automation and data collection
- Raw measurement data

Note that any sensitive data such as private repositories or hostnames have been removed.