POZNAN UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTING AND TELECOMMUNICATION
Institute of Computing Science

Bachelor's thesis

PERFORMANCE EVALUATION AND MODELING OF MICROSERVICE-BASED CLOUD SYSEMS

Mateusz Dębski, 148216
Michał Dropiewski, 148189
Mikołaj Felczyński, 147400
Kacper Wiśniewski, 144591

Supervisor
Professor Maciej Drozdowski

POZNAŃ 2024

Karta pracy dyplomowej inżynierskiej

Uczelnia:	Politechnika Poznańska	Profil studiów:	Ogólnoakademicki
Kierunek:	Informatyka	Forma studiów:	Stacjonarne
Studia w zakresie:	-	Poziom studiów:	Pierwszego stopnia

Zobowiązuję/zobowiązujemy się samodzielnie wykonać pracę w zakresie wyspecyfikowanym niżej. Wszystkie elementy (m.in. rysunki, tabele, cytaty, programy komputerowe, urządzenia itp.), które zostaną wykorzystane w pracy, a nie będą mojego/naszego autorstwa będą w odpowiedni sposób zaznaczone i będzie podane źródło ich pochodzenia.

Jeżeli w wyniku realizacji pracy zostanie dokonany wynalazek, wzór użytkowy, wzór przemysłowy, znak towarowy, prawa do rozwiązań przysługiwać będą Politechnice Poznańskiej. Prawo to zostanie uregulowane odrębną umową.
Oświadczam, iż o wyniku prac wskazanych powyżej, a także o innych, w tym tych, które mogą być przedmiotem tajemnicy Politechniki Poznańskiej, niezwłocznie powiadomię promotora pracy.
Zobowiązuję się ponadto do zachowania w tajemnicy wszystkich informacji technicznych, technologicznych, organizacyjnych, uzyskanych w Politechnice Poznańskiej w okresie od daty rozpoczęcia realizacji prac do 5 lat od daty zakończenia wykonania prac.

	Imię i nazwisko	Nr albumu	Data i podpis
Student:	Mateusz Dębski	148216	
Student:	Michał Dropiewski	148189	
Student:	Mikołaj Felczyński	147400	
Student:	Kacper Wiśniewski	144591	

Tytuł pracy:	Ocena i modelowanie wydajności systemów chmurowych opartych na mikroserwisach
Wersja angielska tytułu:	*Performance evaluation and modeling of microservice-based cloud sysems*
Dane wejściowe:	Literatura na temat systemów chmurowych, aplikacji wykorzystujących mikroserwisy, benchmarku Death Star, systemu Jaeger, Open Tracing, docker.
Zakres pracy:	Opracować oprogramowanie wizualizujące wydajność czasową łańcuchów wywołań mikroserwisów obsługujących żądania wykonania usługi na podstawie zebranych logów. Opracować moduł szacujący czasy transferu między mikroserwisami. Opracować metodę szacującą czas obsługi na podstawie intensywności ruchu.
Termin oddania pracy:	31.01.2024
Promotor:	prof. dr hab. inż. Maciej Drozdowski
Jednostka organizacyjna promotora:	Instytut Informatyki

podpis dyrektora/kierownika jednostki organizacyjnej promotora

data i podpis Dziekana

Contents

Chapter 1

Introduction

One could say that today's world is very digitalized. An average citizen that lives in a relatively well-developed country uses dozens of various apps and visits tens of websites daily. From their perspective it's easily accessible, everything works smoothly, providing countless features and hours of entertainment - all while working as if out of thin air. The *"out of thin air"* part, however, isn't what words literally say. Every online application, regardless of it's size or amount of traffic it receives, has to be backed by appropriate infrastructure with certain software architecture. Both have immense impact on the app performance, thus influencing user experience - regardless if we are talking about speed, availability or fault tolerance.

When designing or setting up the infrastructure for an application, there are multiple factors to consider. One of the most crucial aspects that engineers ought to take into account is the required hardware performance. It's also desirable to have the ability to test the general capacity of said infrastructure, although performance can be a difficult thing to measure in a meaningful and comparable way. Luckily, there are solutions in place - there are specialized pieces of software, capable of performing standardized tests on processors, memory, etc. This type of software is called a **benchmark**. Results of those benchmarks are not only useful for consumers trying to decide what piece of hardware they should pick. They are also indispensable for hardware companies that strive to improve their products. Measurable scores that represent performance of the given hardware allow those companies to validate their progress, market the products and compare them against competition. There are tons of solutions in place that are designed to benchmark single hardware units, but modern infrastructure behind the web and mobile applications requires a different approach. This infrastructure often works in a distributed manner - there is no one single machine that does everything, there are multiple ones that try to act and work as unity. The software that runs on this infrastructure tends to be distributed too - microservice architecture paradigm, where the software is divided into smaller, independent computing units is increasingly popular. Because of that, it is desirable to have a benchmark tool capable of testing a production environment for a modern, distributed, microservice-based application. In fact, there are already tools designed specifically for this purpose. **DeathStarBench** [25] is a widely used benchmark that simulates a microservices application and it can be run on distributed system. After deploying the benchmark, it generates artificial traffic and gathers performance statistics. The fact that it's used across numerous organizations all over the world proves that it's a valuable tool, but as many benchmarks, it doesn't always paint the full, true picture of the system.

While it's great to have a general idea of infrastructure performance, there are some use cases that require deeper insight. Performance tuning, searching for bottlenecks or troubleshooting are very challenging tasks in a simpler, monolithic architecture. Attempting it on a distributed system where applications are organized into microservices and just one request can cause a whole chain reaction of microservice calls is nigh to impossible without access to data and adequate tools. In fact, the lack of tools seems to be the main issue. There are projects like **Jaeger Tracing** [9] that allow tracing of requests and operations between services collecting execution data, available to export into JSON format. It looks however, as if there is no way to conveniently visualize and analyze this data. The aforementioned Jaeger Tracing system provides only basic functionalities in this regardt.

As mentioned before, benchmarks and performance insights are incredibly useful for numerous reasons. In this thesis we have closely collaborated with Intel Corporation and their representative, Mr. Przemysław Tyrkiel - Cloud Software Architect, who introduced us to the challenges that he and his team face when working on Intel products for cloud and distributed systems. We learned, that DeathStarBench is great to get the general idea of system performance, and Jaeger can be used for visualization purposes, but there are missing pieces in this software. Some features, especially when it comes to visual presentation or statistical analysis could provide useful insights.

1.1 The purpose and scope of the thesis

The main objective of this thesis is development of software capable of visualizing and analyzing whole chains of microservices calls. In addition, it should provide features like calculating communication times between services and performance statistics for each operation within services. This will allow a better insight into modern, distributed, microservices applications, making it easier to find bottlenecks, errors and fine-tune software. Statistics of various operations will also be useful to understand general performance of the infrastructure and application components. As an input, the application will use telemetric data in JSON format collected by Jaeger Tracing tool to fully leverage it's untapped potential.

1.2 Tasks distribution

The following tasks were realized by each group member:

- Mateusz Dębski was responsible for trace grouping, designing interfaces, visualizing callGraph, implementing table, headers, managing smaller components such as GroupSelector and FileUploader also dealing with 'Negative start times traces'.
- Michał Dropiewski was responsible for implementing the transfer time algorithm and visualizing it in the form of a directed graph within a web application.
- Mikołaj Felczyński was responsible for constructing components such as scatter plots and histograms, compiling statistics, testing the system's functionality, addressing bugs, and enhancing the software.
- Kacper Wiśniewski was responsible for implementing initial architecture and containerization, contributed to developing the logic for communication time calculation, worked on developing test data generator and performed manual tests to validate the functional correctness of software

Przemysław Tyrkiel from Intel Corporation provided us with substantive support, and played the role of a client.

1.3 Chapters contents

Further organization of the thesis is as follows:

- **Chapter 2** serves as a theoretical introduction to the concepts used in this thesis. It lays foundation for understanding of further chapters.
- **Chapter 3** aims to elaborate on input data format used by the application, what data does it contain and how it is processed.
- **Chapter 4** describes our step-by-step approach in designing the solution with consideration of functional and non-functional requirements, proposed architecture, interface design.
- **Chapter 5** dives deep into the details of the system implementation. It covers effects of our work and technologies used to achieve the goal.
- **Chapter 6** is a concluding chapter - a summary of our final product, achieved goals and possible improvements.

Chapter 2

Performance evaluation for modern IT infrastructure

This chapter serves as introduction into the theory of performance testing of microservice cloud systems. It aims to establish a theoretical foundation that underpins our research by explaining concepts used in this thesis.

2.1 Features of modern infrastructure and architecture solutions

In order for IT infrastructure and software architecture to keep up with the challenges of the digital world, the following quality features must be considered:

- **Availability** - many applications are used in various regions. They have to be highly available in every part of the world.
- **Scalability** - both software and hardware infrastructure have to be scalable. By scalability it is meant that an IT solution can adjust itself to constantly changing service demand. Thus, it is not required to keep extensive infrastructure running when the demand is low. And vice versa, when the demand rises, larger (i.e. more powerful) infrastructure will work more effectively. There is no point in having a great product if a company can't scale it to meet the demand.
- **Fault tolerance** - companies relying on income from digital products will effectively loose money in case of downtime. Obviously it's almost impossible to keep the application running at all times. The main objective is to minimize downtime as much as possible. For example, in case of the hardware failure, another machine should take over the workload. If it's an error in the application itself, it's desirable to instantly detect such an error and correct it.
- **Performance** - some applications are handling hundreds of thousands of users every second all around the globe. This requires extensive hardware resources, which capacity meets the speed and resource requirements for the incoming traffic.

To answer these demands, multiple approaches were proposed. We will however focus on the, ones that are the most adopted and heavily used in the industry. It's worth noting, that these approaches do not exclude each other. In fact, more often than not, they co-exist, allowing for obtaining even better results. Thus, in the following we introduce IT solutions widely used to address the aforementioned demands

2.1.1 Microservices

Microservices architecture is an approach where software is developed as a collection of small units called microservices, that communicate with each other. Each microservice is responsible for one service or a small set of homogeneous services. This allows for modular construction of the software while each module, i.e. microservice, allows to seperate concerns. This way applications becomes more fault tolerant - a failure in one component won't necessarily mean a failure of the whole application. One failed component can be restarted withing seconds, because it's a small piece of code with low resource demands. In addition, this paradigm enables great scalability - to the point where single services can be scaled up (i.e. spawned) or down (i.e. stopped) depending on the demand, as opposed to monolithic architecture where the whole application has to be scaled. Let us note, that microservices naturally support distribution of the computational process.

2.1.2 Distributed systems - clusters

Distributed systems consist of multiple independent nodes - usually, in this context, node, which is a unit executing computations, is a physical or virtual machine. They work in a coordinated, unified fashion to achieve common goal - here it's serving applications and their components. Properly designed and maintained distributed systems provide higher availability and better fault tolerance. Those qualities are often further amplified by orchestration tools like Kubernetes [13], that automate composition and deployment of microservice applications. When a partition occurs in the system, modern orchestration tools are able to detect it and run lost workload on the active nodes. To satisfy the varying demand for hardware resources it's possible to add (or remove) more nodes to the system, thus providing perfectly adjusted performance.

2.1.3 Cloud

Cloud could be described as interconnected network of data centers. It's an enormous collection of servers located all over the world, accessible over the internet. Cloud service providers take care of managing the underlying infrastructure, offering their resources and services to users or companies. This way end users can create their own infrastructure, run applications and use managed services without worrying too much about provisioning and administration of machines or network. Most companies adopted a hybrid approach when it comes to their infrastructure. Part of their workload is deployed in the cloud - for example large scale applications with varying traffic that need constant scaling. Although cloud provides more flexibility, on-premise infrastructure will be more cost-effective for cases with lower, constant traffic. Hybrid approach is the most common, but there is no denying that cloud plays a big role in modern IT infrastructure.

2.2 Benchmarking

Benchmark is a computer program or a set of programs that run standardized tests in order to assess performance of a computer system. Usually, after executing such tests, these programs are able to represent system capacity in a comparable way. For example, one of the most popular options for testing desktop CPUs is Cinebench R23 [15]. Cinebench uses 4D image rendering tests to stress all available cores for a period of time. The final result contains Single-Core and Multi-Core scores represented as integer numbers. Thanks to that, it's possible to directly compare competing products on the market or determine progress over product generations. Another example, which is aimed at online transaction processing server solutions, is the TPC [7] benchmark - HammerDB [23]. It runs series of SQL queries against databases in order to test general system performance. In the end it gathers runtime statistics like Transactions per second or New Orders per minute. Again, we strip down the performance of a whole system to just one number, making it possible to directly compare results between different systems. Yet another benchmark example is SPEC CPU [6] that tests CPU performance. SPEC CPU is a collection of algorithms for fixed point (int) and floating point computations considered typical for computing applications. Performance is expressed in relation to some computer conventionally considered standard. Let us note, that there are numerous further benchmarks dedicated to any aspect of computer system - performance, availability, reliability and scalability.

The above examples provide general idea on how benchmarking tests a piece of hardware or a computing system. Hardware companies like Intel Corporation might find it useful to simulate such environment and determine performance benefits of their server products. To achieve that, engineers set up on-premise clusters equipped with appropriate hardware to mimic production-grade infrastructure. Using this configuration they are able to run specific workload, that is a benchmark that focuses on testing interesting features of a distributed system. An example of such benchmark is DeathStarBench.

DeathStarBench

DeathStarBench (in short DSB) [2] is an open-source tool designed for performance testing of cloud microservices. It's being developed by SAIL group at Cornell University. DSB provides a feature to deploy microservice applications that leverage popular technologies. As of now, released types of applications are:

- Social Network - application similar to popular social media platforms like Facebook or Twitter.
- Media Service - an app for browsing and rating movies.
- Hotel reservation - as name suggests, this application mimics services with hotel reservations.

For our purposes we decided to use Social Network, as it was recommended by Intel representative. An architecture of Social Network is shown below on the figure 2.1:

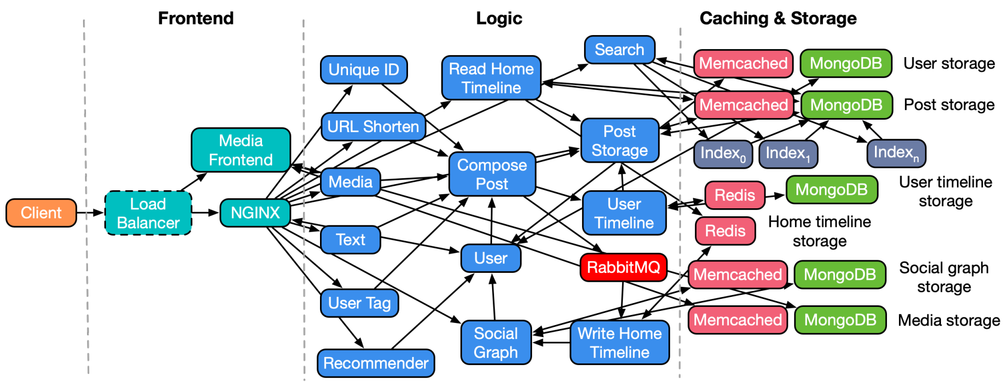


FIGURE 2.1: Social Network architecture [24]

After deploying the Social Network, **wrk2** tool is used to generate constant throughput load on the application. When the test is done, the overall result are presented based on latency histogram and total throughput histogram. The lower the latency for given percentile, the better performance system offers. The project is one of the leading solutions for benchmarking modern infrastructure in the industry. However, basing just on latency histograms it's hard to determine if some microservices act as bottlenecks or where the software could be fine-tuned. Fortunately, DSB deploys Jaeger container alongside the main application, thus utilizing tracing - a practice which will be covered in depth in the next section. For now, let's state that Jaeger is a visualization tool used to address aforementioned issues, at least in theory.

2.3 Tracing

Tracing refers to monitoring and recording the execution of an application to understand its behavior, identify performance bottlenecks, and analyze how different components interact with each other. Tracing is particularly valuable in distributed systems where applications are composed of multiple services that work together to fulfill a request. A tracing system that that we utilized is Jaeger Tracing. It provides useful visualization methods and allows to export telemetry data in JSON format. In the following we explain basic notions used in Jaeger Tracing.

2.3.1 Trace

Trace - is a complete record of the activities and interactions that occur across various components or services in a distributed system when processing a specific request or transaction. A trace basically refers to chain of executed operations. Traces consists of spans.

2.3.2 Span

Span - is a smaller unit of measurement within a trace. It represents a single operation or activity within a larger transaction. Spans contain information such as the start times of an operation, its duration, its predecessors in the calling sequence, name of service executing the action and relevant metadata.

2.3.3 Traces and spans visualized

As mentioned, the trace is a set of spans. It contains a chain of multiple operations that occur in time. Each span represents one singular operation. They can be spotted in a sequential or parallel manner, depending on the request. It's possible for a trace to be forked, i.e. have more than one end. Our implementation aims to provide better readability when visualizing traces and spans.

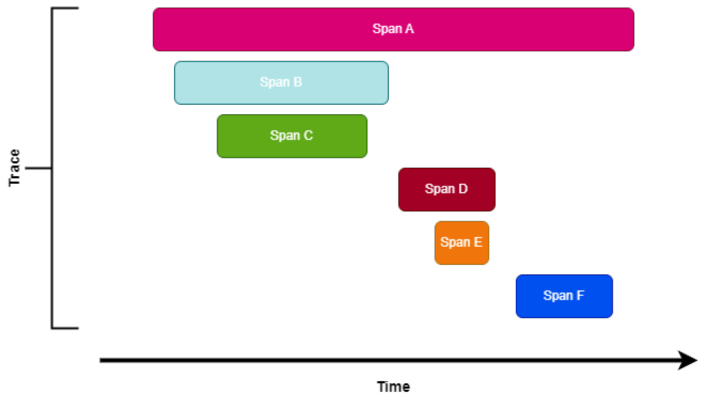
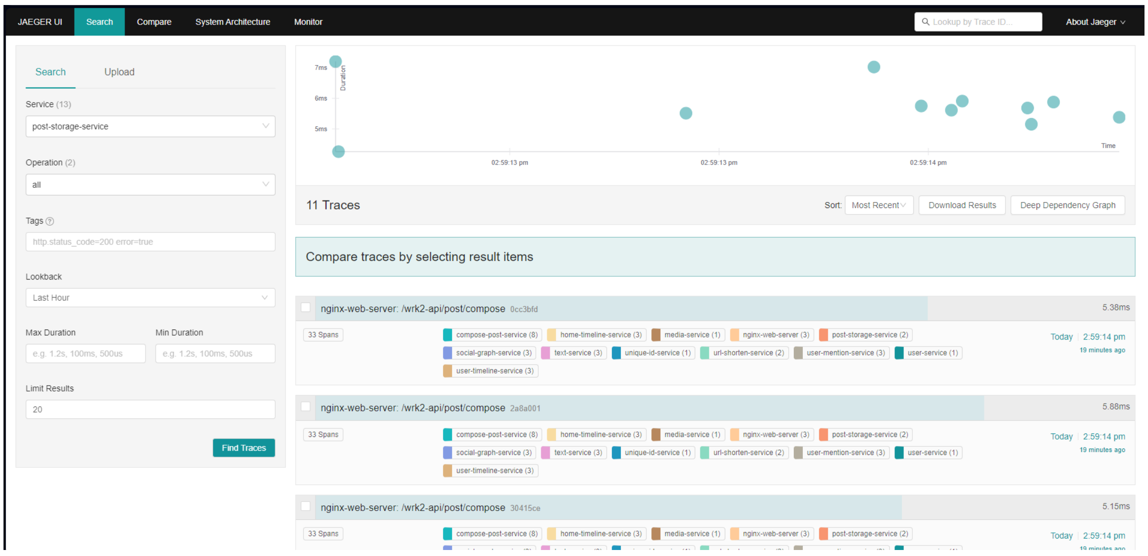


FIGURE 2.2: Traces and spans

2.3.4 Jaeger Tracing

Jaeger Tracing [10] - a CNCF (Cloud Native Computing Foundation) open-source project. According to its documentation: *"Distributed tracing observability platforms, such as Jaeger, are essential for modern software applications that are architected as microservices. Jaeger maps the flow of requests and data as they traverse a distributed system. These requests may make calls to multiple services, which may introduce their own delays or errors. Jaeger connects the dots between these disparate components, helping to identify performance bottlenecks, troubleshoot errors, and improve overall application reliability."* [9] Jaeger Tracing relies on instrumentation within application code. Developers need to code specific instructions within services to make them traceable. When a request enters a service, the instrumentation code generates a unique identifier and span for the trace and associates it with the request. It is then collected and processed by Jaeger Tracing instance. Inside the Jaeger Tracing UI users can gain insight into collected data as shown below on figures 2.3, 2.4, 2.5 and 2.6.



Service (13): post-storage-service
Operation (2): all
Tags: http.status_code=200 error=true
Lookback: Last Hour
Max Duration: e.g. 1.2s, 100ms, 500us
Min Duration: e.g. 1.2s, 100ms, 500us
Limit Results: 20

11 Traces — Sort: Most Recent | Download Results | Deep Dependency Graph

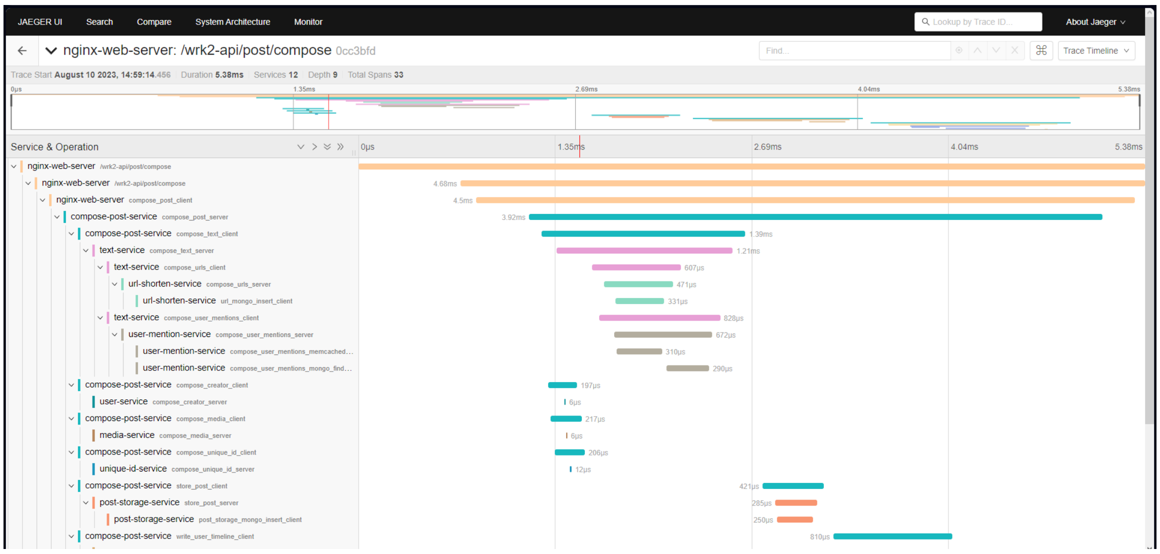
Compare traces by selecting result items

nginx-web-server: /wrk2-api/post/compose 0cc3bfd — 5.38ms
33 Spans — compose-post-service (8), home-timeline-service (3), media-service (1), nginx-web-server (3), post-storage-service (2), social-graph-service (3), text-service (3), unique-id-service (1), url-shorten-service (2), user-mention-service (3), user-service (1), user-timeline-service (3) — Today 2:59:14 pm, 19 minutes ago

nginx-web-server: /wrk2-api/post/compose 2a8a001 — 5.88ms
33 Spans — compose-post-service (8), home-timeline-service (3), media-service (1), nginx-web-server (3), post-storage-service (2), social-graph-service (3), text-service (3), unique-id-service (1), url-shorten-service (2), user-mention-service (3), user-service (1), user-timeline-service (3) — Today 2:59:14 pm, 19 minutes ago

nginx-web-server: /wrk2-api/post/compose 30415ce — 5.15ms
33 Spans — compose-post-service (8), home-timeline-service (3), media-service (1), nginx-web-server (3), post-storage-service (2) — Today 2:59:14 pm

FIGURE 2.3: Main page with search - users can select or compare traces



nginx-web-server: /wrk2-api/post/compose 0cc3bfd

Trace Start August 10 2023, 14:59:14.456 | Duration 5.38ms | Services 12 | Depth 9 | Total Spans 33

```
Service & Operation
nginx-web-server /wrk2-api/post/compose
  nginx-web-server /wrk2-api/post/compose                                4.68ms
    nginx-web-server compose_post_client                                 4.5ms
      compose-post-service compose_post_server                           3.92ms
        compose-post-service compose_text_client                         1.39ms
          text-service compose_text_server                               1.21ms
            text-service compose_urls_client                             607μs
              url-shorten-service compose_urls_server                    471μs
                url-shorten-service url_mongo_insert_client              331μs
            text-service compose_user_mentions_client                    828μs
              user-mention-service compose_user_mentions_server          672μs
                user-mention-service compose_user_mentions_memcached...  310μs
                user-mention-service compose_user_mentions_mongo_find... 290μs
        compose-post-service compose_creator_client                      197μs
          user-service compose_creator_server                            6μs
        compose-post-service compose_media_client                        217μs
          media-service compose_media_server                             6μs
        compose-post-service compose_unique_id_client                    206μs
          unique-id-service compose_unique_id_server                     12μs
        compose-post-service store_post_client                           421μs
          post-storage-service store_post_server                         285μs
            post-storage-service post_storage_mongo_insert_client        250μs
        compose-post-service write_user_timeline_client                  810μs
```

FIGURE 2.4: Spans view - user can inspect spans collected within selected trace

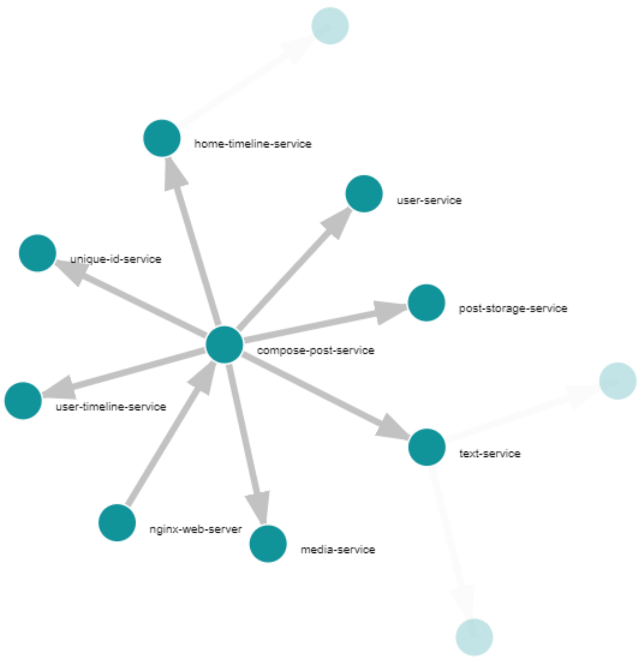


FIGURE 2.5: Force directed graph view

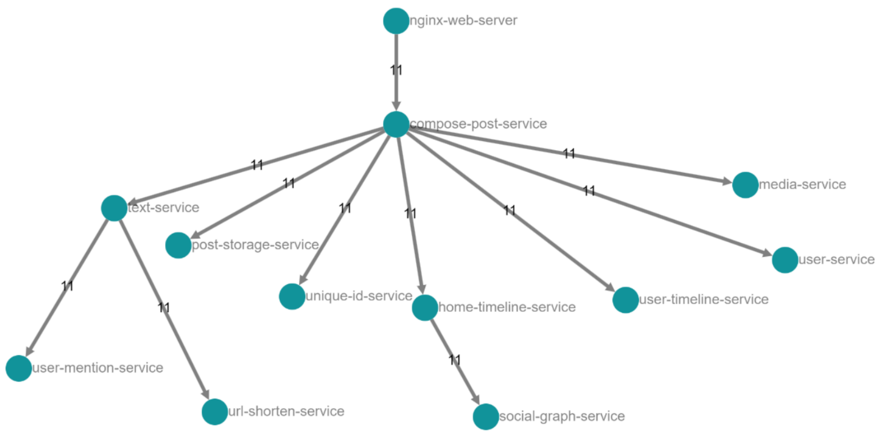


FIGURE 2.6: Directed acyclic graph view

2.3.5 OpenTracing

OpenTracing [5] - an open-source initiative, aimed to furnish vendor-neutral APIs and instrumentation tailored for distributed tracing. In the realm of distributed cloud-native applications, understanding the performance of requests across services proves challenging for engineering teams. This is where the role of distributed tracing becomes pivotal and OpenTracing tried to address this issue.

2.3.6 OpenTelemetry

OpenTelemetry [11] - a project which is a part of the CNCF (Cloud Native Computing Foundation), standardizes the creation and collection of telemetry data (logs, metrics, traces). Born from the merger of OpenTracing and OpenCensus, it comprises APIs, SDKs, and client libraries for vendor-agnostic, exportable telemetry data from application code. **Jaeger is a project within the OpenTelemetry ecosystem.**

float listings adjustbox array caption

Chapter 3

Tools and input analysis

In this chapter we introduce tracing data concepts, the ways of obtaining the tracing data, its format, and the idea of transfer time that substitutes the measurement of communication times

3.1 Schema of creating data to visualization

The information flow in the distributed application is shown in Figure 3.1. This benchmark sends a significant number of queries from multiple processes, with parameters determined by the person executing the benchmark. The tested application processes and monitors this information through Jaeger. Ultimately, we retrieve and analyze data from Jaeger monitor to gain insights into the application's performance under the specified benchmark conditions.

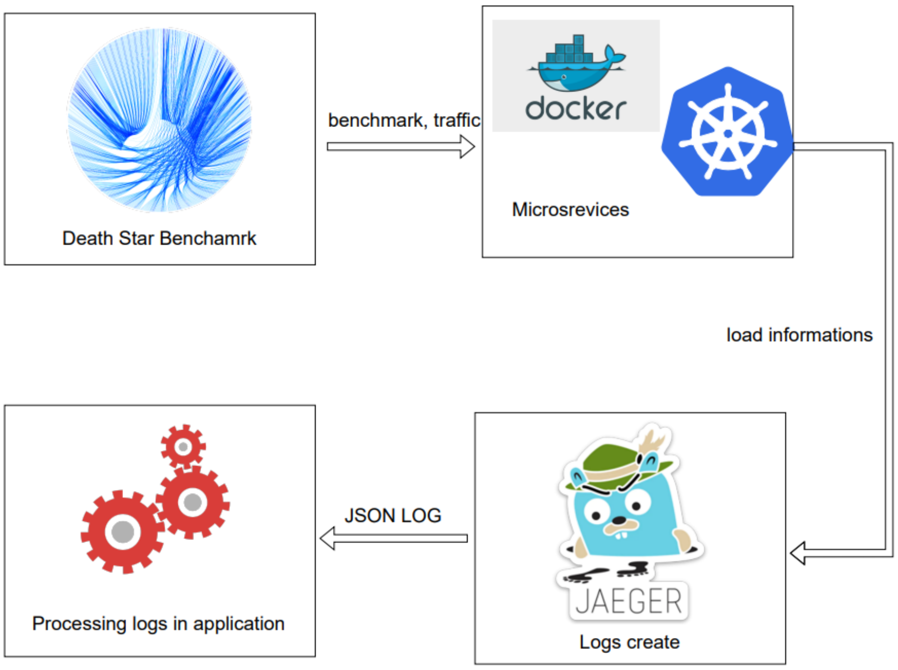


FIGURE 3.1: Graph of operations

3.2 Data to visualize

3.2.1 Data from Jaeger

For the purpose of data, logs from the Jaeger API are required. These logs contain records of the request that the application had to process. Ideally, these data should be extracted from Jaeger during times when the application was under load, and the mentioned Death Star Benchmark can serve as such a load. Offline, the logs are less valuable, as it will be more challenging to various anomalies such as operation hang-ups or prolonged duration of specific operations. In the fallowing table 3.1 we list and explain types of data items provided by Jaeger in the JSON log files. This input data is text file with object like structure.

name of data	description	example (how it looks in log)
traceID	Unique number that identifies trace in data.	2fade568645f89b0
spanID	Unique number that identifies a single span in data.	c7c16bf210f5a963
operationName	Name of operation that microservices execute.	compose_user_mentions _client
refType	Description if the span is the child of another span.	CHILD_OF
startTime	This is a time when the span starts running in microseconds.	number representing microseconds
duration	How long span was processing.	number representing microseconds
warnings	Warnings in execution span.	invalid parent span IDs=6c6b020f2ebf36fc

Table 3.1: Data from Jaeger

3.2.2 Relations between spans and traces

Let us note that a span and a trace are connected. Each span has different attributes that are described in the table. One attribute is the "Operation Name" which shows the name of a process executed in a span. It should be noted that multiple spans can have the same operation indicating repetition or shared processes in the traces. It is also important to mention that traces can vary in size meaning that the number of spans in a trace can be different. This variability in trace size suggests a dynamic and varied range of logged operations and processes which could impact the overall analysis of application behavior. In summary, figure 3.2 the hierarchical relationship between traces and spans, the structure of the input data (JSON log files), the properties of spans, and the possibility of variation in trace widths and the number of spans in each trace.

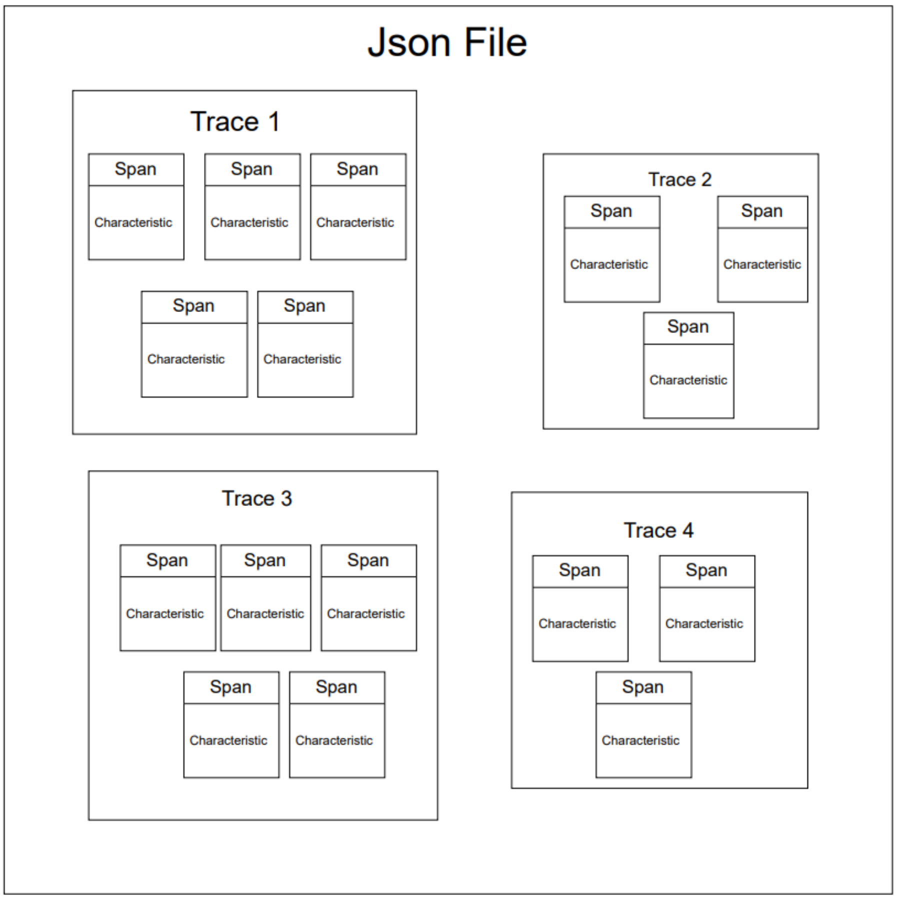


FIGURE 3.2: Visualization of the relation between objects in data.

3.2.3 Groups of traces

The Death Star Benchmark proceeds by repetitively executing a limited set of interaction patterns on the tested microservice system. In the consequence, there is a limited set of microservice sequences that is repetitively present in the JSON file data. Such a sequence of microservice calls will be called a call graph.A call graph shows a calling relationship between spans representing the activity of a preceding microservice and its successor microservice span. A call-graph is a divicted tree. There is an additional abstract data type, that is called a *group*.

The group is a collection of traces that have the same call tree. For example, if one trace calls one span with the name process_1 and this span calls another span with the name process_2, and then in data there is the trace that calls span process_1 and then that span calls process_2, that means the first trace and second trace are in the same group. Every group can contain a lot of traces. In other words, a group is collection of traces representing the same pattern of microservice usage. This is illustrated in figure 3.3.

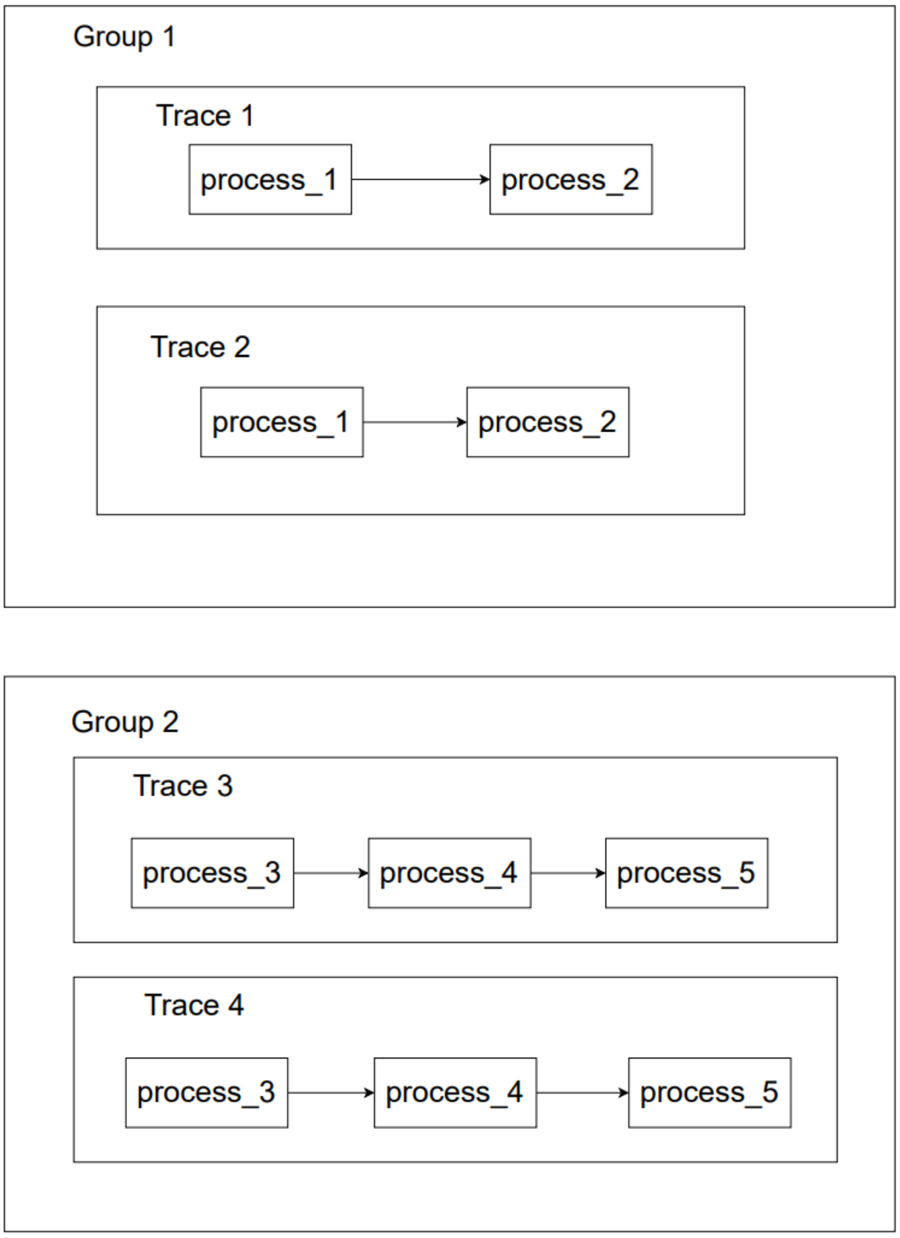


FIGURE 3.3: Relations between trace and group

3.3 Transfer times

There is an introduction to the idea of transfer time, which can be deduced from the obtained data and used to analyze the performance of microservices communication.

3.3.1 The problem of communication time

In the problem of analyzing microservices work it's important to look at the time that span will communicate to another span. Unfortunately, there are no times of communication in the logs from Jaeger.In order to calculate communication time from parent to child it is necessary to know when child microservice is called. In order to calculate communication time from child to parent, child return time that is needed. Neither of the two times is recorded in DSB, Therefore, we propose to calculate a *transfer time*, which is a time of transferring control between actions (spans) that are not executed concurrently. Most of the spans are concurrent, which means that one span is the parent of the next span and the parent span waits for descendants to finish their work, between that spans there is no time gap, which means it is impossible to calculate transfer time. One way to calculate transfer time is when spans are not processing concurrent if one span calls more than one operation. In this situation two separate call tree operations come from the same root this situation allows us it's possible to calculate transfer time. In that method, transfer time doesn't mean the time that the message comes from the span to the second span.

3.3.2 What is the communication time?

Transfer time figure 3.4 is the difference in time between the end of the preceding span and the start of the processing operation of the next span. It should be noted that in transfer time, there is no directed communication between spans, in the sense that spans do not create a connection to transmit information to each other. As mentioned earlier, spans cannot be concurrent for the transfer time to be computable.

3.3.3 How to calculate transfer time?

Let us note that in the current context, transfer time calculations can be done both to the spans of a particular trace, as well as for the groups of traces. In the latter case the transfer time refers to pairs of operations. An algorithm is employed to calculate all transfer times within a group of non-concurrent spans. At the beginning of the algorithm, calculations are performed for the differences between the end times of the spans execution and the start times of other spans, regardless of whether they are concurrent or not. Subsequently, the algorithm filters out times that turn out to be negative, thereby partially eliminating cases where predecessors are spans that execute later. However, this alone is not sufficient to ensure that the times are genuine transfer times. It's possible that a concurrent span got delayed and started executing later than it should have, and in the algorithm, it would be treated as a sequential process. The next step involves filtering out such cases by checking whether the successor is not a descendant of the preceding span. As a result, only spans that genuinely represent transfer times are retained. The algorithm calculating transfer time can be summarized in the following pseudocode:

```
1. Select trace of given group
2. For each trace:
    2.1 for each pair(span1,span2)
    2.1.1 calculate time from the end of span1 to the begin of span2.
    2.1.2 if the time is less then zero continue next iteration,
        else add to potential transfer.
3.for each pair(potential_span1,potential_span2)
    3.1 if percendance was in less then 20\% of all traces
        ignore this pair.
    3.2 if potential_span1 is parent of potential_span2
        then ignore this pair, else add pair to transfer time.
```

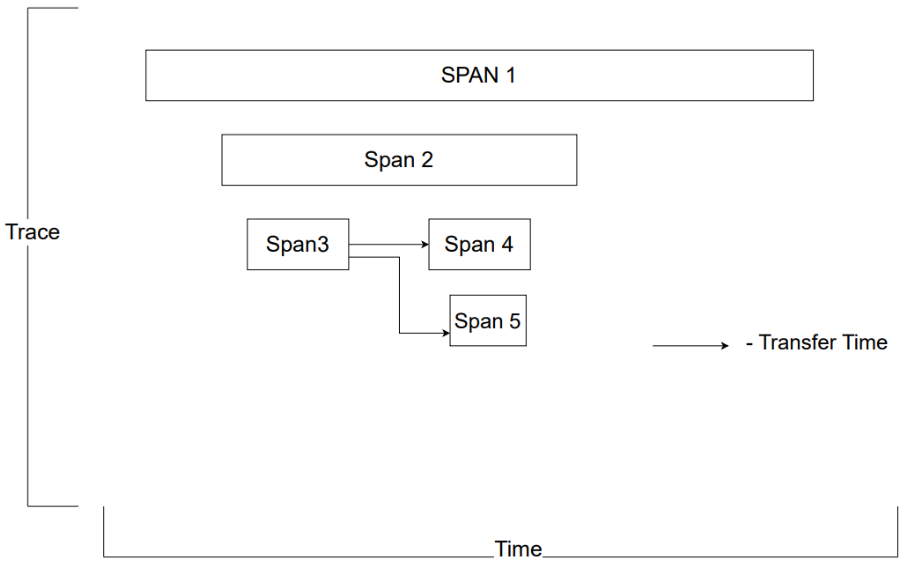


FIGURE 3.4: Visualization of transfer time

Chapter 4

Project

In this chapter, we present the design of Jaeger trace processing and data visualizing application. System requirements, architecture, GUI and testing will be outlined.

4.1 Functional and Non-functional requirements

4.1.1 Functional requirements

The initial step to begin the project involves performing tests on Intel infrastructure using the Death Star Benchmark. These tests will evaluate the system's performance under heavy loads and generate runtime logs also known as "traces." The data collected from these logs will be essential for analyzing and improving our application's performance.

Developing a Module to Visualize Microservices Invocation Chains As part of further analysis, we plan to develop a module to visualize invocation chains of microservices based on the collected logs (traces). This module will serve several key functions:

- The module will present a directed graph depicting relationships between microservices, particularly identifying the most common invocation chains.
- It will display distributions of execution times in individual microservices, enabling a quick understanding of the efficiency of specific system components.
- It will provide statistics on communication times between microservices, helping to identify potential delays in data transmission.
- The final visualization will be accessible in a web browser, providing easy access and interactivity for users.
- In cases where the network of communication between services does not form a chain, the module will identify critical paths, allowing the identification of potential performance issues.

The planned visualization module is a critical element of our application's performance analysis, enabling a quick understanding of interactions between microservices and identifying areas requiring optimization.

4.1.2 Non-functional requirements

The fallowing non-functional requirements for the system were defined:

- Python will be the primary programming language utilized for trace analysis and statistical processing tasks in the project. Its versatility and extensive libraries make it well-suited for these purposes.
- For web-based visualizations, any JavaScript library can be used with a preference for a well-established one that ensures compatibility and flexibility in various web environments.
- The software's design is intentionally general-purpose to avoid being exclusively tied to the Death Star Benchmark (DSB). It aims to be adaptable to a wide range of scenarios involving trace analysis and statistical processing.
- The project promotes the use of stable and high-quality open-source software to encourage transparency and collaboration in the development process.
- The thesis will be written in English to ensure clarity and accessibility for a global audience.
- Lastly, the intention is to release the developed software as open source aligning with the principles of collaborative development and allowing for broader community access contribution and benefit from the project.
- The system works offline, processing a set of logs and visualizing data without the need for internet connectivity.

4.2 System architecture

4.2.1 Context diagram

In the context diagram, ee figure 4.1 there are four entities interacting with the system. The first entity is the user, engineers who wants to test their application infrastructure. These users are a crucial element in the context diagram as they directly influence what is processed in our application. The next element in the project's context is the Death Star Benchmark, as mentioned earlier, it is responsible for stress-testing the application. Through it, one can observe certain inconsistencies in the tested application (something working slower, experiencing freezes). Jaeger monitors and collects information on how the system behaves under the load and records it as JSON file, subsequently processed within the application.

Another element in the context is the cloud infrastructure. What is cloud infrastructure? Cloud infrastructure, in the context of computing services, allows access to computer resources such as servers, storage, networks, and software over the internet. In contrast to traditional infrastructure models where resources are locally installed and managed, cloud infrastructure offers flexibility and scalability by providing these resources on demand. The tested application operates within the cloud infrastructure, and its performance is heavily dependent on it.

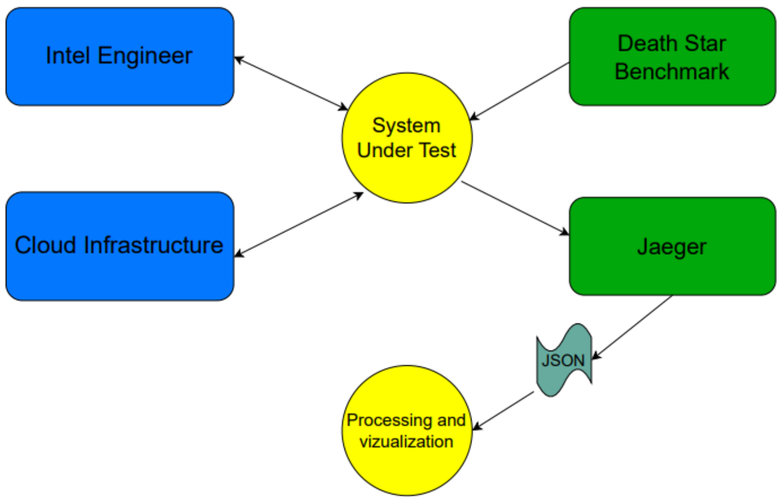


FIGURE 4.1: Diagram Context

4.2.2 Use cases

A diagram illustrating use cases for the system is presented in Figure 4.2. The main use cases are outlined as follows:

- Import data
  The user begins by interacting with the system, uploading the JSON file containing the data. Subsequently, the system validates the uploaded file and calculates the statistical values. Once the processing is complete, the user can analyze the data by exploring five types of visualizations: a table with statistical values, a call graph, a scatter plot, a histogram, or a precedence graph.

- View table
  The table with statistical values can be seen for operations within the groups and for traces.

- View call graph
  The call graph can be viewed for traces and for groups of traces.

- View scatter plot
  The scatter plot can be opened for groups of traces, for operations within the groups, and for spans within the operations.

- View histogram
  The histogram can be seen for groups of traces, for operations within the groups, for spans within the operations, for spans within the groups, and for spans within the file.

- View precedence graph
  The precedence graph can be viewed for groups of traces, and it will show a graph of transfer times if such times could be calculated for the group.

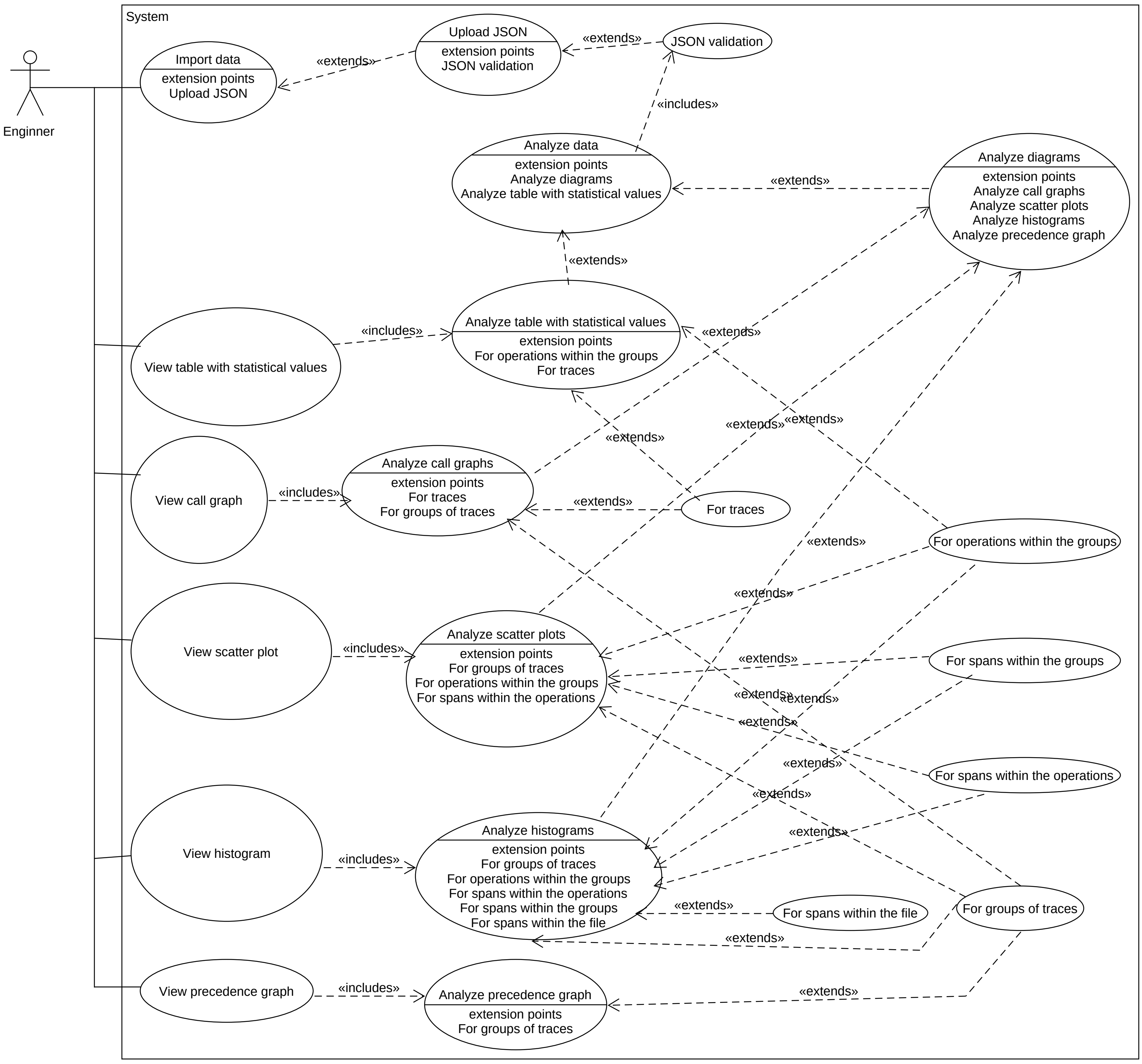


Figure 4.2: Use case diagram

4.3 System architecture

System architecture is shown in Figure 4.3. The system contains three main components react, flask, and docker platform. In the frontend layer, React was employed for:

- graph visualization in the user interface
- calculations, and enabling users to upload files.

In the backend, Python with the Flask framework was mainly used for:

- processing log files,
- handling endpoints,
- performing a significant portion of computations.

Docker platform used for:

- Environment isolation is offered which resolves problems that arise from variations in operating system configurations.
- Facilitates the movement of applications across different environments, accelerating development and deployment.
- Docker containers can be launched very quickly, speeding up the application deployment process.

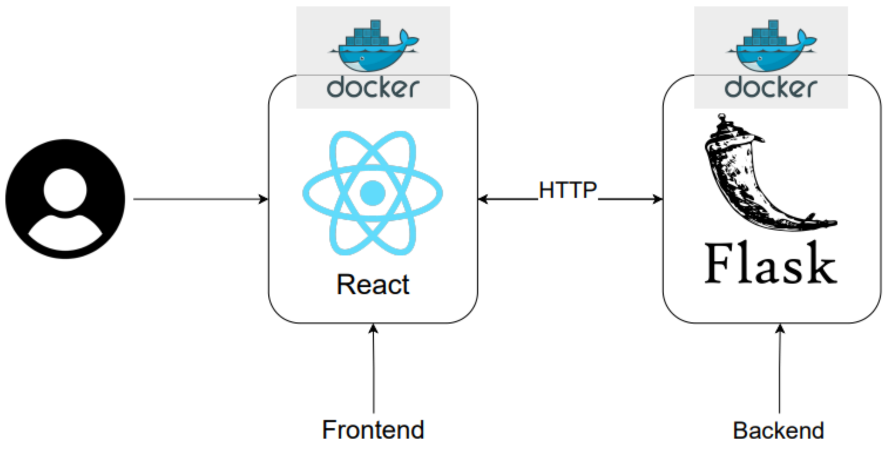


FIGURE 4.3: System Architecture

4.4 Interface mockup

Am interface mockup is presenting a schematic organization of the application graphical user interface. It defines defines interface blocks allowing to access system functionalities defined as use-cases. The interface mockup is crucial as it needs to be tailored to the technology being utilized. Therefore, our mockup is customized with fixed elements, such as the tab selection, and underneath is the space where charts will be displayed. This is enabled by the React framework, allowing us to adjust components on an already functioning page without completely changing it—just by launching a new component. This approach is more optimal as certain elements on the page remain unchanged. The mockup has been designed in accordance with the requirements of both the supervisor and the client to be as intuitive in navigation as possible. The goal is to make it a tool that is easy to learn.

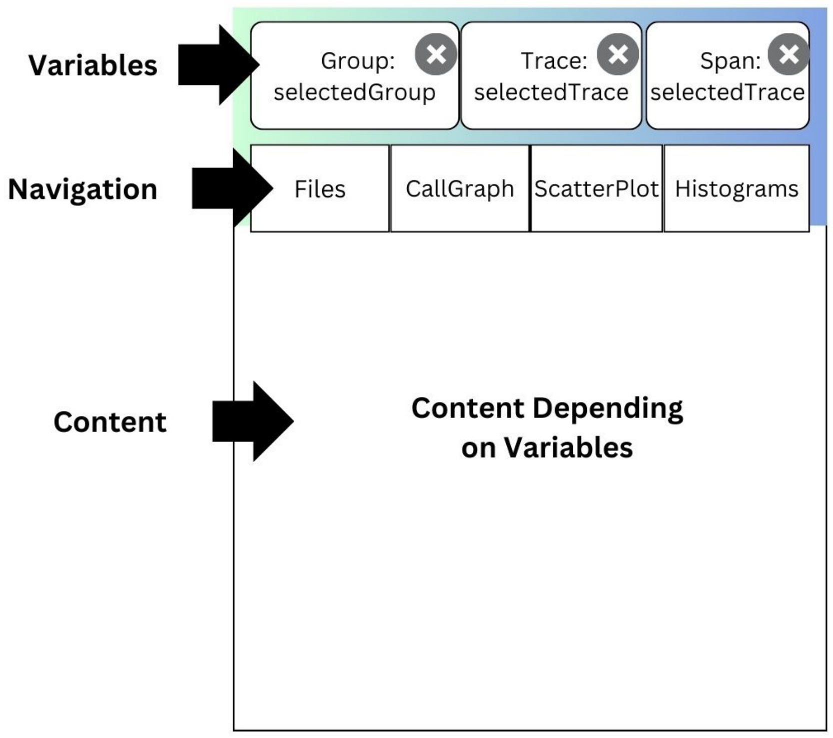


FIGURE 4.4: Interface Mockup

The diagram illustrates three zones. The first one, ’Variables,’ shows the variables selected by the user in the application, such as the chosen group or trace, for example. The next zone in the interface is ’Navigation,’ where users can select the visualization the system should display. The ’Content’ zone displays all components, including visualizations of graphs, histograms, and tables.

4.5 Testing

When developing software it’s necessary to test it. In the case of this project, the aspect of testing wasn’t so obvious - its main feature is visualization and statistical calculation based on an input file generated by real distributed system. To create such a file, one approach could be to generate traces using the DSB benchmark, but it’s execution and output values are something we can’t control, so it can be considered completely random. Therefore, we need to generate files with set values like duration times, communication times, etc. As a result, we will develop a Python script to generate artificial traces. The script will work in a parameterized way - the user can specify the number of traces, maximum number of spans per trace, specify duration or transfer time to a fixed value (random otherwise) and input the number of microservices. These generated JSON files will be used to validate calculations and visualizations generated by our software. Because of

the specifics of our project, it doesn't have the regular unit tests - it will be necessary to validate the software "by hand".
Below is a list containing features planned to test:

- Call graphs - we will generate traces with specified number of spans. We will check, if the number of spans in the graph is aligned with the set value. For smaller, linear instances we will also validate the order of spans on said graph. It's also necessary to check if given operations are assigned to their services correctly.
- Scatter plots - our goal is to generate traces, where part of the spans will have set duration time, while randomly selected few will have lower or higher values by specified percentage. This way we will validate if values on scatter plots align with expectations.
- Tables with statistical data - using the script we will generate spans with set duration times. We should see the same averages, means and percentiles.
- Transfer times - the test will consist of checking if calculated values align with the transfer times set in the script. For instance, if the result of the script will contain services for which transfer time was set to 200ms, we should see the same number in the application.

Aside from these, we aim to evaluate performance of the application by measuring average response time when rendering visualisations in response to file upload.

Chapter 5

Implementation and tests

In this chapter, we present details of system implementation, testing, and discovered data artifacts, as well as the way to use the project.

5.1 Used technologies

5.1.1 Docker

Docker [14] is an open-source platform designed to deploy, scale, and manage applications through containerization. This approach involves packaging an application along with its dependencies (i.e., required software) into a standardized image, simplifying software deployment. Docker generates containers, which are self-contained environments that encompass everything necessary to execute a program, ranging from operating system packages to source code. These containers are lightweight, ensuring portability, and include all essentials for running the application, eliminating dependence on host platform installations. The ability to share containers facilitates collaborative work, ensuring consistency in functionality across users. Our project adopts Docker to simplify environment installation on diverse hardware. We opted for this technology due to its well-documented nature and the professional experience team members possess with it. Our project utilizes two containers: one for React handling the frontend, and another for Flask managing the backend.

5.1.2 GitHub

GitHub [12] is an online source-code hosting service designed for version control and collaborative software development. It facilitates seamless collaboration among developers, enabling them to work on projects from any location. Utilizing Git, a distributed version control system, GitHub tracks changes in source code throughout the software development process. We selected GitHub to guarantee that all team members can access various code versions and contribute changes visible to others. This technology also enables us to revert to previous working code versions in case of emerging software issues. Additionally, we plan to publish our project as an open-source repository on GitHub.

5.1.3 Python

Python [21] is a high-level, interpreted programming language renowned for its simplicity, readability, and sound design principles. Supporting various programming paradigms, including procedural, object-oriented, and functional programming, Python stands out as an ideal language for beginners. In our system, Python serves as the backend programming language, enabling us to write code that manages the environment, processes data, and communicates with the frontend seamlessly. Moreover, Python provides a wide range of libraries and tools for data processing. Thanks to it, the development of data processing components in our system was simplified.

5.1.4 Flask

Flask [20] is a lightweight web application programming framework written in Python. We chose Flask for our project, recognizing its features as well-aligned with our needs, and employed it as a tool to construct the backend environment. Furthermore, using Flask allowed seamless connection of the web interface with data processing components also written in Python.

5.1.5 JavaScript

JavaScript [8] is a versatile, cross-platform, object-oriented scripting language primarily employed to enhance the interactivity of web pages. It facilitates the creation of sophisticated animations, interactive buttons, pop-up menus, and various other web elements. Thus, JavaScript was primarily executed in a browser environment. Additionally, JavaScript functionality can be extended to work on the server side, as exemplified by technologies like Node.js. Given our team's lack of prior experience in frontend development, we opted for JavaScript due to its extensive learning resources and widespread use in the field. What is more, JavaScript is a de facto standard for client-side (i.e., in-browser) applications.

5.1.6 React.js

React.js [16] is a JavaScript library developed by Facebook and opened to the public to simplify the creation of interactive user interfaces. Developers can build applications by assembling reusable components responsible for generating concise, reusable fragments of HTML code. We employed React.js as the primary technology for constructing the graphical user interface in our system. The choice was driven by the extensive community support and the abundance of helpful packages that significantly expedited the application development process. The ability to build independent reusable components provided our team members with the opportunity to develop these components separately before integrating them into the complete application.

5.1.7 React-Vis

React-Vis [22] is a data visualization library built on React by Uber Technologies. Offering support for various plots, charts, and other visual representations of data, React-Vis facilitates the seamless integration of data visualization into React applications. In our system, we leverage this library to generate scatter plots. The decision to use React-Vis was influenced by our observation of its well-crafted scatter plots in other contexts and by the stability of this library.

5.1.8 D3.js

D3.js [18] is an abbreviation for Data-Driven Documents. It is a JavaScript package employed to craft data visualizations within web browsers. It makes use of standard web technologies such as HTML, SVG, and CSS. In our project, we used the D3.js package to generate histograms. The inherent flexibility of its low-level nature allowed us to finely tailor the histograms to fulfill the intricate demands of data visualization, like determining a custom number of bins and using a logarithmic scale.

5.2 Installation

Prerequisites

Follow the steps in these instructions (https://docs.docker.com/desktop/install/windows-install/) to install Docker and Docker Compose.

Get project files

If you already have the zip file containing the project files, unzip it, and you can skip to the 'Run the app' subsection in 5.2.

Download repository

In the chosen directory, use the command below to clone the git repository.

```
git clone https://github.com/Kxpi/performance-assessment-and-modelling-of-
microservices-based-cloud-systems.git
```

Update repository

Use the command below to update the git repository.

```
git pull
```

Switch the branch to init_webapp

In the repository's directory, use the command below to change to the 'init_webapp' branch.

```
git checkout init_webapp
```

Run the app

Start the Docker Engine by running the Docker Desktop application. If you've made changes or are running the app for the first time, you'll need to rebuild the images. Use the following command in the project directory to do it:

```
docker compose build --no-cache
```

Run the following command in the project directory to start up:

```
docker compose up
```

Open

Open this link in your web browser.
http://localhost:3000/

Exit the program

To stop the program, send the SIGINT (Signal Interrupt) command, which is [ctrl+c], to the process running 'docker compose up'.

5.3 Usage and logic

5.3.1 Header

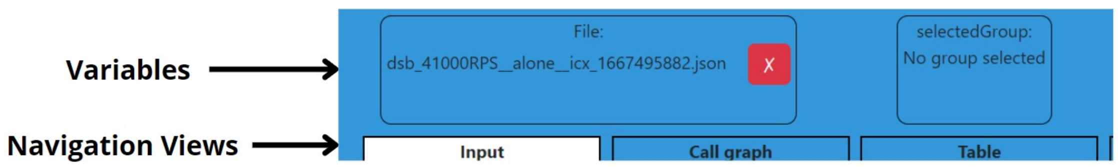


FIGURE 5.1: Header

The Header component(see Fig. 5.1) functions as the central command hub, it is divided into two sections: Navigation Section and Variable Section.
Navigation Section enables users to switch between various views. Each tab within this section represents a unique view, such as ”Input” or “Scatter Plot”.
The variable section manages the state of data item variables enabling switching between views without having to set the data item variable again. This is crucial as the content of views depends on the set data item variable values. Additionally each tile in variable section is equipped with an ’x’ button for resetting its value.

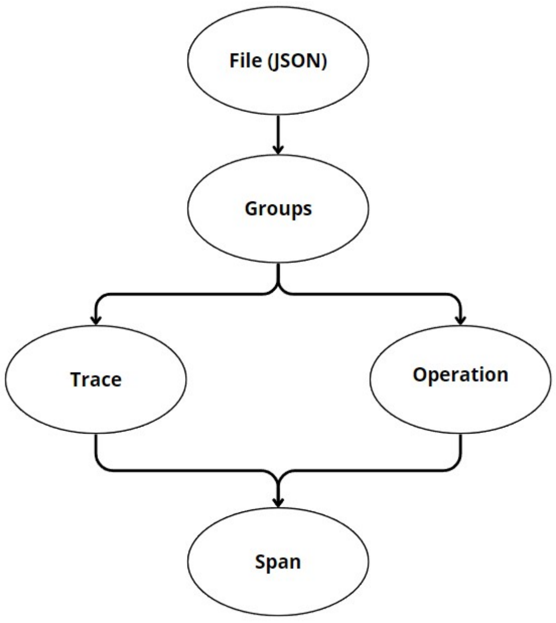


FIGURE 5.2: Data item variables hierarchy

It is important to note that when when modifying or clearing data item variables, that they are organized in a hierarchical structure(cf. Fig.5.2). The meaning of a group, trace, span was explained in sections 2.3 and 3.2. Clearing or modifying one variable will also affect the variables below it in the hierarchy, because data items are not only identified by their own values, but also exist in the data sets higher in the hierarchy. For instance, if a group and a trace are selected, and then it is decided to change the group, then the trace will no longer be associated with the selected group value. This hierarchical structure ensures consistency in managing data item relationships.

5.3.2 Input

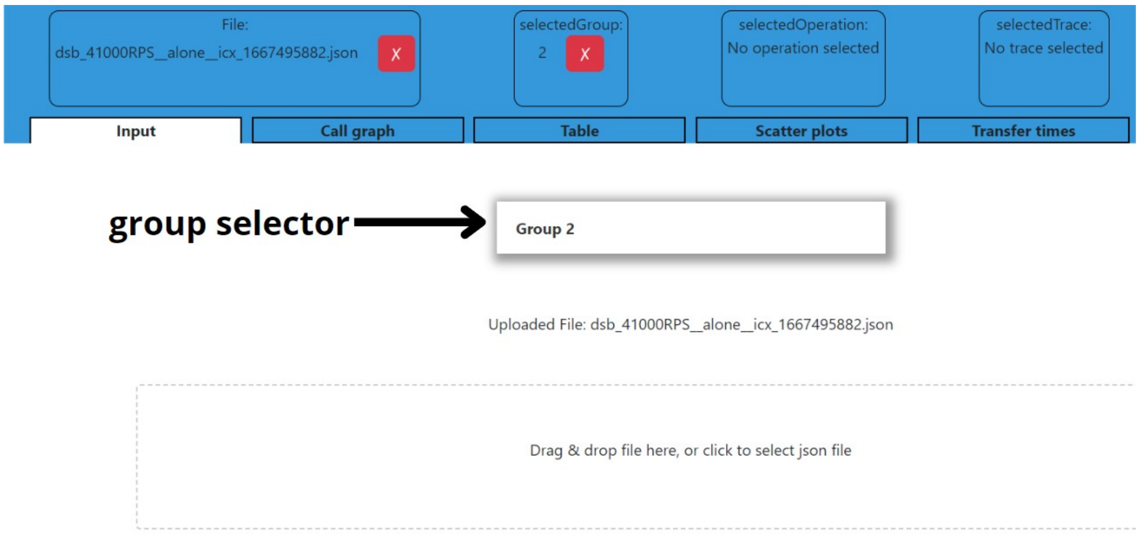


FIGURE 5.3: Input Tab

The Input tab(Fig.5.3) in the application acts as the main interface where users can upload files and choose a group. This page is crucial as it sets the data items visualization in all other parts of the application. Hence views in the application rely on the Input page. It is necessary to upload a JSON file and in some cases select a group. The functionality of the callGraph view, for example, relies on the selected group.

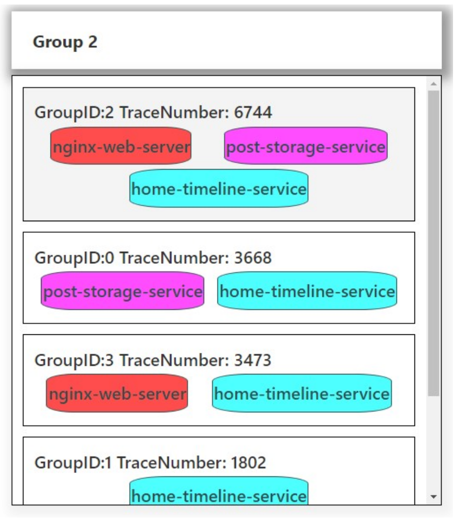


FIGURE 5.4: Group Selector

Group Selector(Fig.5.4) is a feature that enables users to choose a particular group in the input file. By clicking on the component, a drop-down menu will appear, displaying all the available groups. Each group in the drop-down is presented with a group ID, a trace number, and colorful badges indicating the service names. There may also be a special group displayed which includes all traces with negative start times(see 5.3.1).

FIGURE 5.5: The legend of the scatter plot of groups

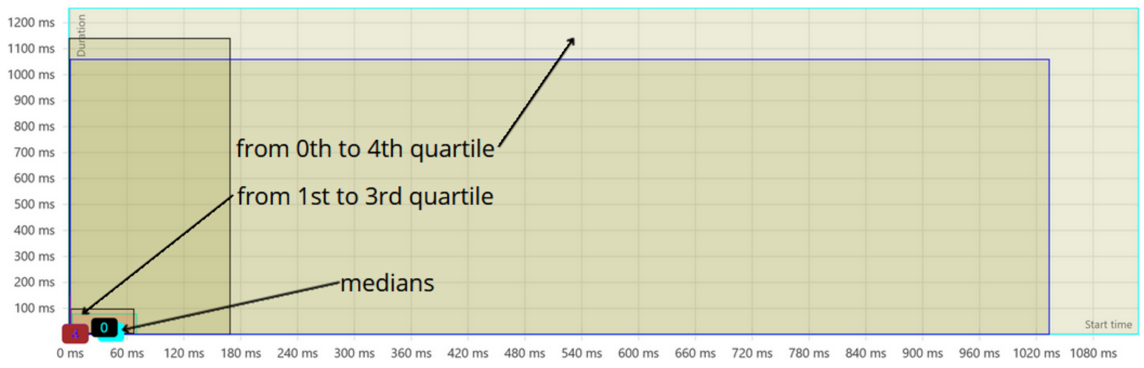


FIGURE 5.6: Scatter plot of groups

5.3.3 Scatter plots

In the scatter plots tab, the start times and durations of groups, operations, and spans are displayed. The y-axis represents duration, while the x-axis indicates the start time. The plots use milliseconds as the unit of measurement. Users can zoom in on the plot by holding down the Shift key and the left mouse button while selecting the desired area. To return to the default view, users can hold down the Shift key and click the left or right mouse button.

Scatter plot of groups

This component, shown in Fig. 5.6, displays data items representing groups in the JSON file as boxes. Each group is visually distinguished by a unique color and corresponding number in the box, as indicated in the legend presented in Fig. 5.5. The points in group boxes (shown as numbers) depict the median durations and median start times of spans within each group. Additionally, each group includes two rectangles with frames matching their respective colors. The smaller rectangle, filled with light brown, represents values from the first quartile to the third quartile of both duration and start time within the group. The larger rectangle, filled with light green, signifies values from the zeroth quartile to the fourth quartile of both duration and start time within the group. Users can select a group by clicking on the object with the group number, leading them to the scatter plot of group operations view. The chosen data item variable is shown in the page header variable section (see section 5.3.1, Fig. 5.1).

Scatter plot of group operations

The scatter plot of operations repeats the visualization scheme of groups in files. This component, presented in Fig. 5.8, shows the operations within the group. Each operation is distinguished by a unique color and the accompanying number (in the box) indicating the group number, as illustrated in the legend shown in Fig. 5.7. The numbers in a box bearing the group number and operation's color represent the median durations and median start times of spans within the operation in the

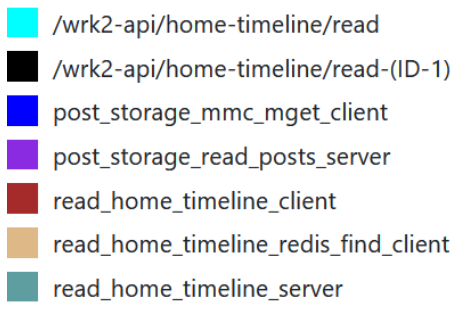


FIGURE 5.7: The legend of the scatter plot of group operations

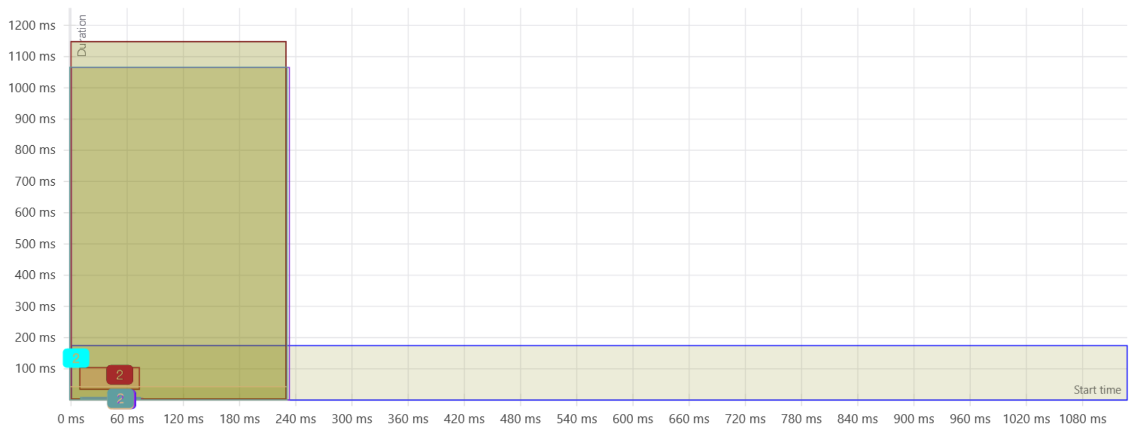


FIGURE 5.8: Scatter plot of group operations

group. Each operation is shown as two rectangles with frames matching the operation's color. The smaller rectangle, filled with a light brown color, signifies values from the first quartile to the third quartile of both duration and start time of spans within the operation in the group. The larger rectangle, filled with light green, denotes values from the zeroth quartile to the fourth quartile of both duration and start time of the spans within the operation in the group. Users can choose an operation by clicking on the object with the group number and operation's color, prompting the appearance of the scatter plot of operation spans below. The chosen data item in the hierarchy is indicated in the variable section at the top of the page (see section 5.3.1, Fig. 5.1).

Scatter plot of operation spans

This component, shown in Fig. 5.10, displays the objects representing spans within the group operation. Due to the large number of spans and potential browser performance issues, users have the option to choose the percentage of spans to display, as illustrated in Fig. 5.9. The objects indicate the values of both duration and start time for the spans. Users can choose a span by clicking on the corresponding number, after which the details of the selected span will be displayed below the scatter plot. Example results are shown in Fig. 5.11.

5.3.4 Histograms

The Histograms tab serves the purpose of displaying histograms of frequencies of duration and start time in groups, operations, and particular spans. For determining the number of bins in histograms with a logarithmic scale, we applied the following formula:

$$\text{Number of bins} = \lceil \log_2(\text{Number of spans analyzed in the histogram}) + 1 \rceil$$

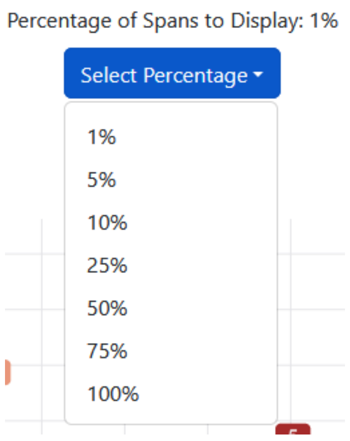


FIGURE 5.9: The selection of the percentage of spans to display

Duration histogram of groups

This component is shown in Figure 5.12. In the group histogram, the y-axis represents the 99th percentile of the duration for spans within each group. Group IDs are depicted on the x-axis. Bars, corresponding to the groups, are arranged in descending order based on their values on the y-axis. Users can choose a group by clicking on its respective bar. The selected group is indicated in the header variable part of the page (see section 5.3.1, Fig. 5.1).

Start time histogram of groups

This component is presented in Figure 5.13. In the group start time histogram, the y-axis displays the 99th percentile of start time values for spans within each group. Group IDs are represented on the x-axis. The bars, signifying the groups, are arranged in descending order based on their values on the y-axis. Users can choose a group by clicking on its respective bar. The selected group is indicated in the header variable part of the page (see section 5.3.1, Fig. 5.1).

Duration histogram of spans

This component is shown in Figure 5.14. In the spans duration histogram, the y-axis illustrates the counts of spans in the analyzed file within specified bins on a logarithmic scale with a base of 10. The x-axis denotes the values of duration.

Start time histogram of spans

This component can be seen in Figure 5.15. In the spans start time histogram, the y-axis illustrates the counts of spans in the analyzed file within specified bins on a logarithmic scale with a base of 10. The x-axis denotes the values of the start time.

Duration histogram of group operations

The group histograms section is accessible from file histograms by choosing a group. The visualization scheme for operations in a group repeats the visualization scheme of groups in a JSON file described earlier. This component is presented in Figure 5.16. The y-axis displays the 99th percentiles of the duration for spans within the group operations. Operation names are depicted on the x-axis. Bars, representing the group operations, are arranged in descending order based on

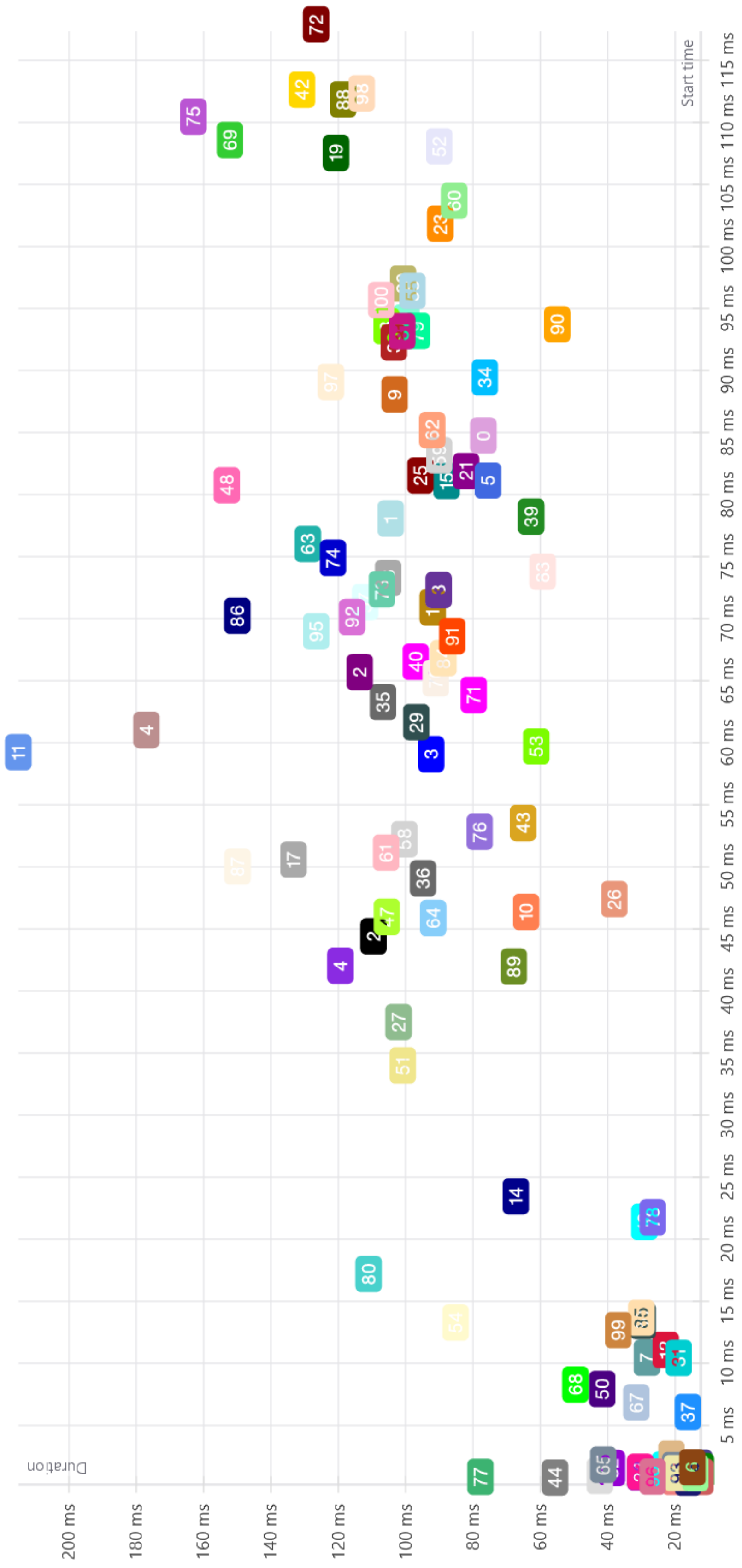
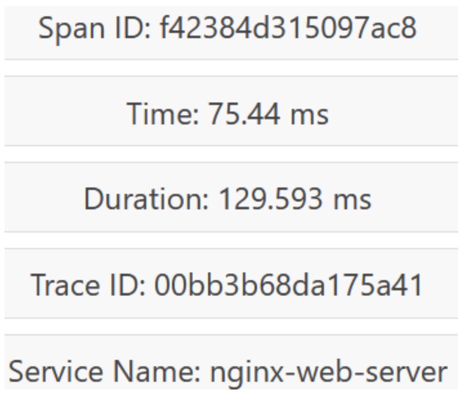


FIGURE 5.10: Scatter plot of operation spans



Span ID: f42384d315097ac8

Time: 75.44 ms

Duration: 129.593 ms

Trace ID: 00bb3b68da175a41

Service Name: nginx-web-server

FIGURE 5.11: Details of the selected span

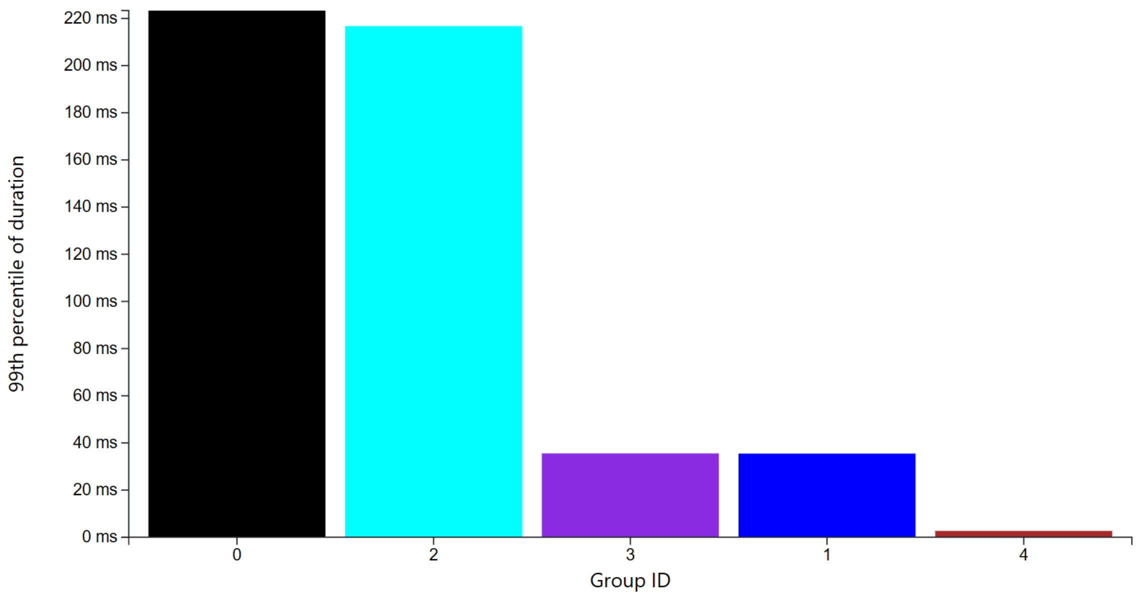


FIGURE 5.12: Duration histogram of groups

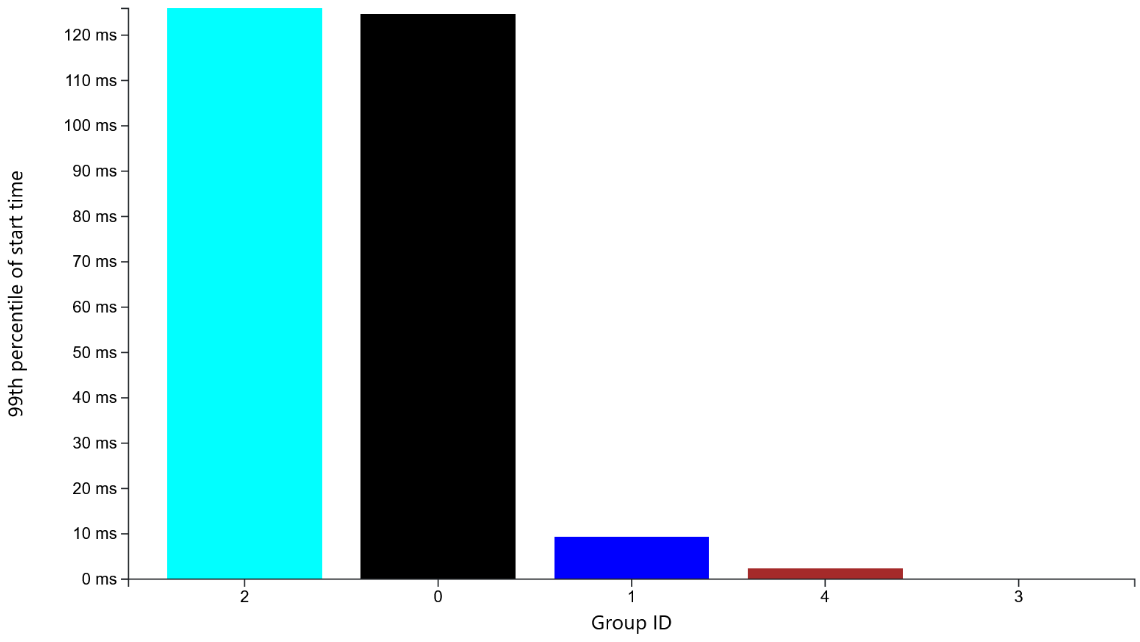


FIGURE 5.13: Start time histogram of groups

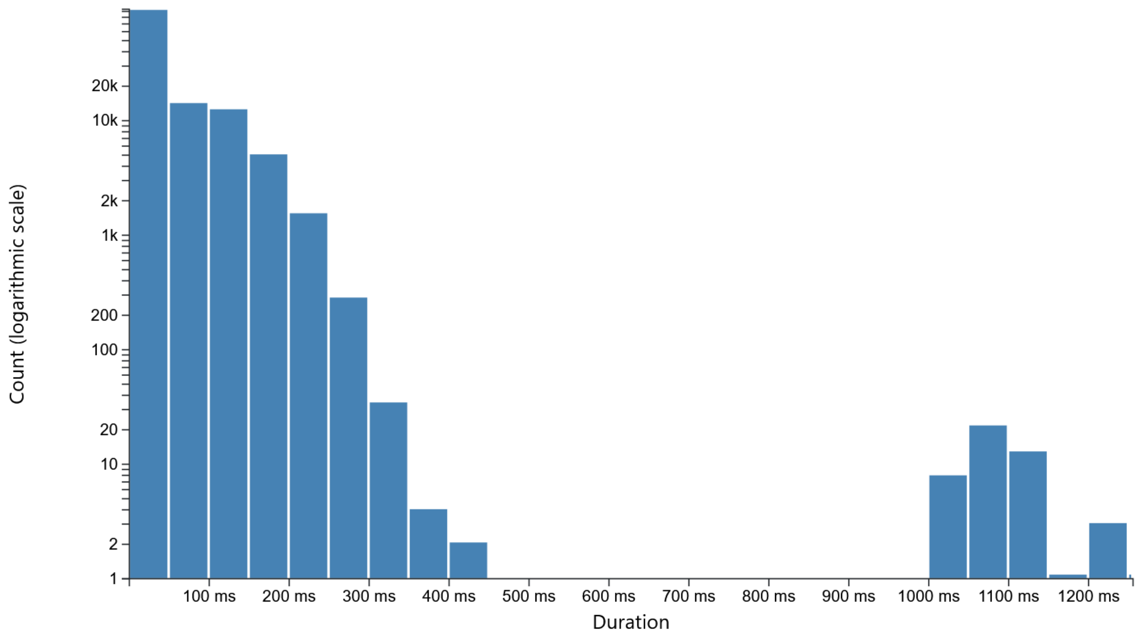


FIGURE 5.14: Duration histogram of spans

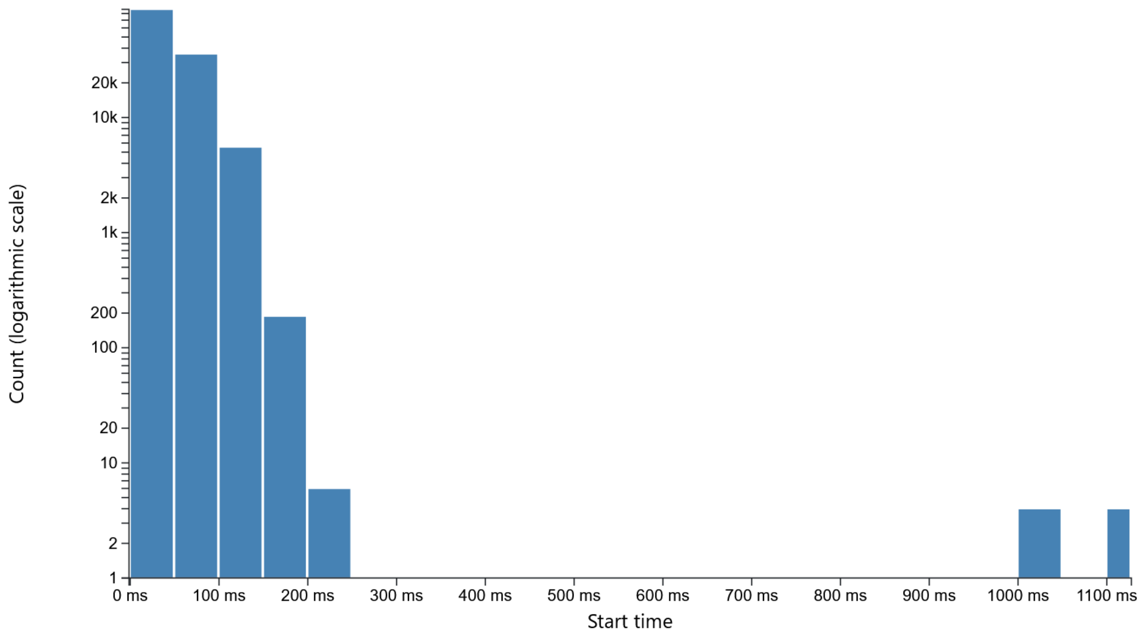


FIGURE 5.15: Start time histogram of spans

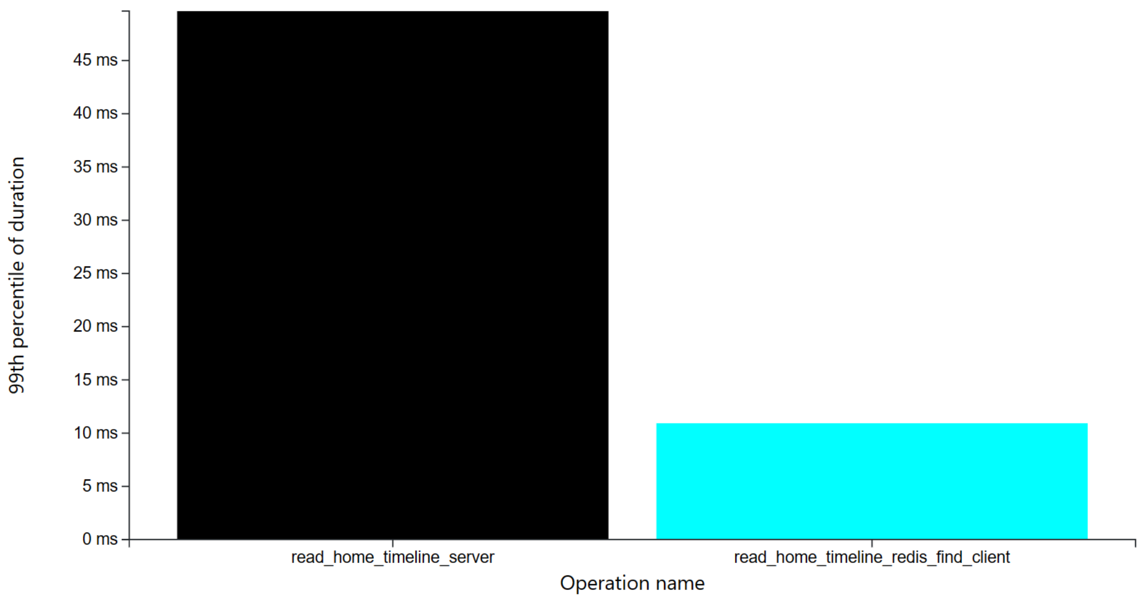


FIGURE 5.16: Duration histogram of group operations

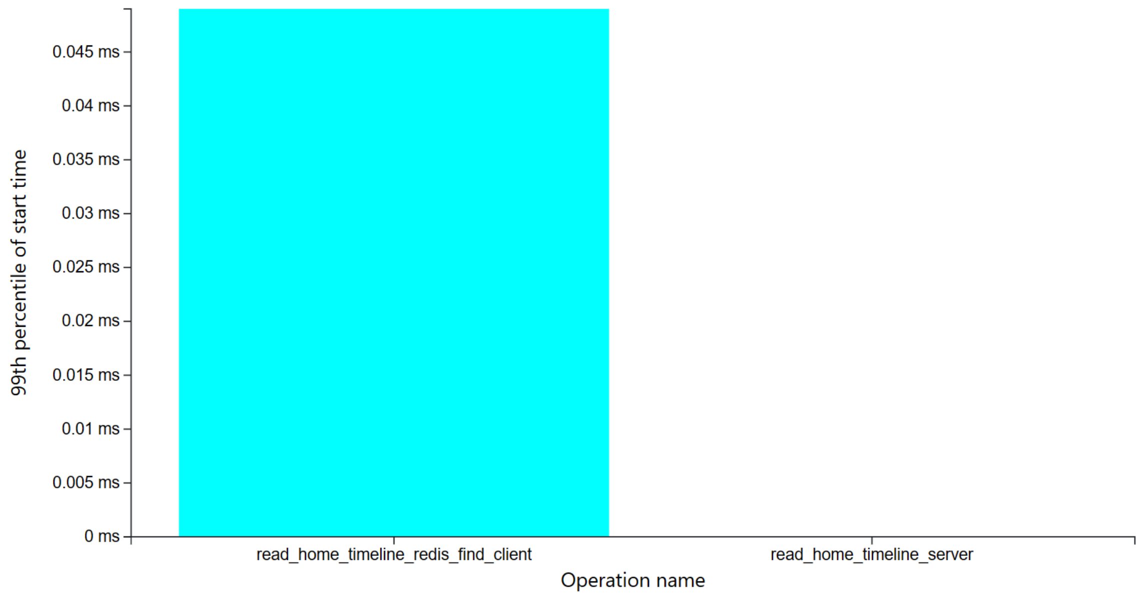


FIGURE 5.17: Start time histogram of group operations

their values on the y-axis. Users can choose an operation by clicking on its respective bar. The selected operation is indicated in the header variable part of the page (see section 5.3.1, Fig. 5.1).

Start time histogram of group operations

This component is shown in Figure 5.17. In the start time histogram of operations in a group, the y-axis represents the 99th percentiles of the start time for spans within the group operations. Operation names are displayed on the x-axis. Bars, indicative of the group operations, are organized in descending order based on their values on the y-axis. Users can choose an operation by clicking on its respective bar. The selected operation is indicated in the header variable part of the page (see section 5.3.1, Fig. 5.1).

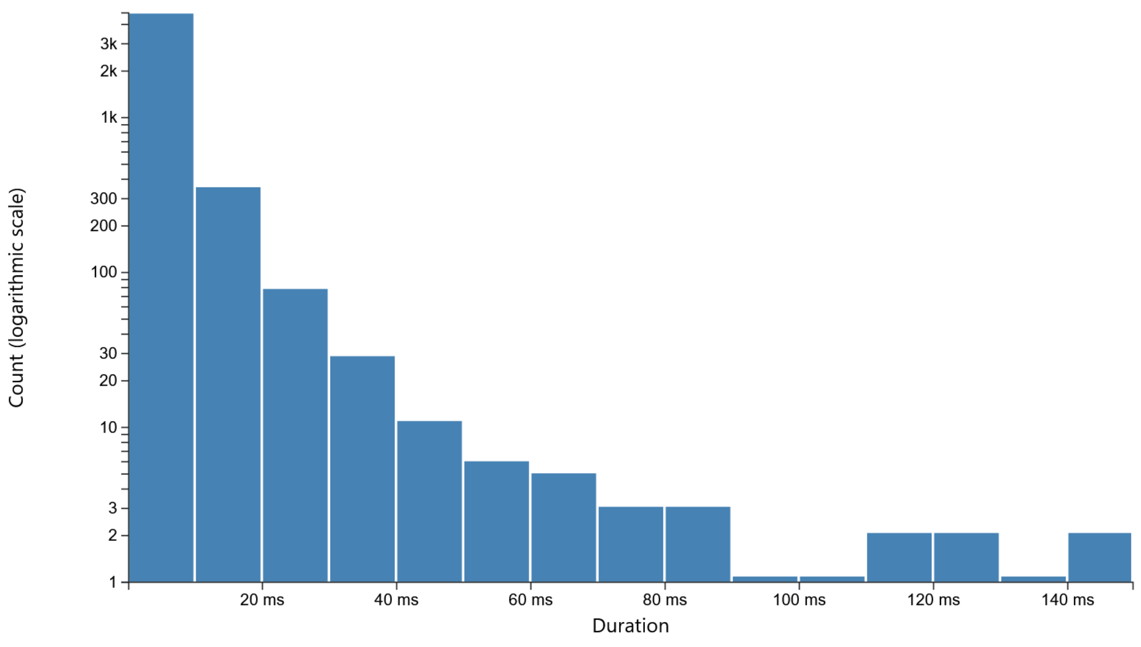


FIGURE 5.18: Duration histogram of group spans

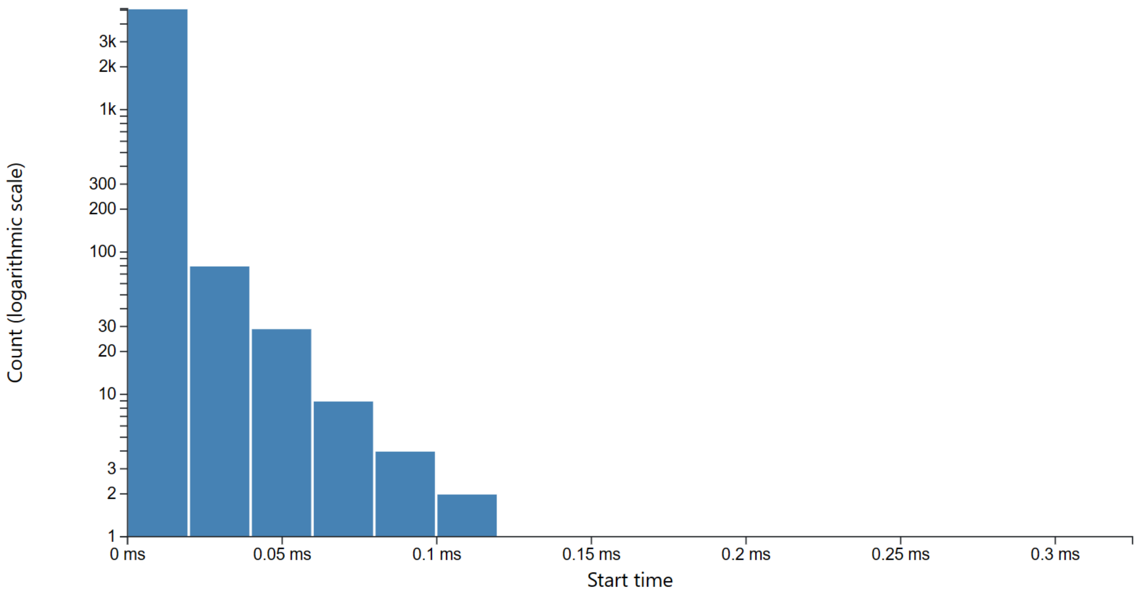


FIGURE 5.19: Start time histogram of group spans

Duration histogram of group spans

This component can be seen in Figure 5.18. In the duration histogram for spans of a group, the y-axis illustrates the counts of spans within the group, categorized within specified bins on a logarithmic scale with a base of 10. The x-axis denotes the values of the duration.

Start time histogram of group spans

This component is presented in Figure 5.19. In the start time histogram of group spans, the y-axis depicts the counts of spans within the group, distributed within specified bins on a logarithmic scale with a base of 10. The x-axis represents the values of the start time.

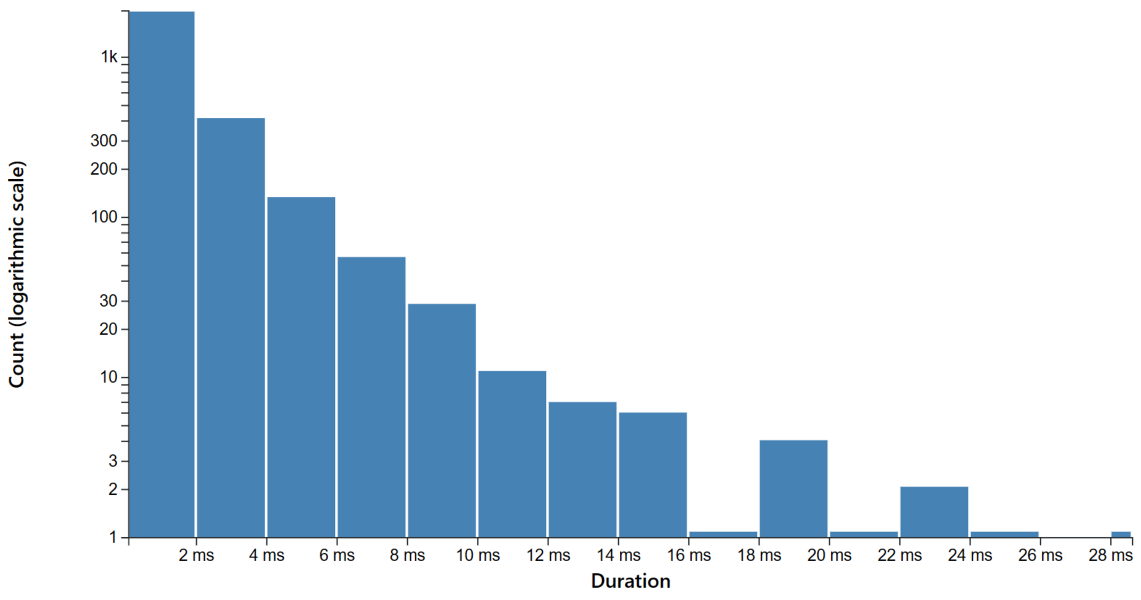


FIGURE 5.20: Duration histogram of operation spans

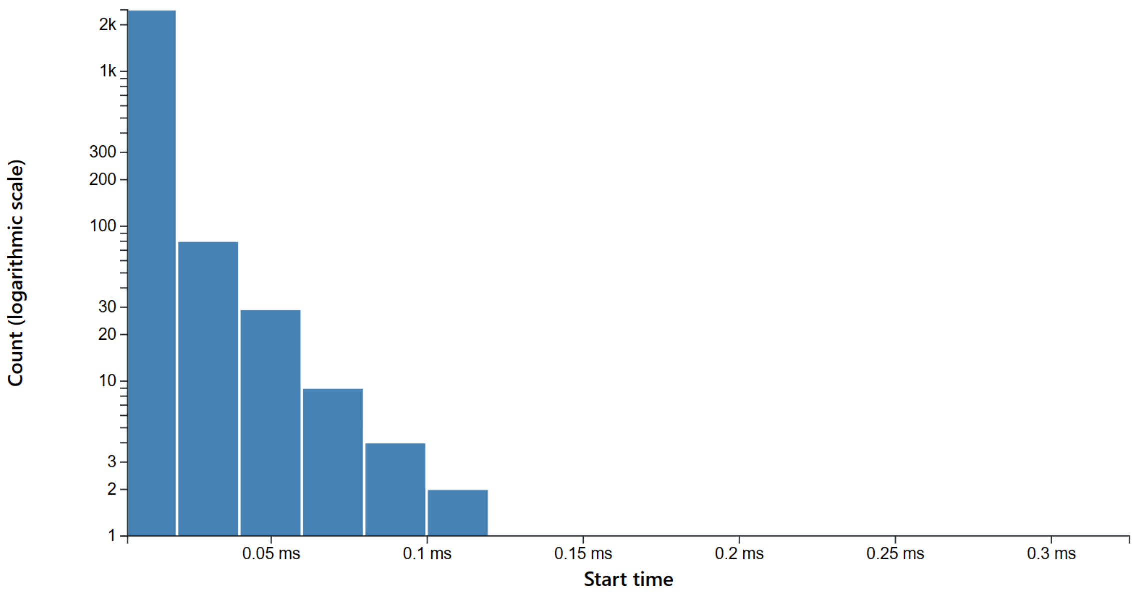


FIGURE 5.21: Start time histogram of operation spans

Duration histogram of operation spans

This component is shown in Figure 5.20. For the duration histogram of operation spans, the y-axis illustrates the counts of spans within the operation within the group, categorized within specified bins on a logarithmic scale with a base of 10. The x-axis denotes the values of the duration.

Start time histogram of operation spans

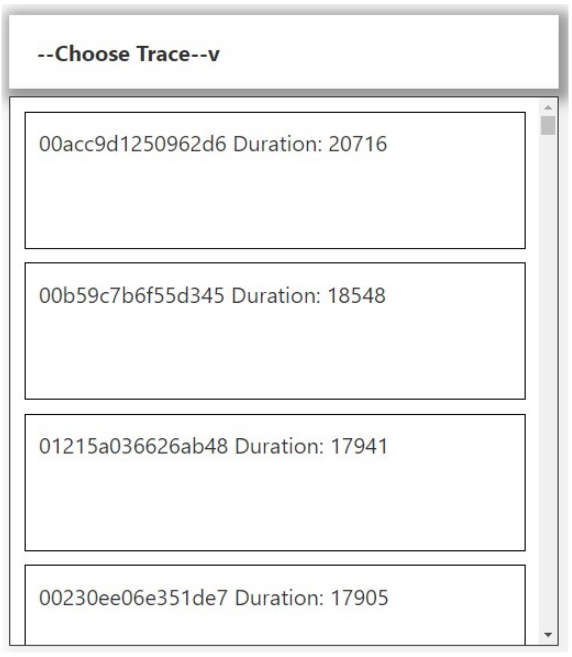
This component can be seen in Figure 5.21. In the start time histogram of operation spans, the y-axis depicts the counts of spans within the operation within the group, distributed within specified bins on a logarithmic scale with a base of 10. The x-axis represents the values of the start time.

5.3.5 Call Graph

The CallGraph visualizes the call tree of spans in the trace. Spans are represented by nodes, with node color indicating the microservice they belong to. The legend displays the services observed in the trace and their corresponding colors. Edges on the graph show the parent-child relationships between spans, which are specified in the span's references as CHILD_OF reference in the json file. On the call graph, the parent is the source of the edge, and the child is the target. There are two types of callGraphs:
one for individual trace - **TraceCallGraph** and one for groups - **GroupCallGraph**.

TraceCallGraph

To view the TraceCallGraph, first select a group and then choose one of the traces within that group. Click on a node to select a span and access more details.



--Choose Trace--v

00acc9d1250962d6 Duration: 20716

00b59c7b6f55d345 Duration: 18548

01215a036626ab48 Duration: 17941

00230ee06e351de7 Duration: 17905

FIGURE 5.22: Trace selection from the CallGraph page

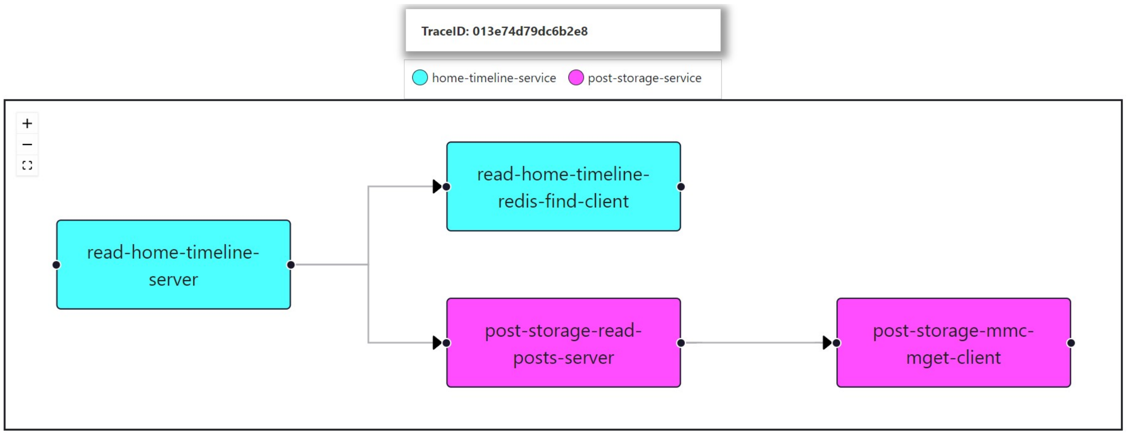
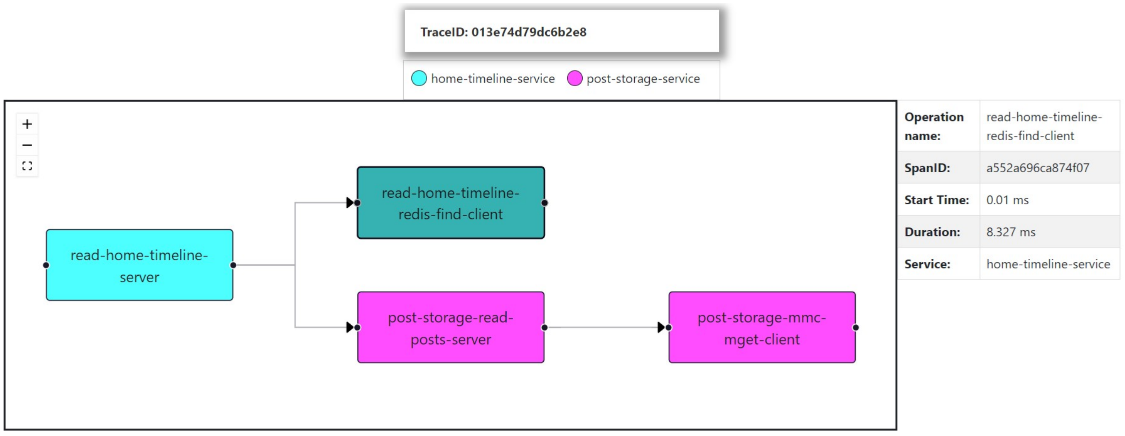

TraceID: 013e74d79dc6b2e8
home-timeline-service
post-storage-service
read-home-timeline-server
read-home-timeline-redis-find-client
post-storage-read-posts-server
post-storage-mmc-mget-client

FIGURE 5.23: Spans on TraceCallGraph


TraceID: 013e74d79dc6b2e8
home-timeline-service
post-storage-service
read-home-timeline-server
read-home-timeline-redis-find-client
post-storage-read-posts-server
post-storage-mmc-mget-client

Operation name:	read-home-timeline-redis-find-client
SpanID:	a552a696ca874f07
Start Time:	0.01 ms
Duration:	8.327 ms
Service:	home-timeline-service

FIGURE 5.24: Span selection from the TraceCallGraph

GroupCallGraph

This type of callGraph represents the entire group of traces that share the same call tree, resulting in the same appearance for all traces. Single node represents aggregated spans from the entire group with the same operationName. For simplicity we refer to it as an **operation**. By clicking on a node, you can select operation and view its statistics.

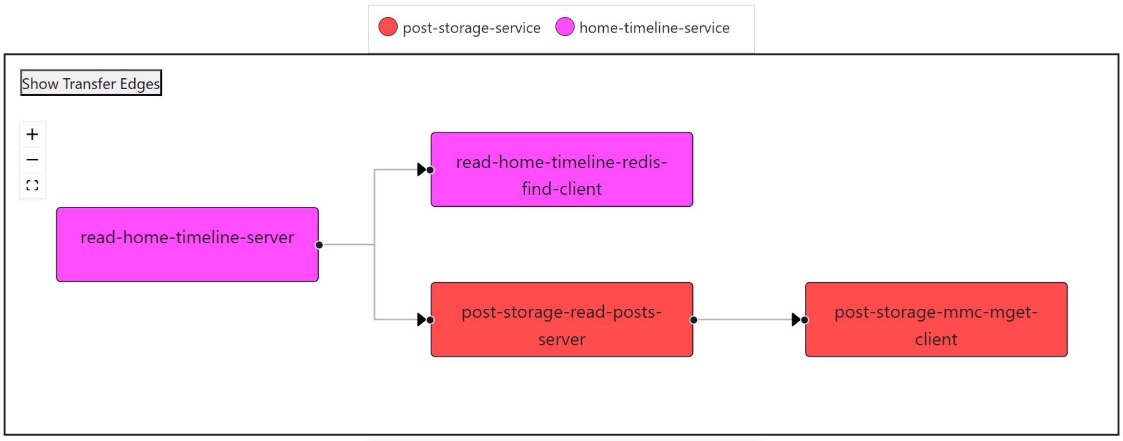
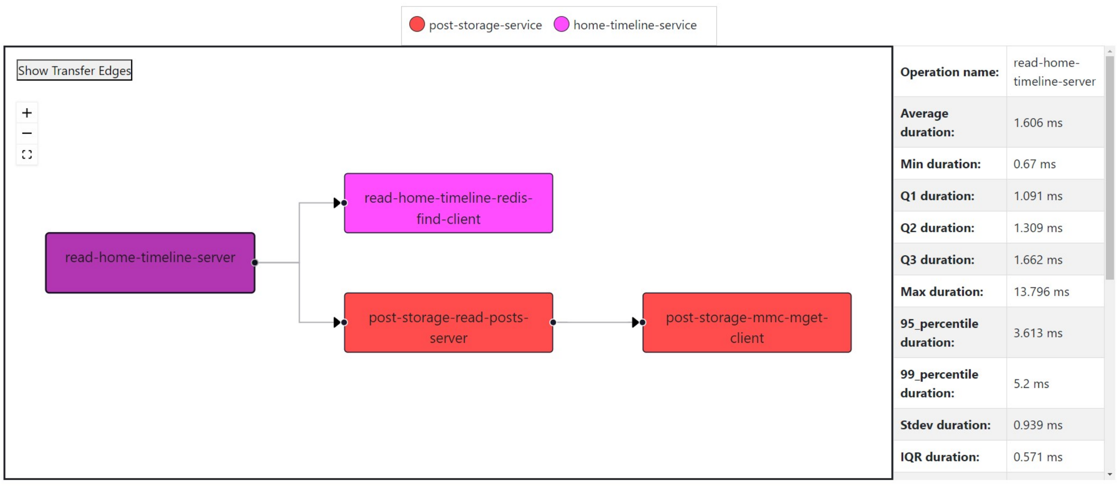
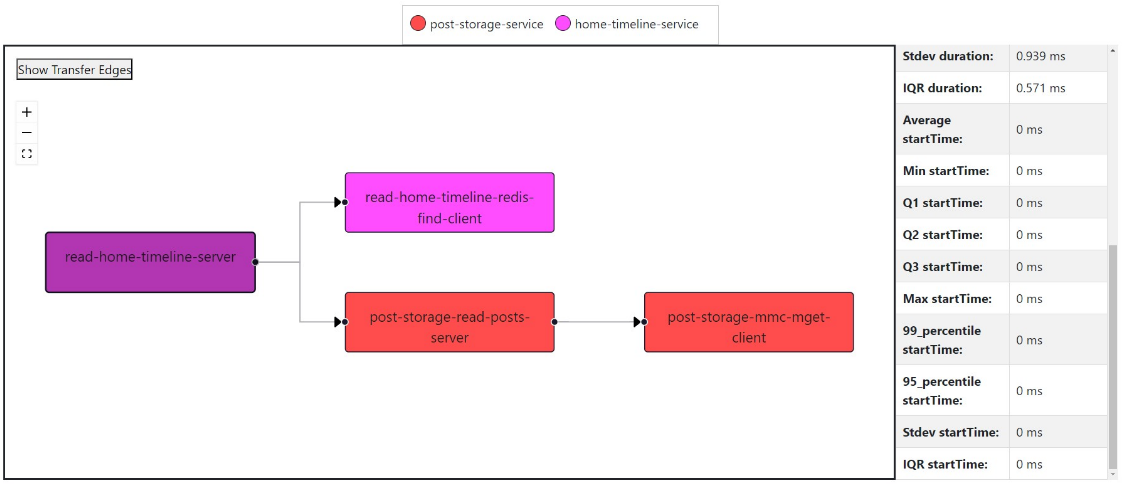

post-storage-service
home-timeline-service
Show Transfer Edges
read-home-timeline-server
read-home-timeline-redis-find-client
post-storage-read-posts-server
post-storage-mmc-mget-client

FIGURE 5.25: Operations on GroupCallGraph


● post-storage-service
● home-timeline-service
Show Transfer Edges
read-home-timeline-server
read-home-timeline-redis-find-client
post-storage-read-posts-server
post-storage-mmc-mget-client

Operation name:	read-home-timeline-server
Average duration:	1.606 ms
Min duration:	0.67 ms
Q1 duration:	1.091 ms
Q2 duration:	1.309 ms
Q3 duration:	1.662 ms
Max duration:	13.796 ms
95_percentile duration:	3.613 ms
99_percentile duration:	5.2 ms
Stdev duration:	0.939 ms
IQR duration:	0.571 ms

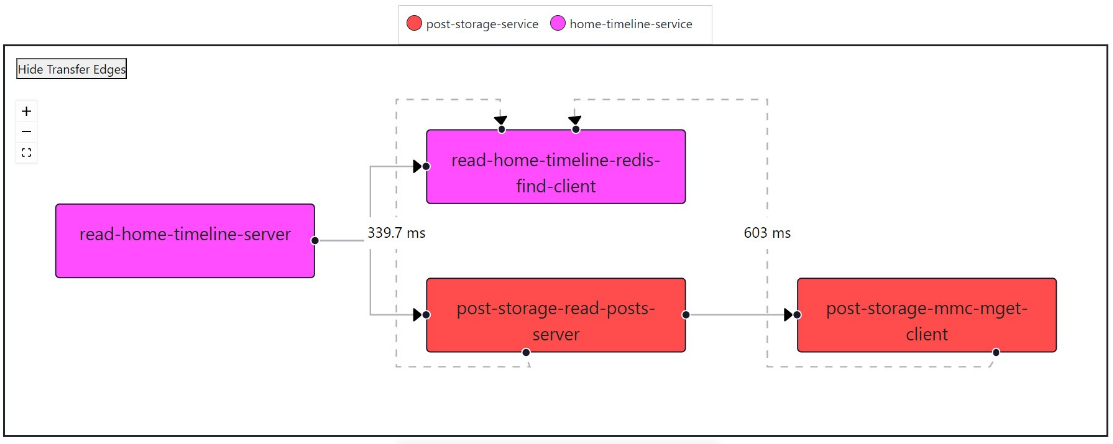
FIGURE 5.26: Operation selection and statistics (1/2)


● post-storage-service
● home-timeline-service
Show Transfer Edges
read-home-timeline-server
read-home-timeline-redis-find-client
post-storage-read-posts-server
post-storage-mmc-mget-client

Stdev duration:	0.939 ms
IQR duration:	0.571 ms
Average startTime:	0 ms
Min startTime:	0 ms
Q1 startTime:	0 ms
Q2 startTime:	0 ms
Q3 startTime:	0 ms
Max startTime:	0 ms
99_percentile startTime:	0 ms
95_percentile startTime:	0 ms
Stdev startTime:	0 ms
IQR startTime:	0 ms

FIGURE 5.27: Operation selection and statistics (2/2)

The "Show Transfer Edges" button can be found in the left corner. Clicking this button will display additional dashed line edges that represent transfer in relation to the call tree. If there are no transfer edges, the button will be replaced with the text
"No transfer edges."



post-storage-service
home-timeline-service
Hide Transfer Edges
read-home-timeline-server
read-home-timeline-redis-find-client
post-storage-read-posts-server
post-storage-mmc-mget-client
339.7 ms
603 ms

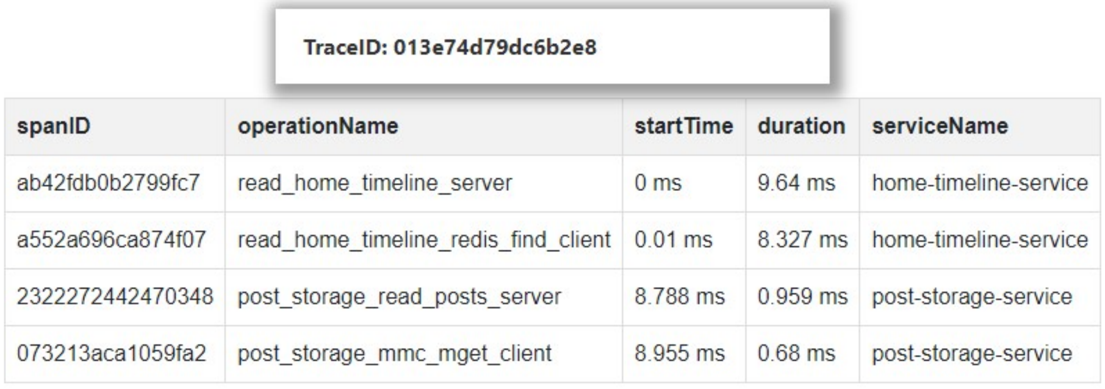
FIGURE 5.28: Show transfer edges

Implementation

Both call graphs are react components which were created utilizing the reactflow [26] library for nodes and edges as well as the dagre [1] library for layout. The transfer edges from GroupGraph were built using the'react-flow-smart-edge' [19] package.

5.3.6 Table

The Group Table(Fig 5.29) offers a user-friendly way to view operation statistics. It consists of two distinct tables: one for duration times and another for start times. Operations are listed in rows and statistical data is organized in columns. By clicking on a row you can highlight and select a specific operation.

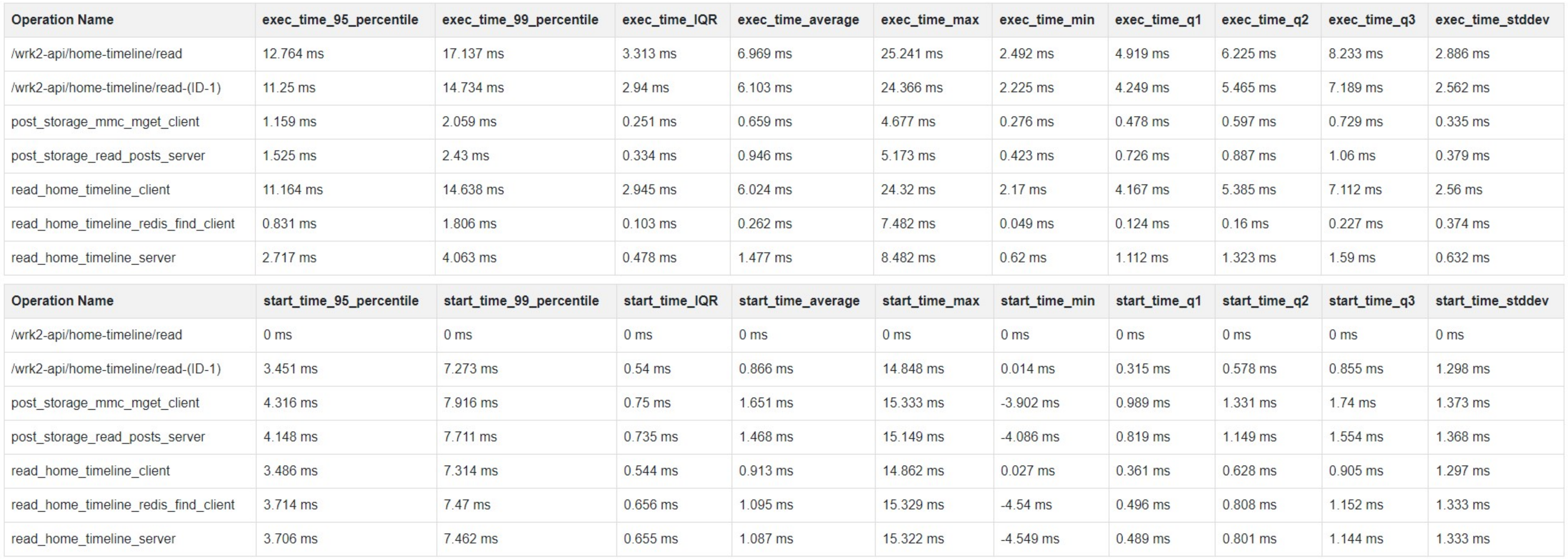


TraceID: 013e74d79dc6b2e8

spanID	operationName	startTime	duration	serviceName
ab42fdb0b2799fc7	read_home_timeline_server	0 ms	9.64 ms	home-timeline-service
a552a696ca874f07	read_home_timeline_redis_find_client	0.01 ms	8.327 ms	home-timeline-service
2322272442470348	post_storage_read_posts_server	8.788 ms	0.959 ms	post-storage-service
073213aca1059fa2	post_storage_mmc_mget_client	8.955 ms	0.68 ms	post-storage-service

FIGURE 5.30: Table for selectedTrace

The Trace Table(Fig.5.30) is an alternative method for viewing traces in a basic table format. Similar to the call graph case, the trace table requires a selected trace. Spans are displayed in rows, and their attributes are arranged in columns. By selecting a row, you can highlight and choose a specific span.



Operation Name	exec_time_95_percentile	exec_time_99_percentile	exec_time_IQR	exec_time_average	exec_time_max	exec_time_min	exec_time_q1	exec_time_q2	exec_time_q3	exec_time_stddev
/wrk2-api/home-timeline/read	12.764 ms	17.137 ms	3.313 ms	6.969 ms	25.241 ms	2.492 ms	4.919 ms	6.225 ms	8.233 ms	2.886 ms
/wrk2-api/home-timeline/read-(ID-1)	11.25 ms	14.734 ms	2.94 ms	6.103 ms	24.366 ms	2.225 ms	4.249 ms	5.465 ms	7.189 ms	2.562 ms
post_storage_mmc_mget_client	1.159 ms	2.059 ms	0.251 ms	0.659 ms	4.677 ms	0.276 ms	0.478 ms	0.597 ms	0.729 ms	0.335 ms
post_storage_read_posts_server	1.525 ms	2.43 ms	0.334 ms	0.946 ms	5.173 ms	0.423 ms	0.726 ms	0.887 ms	1.06 ms	0.379 ms
read_home_timeline_client	11.164 ms	14.638 ms	2.945 ms	6.024 ms	24.32 ms	2.17 ms	4.167 ms	5.385 ms	7.112 ms	2.56 ms
read_home_timeline_redis_find_client	0.831 ms	1.806 ms	0.103 ms	0.262 ms	7.482 ms	0.049 ms	0.124 ms	0.16 ms	0.227 ms	0.374 ms
read_home_timeline_server	2.717 ms	4.063 ms	0.478 ms	1.477 ms	8.482 ms	0.62 ms	1.112 ms	1.323 ms	1.59 ms	0.632 ms

Operation Name	start_time_95_percentile	start_time_99_percentile	start_time_IQR	start_time_average	start_time_max	start_time_min	start_time_q1	start_time_q2	start_time_q3	start_time_stddev
/wrk2-api/home-timeline/read	0 ms	0 ms	0 ms	0 ms	0 ms	0 ms	0 ms	0 ms	0 ms	0 ms
/wrk2-api/home-timeline/read-(ID-1)	3.451 ms	7.273 ms	0.54 ms	0.866 ms	14.848 ms	0.014 ms	0.315 ms	0.578 ms	0.855 ms	1.298 ms
post_storage_mmc_mget_client	4.316 ms	7.916 ms	0.75 ms	1.651 ms	15.333 ms	-3.902 ms	0.989 ms	1.331 ms	1.74 ms	1.373 ms
post_storage_read_posts_server	4.148 ms	7.711 ms	0.735 ms	1.468 ms	15.149 ms	-4.086 ms	0.819 ms	1.149 ms	1.554 ms	1.368 ms
read_home_timeline_client	3.486 ms	7.314 ms	0.544 ms	0.913 ms	14.862 ms	0.027 ms	0.361 ms	0.628 ms	0.905 ms	1.297 ms
read_home_timeline_redis_find_client	3.714 ms	7.47 ms	0.656 ms	1.095 ms	15.329 ms	-4.54 ms	0.496 ms	0.808 ms	1.152 ms	1.333 ms
read_home_timeline_server	3.706 ms	7.462 ms	0.655 ms	1.087 ms	15.322 ms	-4.549 ms	0.489 ms	0.801 ms	1.144 ms	1.333 ms

Figure 5.29: Table for selectedGroup shows statistics for operations

5.4 Discovered artifacts

5.4.1 Negative transfer times

While analyzing the statistics of start times in different groups, we noticed that in certain groups, the minimum start time values are less than 0. This is concerning because of the method we use to calculate the new start time for each span within the trace. We calculate the new start times by subtracting the root's original start time from the span's original start time. In this scenario, resulting the new start time for the root should be 0, and there should not be any negative start times for its successors.

Yet, certain spans within the trace exhibited start time values lower than the root of the trace. To address this, we developed a code to pinpoint and confirm instances where such situations arise. Example results are shown in listing 5.1.

Listing 5.1: The output of our code

```
{
    "trace_id": "00232f628e62893e",
    "trace_root_span_id": "00232f628e62893e",
    "trace_root_span_operation_name": "/wrk2-api/home-timeline/read",
    "trace_root_span_start_time": 1667496189420000,
    "original_span_id": "e933d74bc56c47ba",
    "original_span_start_time": 1667496189418350,
    "original_span_operation_name": "read_home_timeline_redis_find_client"
}
```

Our suspicions were validated. Upon analyzing the code output, it was confirmed that the start time value of the span with ID e933d74bc56c47ba(i.e. a called span) was indeed lower than its root span. Furthermore, this was not an isolated incident, as we discovered that approximately 0.2% of traces in a typical JSON file contain at least one span with a negative start time value. The exact cause of this issue is unclear, but it is likely due to minor inconsistencies in clock synchronization among the microservices.

Despite this issue, these traces are included in statistical calculations and still contain valuable information, as most of the spans in these traces do not have this problem. A special group was created for the traces with negative start times. By selecting the specific group called "Negative start times" the user can browse these traces and check on the call graph which spans are affected and which microservices are involved. This group is only for browsing and contains traces from various groups.

5.4.2 Invalid parent span

During the process of grouping we may encounter spans whose parent does not exist in the entire trace. This can be easily identified by the presence of the warning message:
"invalid parent span IDs='missingParentID'; skipping clock skew adjustment".(Example in Listing 5.2)

Listing 5.2: Invalid parent span example

```
{
   "traceID": "005fe0fbf9bd0f74",
   "spanID": "d0da934930a94c4f",
   "flags": 1,
   "operationName": "read_home_timeline_server",
   "references":
   [
        {
            "refType": "CHILD_OF",
            "traceID": "005fe0fbf9bd0f74",
            "spanID": "8c6e23338187e5dc"
        }
   ],
   ...
    "warnings":
     [
      "invalid parent span IDs=8c6e23338187e5dc; skipping clock skew
          adjustment"
     ]
}
```

There are two scenarios:

1. The span is the root of the trace. The main method to find the root is by finding a span without a parent reference. If there are no spans without a parent reference but one of the spans has the "invalid parentID" warning then that span becomes the root as its parent does not exist in the trace. However this suspected span must also have at least one child.

2. It happens when the span don't have any child then the span is the root of another call tree within the trace. This call tree typically consists of a single span. On callGraph, this call tree is displayed below the main call tree.(Fig.5.31)

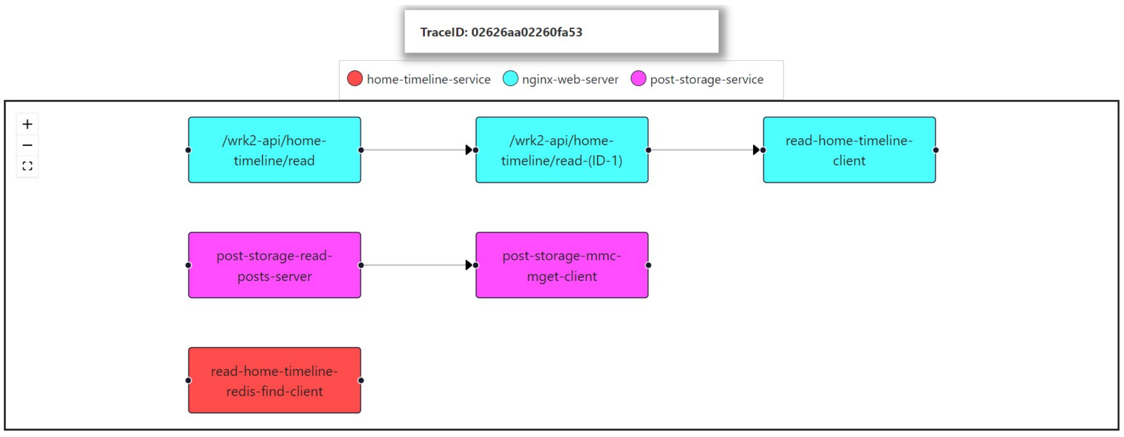

TraceID: 02626aa02260fa53
🟥 home-timeline-service
🟦 nginx-web-server
🟪 post-storage-service
+
−
/wrk2-api/home-timeline/read
/wrk2-api/home-timeline/read-(ID-1)
read-home-timeline-client
post-storage-read-posts-server
post-storage-mmc-mget-client
read-home-timeline-redis-find-client

Figure 5.31: Invalid parent: multiple call trees

We investigated the root cause of this issue. It is possible that the span data is incorrect due to a misconfiguration of the OpenTelemetry SDKs. This could be caused by using different ID lengths. If the trace IDs are not the same, Jaeger will not identify the span as belonging to the same trace.[3] Another reason for this warning could be a discrepancy in the order of span reporting. It can happen when the child span is sent to the collector before the parent span, causing inconsistencies that trigger the warning message.[4]

5.5 Transfer time

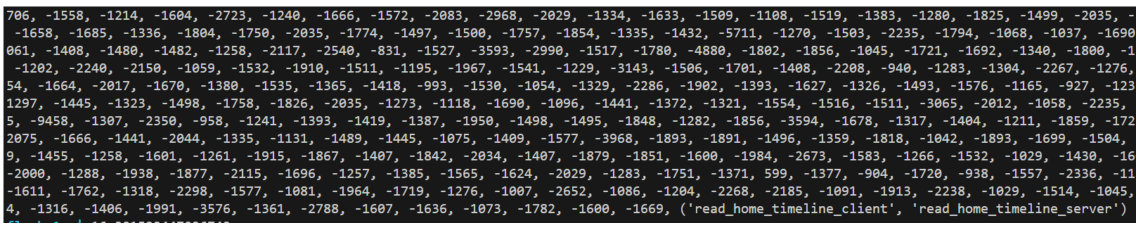
In this section, we describe an implementation of transfer time algorithm and obtained results.

5.5.1 Calculation of transfer times

To check if spans are not concurrent, two methods are employed. The first method involves counting how many times one span preceded another in all traces within the group while simultaneously tallying the total number of spans in the group. If, in each trace within the group, spans are found to precede each other, it indicates that they can be considered for transfer time. However, in practice, this method does not account for cases where even if spans preceded each other and were not concurrent, the execution times of spans could overlap, causing the trace not to be counted. It treats the pair of spans as unsuitable for transfer time statistics, even if they are suitable.

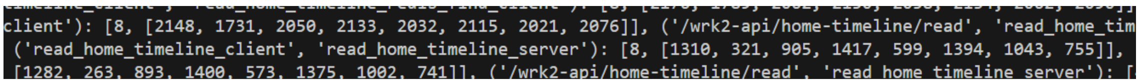
The second method addresses this issue. For each pair of spans where the precedence exceeds 20% (this parameter is chosen for optimization purposes, aiming to filter out cases where precedence occurs in a very small number of traces, which would not be of interest to the potential user for transfer time statistics), by filtering out the least significant cases, we can now check potential pairs of spans to ensure they are not executed concurrently. The algorithm then proceeds to check in the call graph whether one span is the ancestor of another. This is done recursively, as the call graph is a nested dictionary. Filtering unnecessary pairs of spans is crucial to save processing time. A unit used in the project to represent transfer time in microseconds.

It may be intriguing why there are two conditions for spans to be included in the predecessor. The first condition is optimization-related, aiming to reduce the number of spans for processing to check if the successor span in the table is indeed a predecessor. The primary reason for this approach is that there are spans where the transfer time is mostly negative. We exclude negative time from the graph, which may be counterintuitive in interpretation. The values are typically negative (Fig. 5.32), but after filtering out the negative values, occasional positive values still remain (Fig. 5.33). Here, positive time is observed in eight cases, but for the vast majority, it is negative. Therefore, a two-stage filtration is needed.



```
706, -1558, -1214, -1604, -2723, -1240, -1666, -1572, -2083, -2968, -2029, -1334, -1633, -1509, -1108, -1519, -1383, -1280, -1825, -1499, -2035, -
 -1658, -1685, -1336, -1804, -1750, -2035, -1774, -1497, -1500, -1757, -1854, -1335, -1432, -5711, -1270, -1503, -2235, -1794, -1068, -1037, -1690
061, -1408, -1480, -1482, -1258, -2117, -2540, -831, -1527, -3593, -2990, -1517, -1780, -4880, -1802, -1856, -1045, -1721, -1692, -1340, -1800, -1
 -1202, -2240, -2150, -1059, -1532, -1910, -1511, -1195, -1967, -1541, -1229, -3143, -1506, -1701, -1408, -2208, -940, -1283, -1304, -2267, -1276,
54, -1664, -2017, -1670, -1380, -1535, -1365, -1418, -993, -1530, -1054, -1329, -2286, -1902, -1393, -1627, -1326, -1493, -1576, -1165, -927, -123
1297, -1445, -1323, -1498, -1758, -1826, -2035, -1273, -1118, -1690, -1096, -1441, -1372, -1321, -1554, -1516, -1511, -3065, -2012, -1058, -2235,
5, -9458, -1307, -2350, -958, -1241, -1393, -1419, -1387, -1950, -1498, -1495, -1848, -1282, -1856, -3594, -1678, -1317, -1404, -1211, -1859, -172
2075, -1666, -1441, -2044, -1335, -1131, -1489, -1445, -1075, -1409, -1577, -3968, -1893, -1891, -1496, -1359, -1818, -1042, -1893, -1699, -1504,
9, -1455, -1258, -1601, -1261, -1915, -1867, -1407, -1842, -2034, -1407, -1879, -1851, -1600, -1984, -2673, -1583, -1266, -1532, -1029, -1430, -16
-2000, -1288, -1938, -1877, -2115, -1696, -1257, -1385, -1565, -1624, -2029, -1283, -1751, -1371, 599, -1377, -904, -1720, -938, -1557, -2336, -11
-1611, -1762, -1318, -2298, -1577, -1081, -1964, -1719, -1276, -1007, -2652, -1086, -1204, -2268, -2185, -1091, -1913, -2238, -1029, -1514, -1045,
4, -1316, -1406, -1991, -3576, -1361, -2788, -1607, -1636, -1073, -1782, -1600, -1669, ('read_home_timeline_client', 'read_home_timeline_server')
```

FIGURE 5.32: Example of minus transfer time



```
client'): [8, [2148, 1731, 2050, 2133, 2032, 2115, 2021, 2076]], ('/wrk2-api/home-timeline/read', 'read_home_tim
 ('read_home_timeline_client', 'read_home_timeline_server'): [8, [1310, 321, 905, 1417, 599, 1394, 1043, 755]],
 [1282, 263, 893, 1400, 573, 1375, 1002, 741]], ('/wrk2-api/home-timeline/read', 'read_home_timeline_server'): [
```

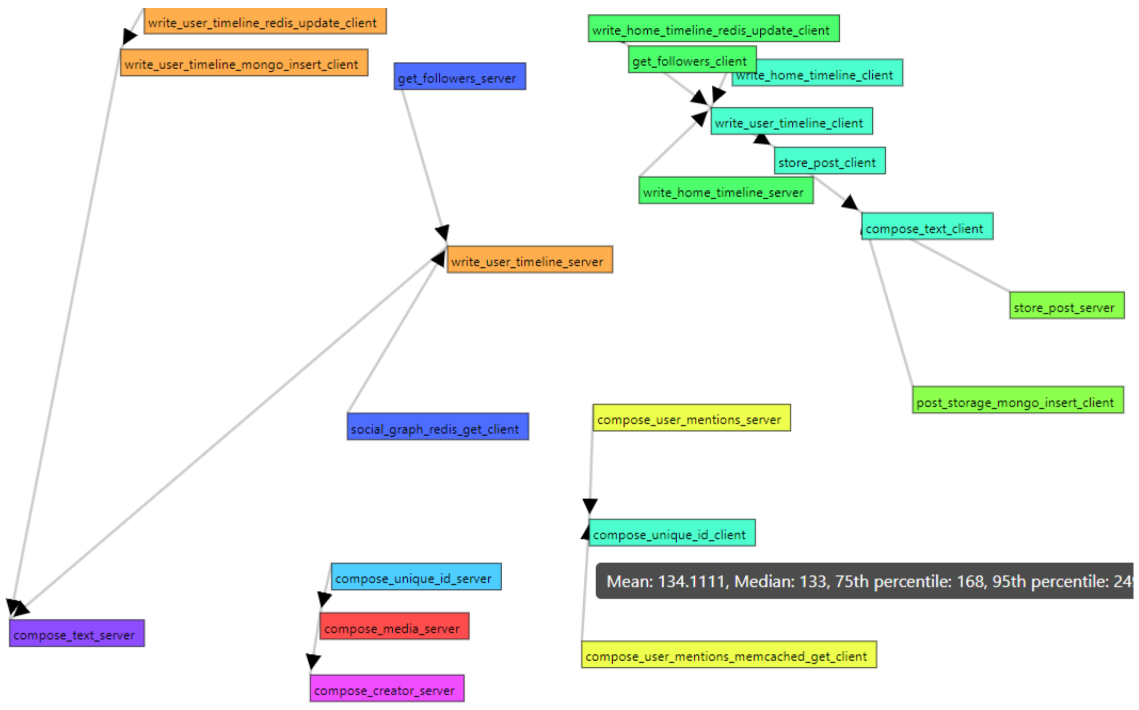
FIGURE 5.33: Example of plus transfer time

5.5.2 Transfer times graph visualization

The JavaScript library D3.js [17] was used to generate the graph. D3.js (Data-Driven Documents) is a popular JavaScript library used for creating interactive and dynamic data visualizations on websites. This library requires data in the form of a JSON.

Listing 5.3: Input of d3 code

```
{
    "nodes": [{
    "id": operation name,
    "x": coordinate x of node,
    "y": coordinate y of node
}],
    "links": [{
    "Statistic": list of statistics of these connections,
    "Index": index of connection,
    "source": source node of connection,
    "target": target node connection
}]
}
```


write_user_timeline_redis_update_client
write_user_timeline_mongo_insert_client
get_followers_server
write_home_timeline_redis_update_client
get_followers_client
write_home_timeline_client
write_user_timeline_client
store_post_client
write_home_timeline_server
compose_text_client
write_user_timeline_server
store_post_server
post_storage_mongo_insert_client
social_graph_redis_get_client
compose_user_mentions_server
compose_unique_id_client
compose_unique_id_server
compose_media_server
compose_creator_server
compose_text_server
compose_user_mentions_memcached_get_client

Mean: 134.1111, Median: 133, 75th percentile: 168, 95th percentile: 24

Figure 5.34: Transfer time graph in application

The transfer time graph see Fig 5.33 is designed with interactivity, allowing users to move each vertex according to their preferences. Upon hovering over the connections between vertices, users can see detailed statistics for the respective connection, derived from the comprehensive analysis of all traces within the group. The provided statistics include the average time between these operations during precedence, followed by the median, 75th percentile, and 95th percentile. Notably, the colors assigned to the graph align with the colors of operations in the call graph, enhancing the intuitive interpretation of the visual representation. The connection's lengths are proportionate to the times of precedence between these spans. Consequently, a greater transfer time results in a more considerable distance between the associated spans on the graph.

5.6 Testing

5.6.1 Functional correctness

To test the correctness of our application, we developed a Python script to generate artificial tracing data in JSON format. Listing 5.3 showcases the usage of this script.

Listing 5.4: Test generator usage instruction

```
$ python3 scripts/test_generator.py -h

optional arguments:
  -h, --help            show this help message and exit
  -t, --traces          Number of traces
  -s, --spans           Max number of spans per trace
  -c, --comm-time       Fixed time of communication
  -d, --duration        Fixed duration time
  -p, --processes       Number of processes - esentially microservices
```

Features tested and conclusions are as follows:

- Call graphs - after generating multiple instances of artificial data with linear or forked traces we were able to validate, that call graphs are presented correctly. The number of elements was aligned with the number of spans per trace set in the generator. For every test, we observed that operations are assigned to the correct services.
- Scatter plots - This part was the most challenging because we had to prepare the data and calculate expected results. In the end we validated that scatter plots present accurate ranges of values.
- Tables with statistical data - since the test generator allows to set duration time values to a fixed number (using the –**duration** flag), we were able to test if the calculated statistic align with values set by us in the script. It turned out, that statistics were computed properly, and every manual change in values resulted in expected adjustment of values in our application.
- Transfer times - using the –**comm-time** flag we set transfer time between every span to a fixed value. After inspecting the tab with transfer times we confirmed, that these values were correctly presented on each connection. We also conducted tests with random values for smaller traces and checked if transfer times were computed correctly - once again, validating that they aligned with generated data.

Using the developed script and insight of the Intel representative we were able to properly test the software and determine its functional correctness.

5.6.2 Performance

Evaluating the performance is important in optimizing application usability. To achieve this, a set of test files with varying sizes, ranging from 1 MB to 100 MB was prepared. Each of these files was loaded into the application, followed by a computation process on the backend in order to prepare call graph representations and transfer time values. Another aspect

Collected statistics, including average response time, minimum time, maximum time, and median (shown in table 5.1) provide a comprehensive overview of the system's performance based on the size of the processed file. The average response time helps us understand the overall trend, while the minimum and maximum times illustrate the range of variability. The obtained data was carefully analyzed and then presented in figure 5.34, illustrating the relationship between file size and response time.

Using the script to generate artificial data, we prepared multiple sample files with artificial traces - all of them had maximum of 32 spans per trace. We performed tests for these instances, each size was tested 5 times:

- 1MB - 64 traces (2048 spans)
- 10MB - 640 traces (20480 spans)
- 50MB - 3000 traces (96000 spans)
- 100MB - 6200 traces (198400 spans)

To gather response times we utilized built-in performance measuring tools in Chrome web browser (Developer Tools tab). The test was run on a platform with:

- CPU - Ryzen 7 5800H
- RAM - 2x8GB DDR4 3200MHZ
- Storage - Samsung 980 NVMe SSD
- Browser - Google Chrome
- OS - Windows 11 Professional

The results are as follows:

File Size (MB)	Response Times (s)			
	Avg RS	Max RS	Min RS	Median RS
1	2.03	2.25	1.78	2.15
10	45.43	46.31	44.90	45.72
50	711.32	738.23	690.45	703.23
100	2357.20	2480.62	2285.45	2340.69

TABLE 5.1: Response times analysis

Since the structure of telemetry data is far from being simple and straight-forward to analyze, our algorithm results in relatively high complexity of O(n^3) which can be seen on figure 5.34.

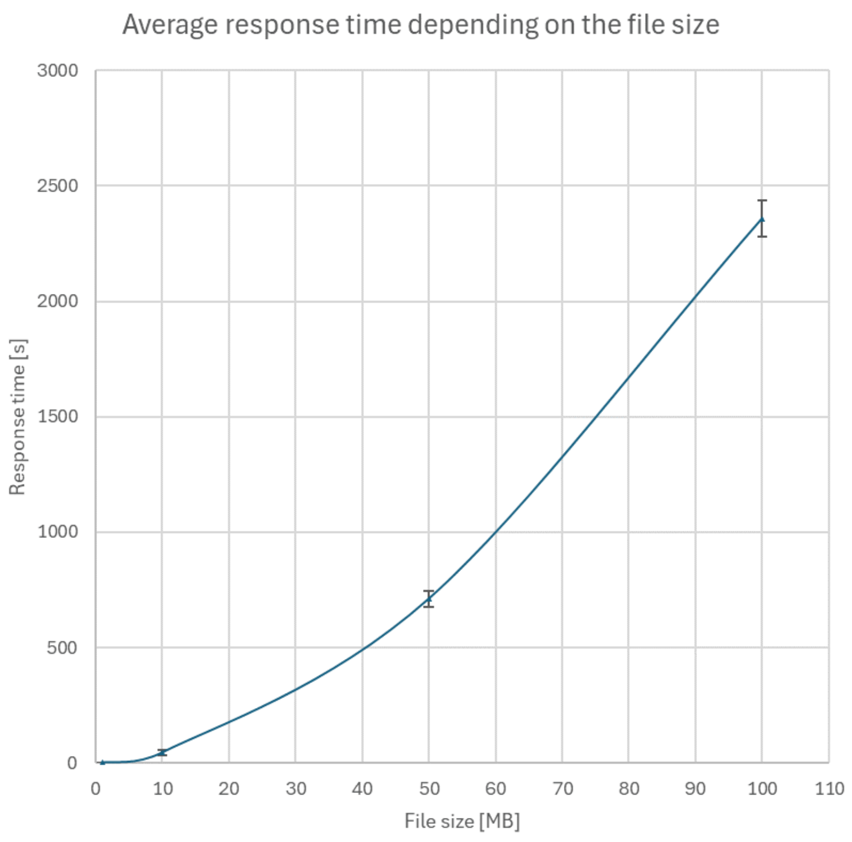


FIGURE 5.35: Performance depending on file size

Chapter 6

Conclusions

6.1 Projects plans

The project successfully realized plans regarding the development of advanced software for analyzing microservices-based applications. The implemented functionalities include the analysis of the sequence of calls for individual operations in a component referred to as a call graph. Another achieved milestone is the algorithm for calculating transfer times, allowing the observation of times at which individual operations unrelated by concurrency are spaced apart. Additionally, components such as scatter plots and histograms were implemented, illustrating the distribution of individual spans, traces, and groups over time, along with statistics on these operations, such as execution time, start time of the operation, or span. These statistics are highly flexible, allowing them to be viewed for different abstraction layers, such as groups, individual spans, or operations. Initially, it planned to analyze statistics of communication times between operations and spans. Unfortunately, the current logs from Jaeger do not contain sufficient information to calculate communication times. With this aspect, DSB log is poorly instrumented. A minimum extension to allow communication time calculation is recording times when microservice is called in the parent process and the time when the called returns to the parent. For this reason, the concept of transfer time was introduced to trace communication delays indirectly. It was also influenced to analyze the critical path between operations. It appeared that the microservice-based applications of DSB. to the extent of data obtained for this study, always had trees call-graph. This means that the root generation was always calling further descendant services and waiting for them to return. Consequently, the root operation was always the longest-running part of the Jaeger trace. In effect, searching for the longest path had little utility, because the root span was always the longest path of a single operation.

There is some ways to exand project.

- A universal platform for conducting tests under changing load (traffic) to the microservice, available computing power (
- A module generating a response time model based on load parameters, available computing power, memory, and processor type (e.g., multidimensional linear regression, any machine learning models are permissible).

The project would handle multiple files simultaneously, and the hardware available to those working on the project might prove inadequate, considering the large file sizes and resources they would occupy. However, the project has the potential for further development, especially concerning the simultaneous processing of multiple files, which can be considered in future project expansions.

Another aspect for further development is adjusting benchmark parameters based on the analysis of results from multiple files, utilizing machine learning.

The project is dynamic and forward-looking, particularly given the increasing reliance on microservices-based applications. Currently, there is a tendency to avoid solutions with a centralized structure, significantly strengthening the project's market potential.

Bibliography

[1] dagre. https://github.com/dagrejs/dagre.

[2] Death star benchmark. https://github.com/delimitrou/DeathStarBench.

[3] Invalid parent github issue 1/2. https://github.com/jaegertracing/jaeger/issues/3084.

[4] Invalid parent github issue 2/2. https://github.com/jaegertracing/jaeger/issues/2121.

[5] Opentracing repository. https://github.com/opentracing.

[6] Spec cpu benchmark. https://www.spec.org/cpu2017/.

[7] Tpc benchmark. https://www.tpc.org.

[8] Brendan Eich. Javascript, frontend programming language. https://pl.wikipedia.org/wiki/JavaScript.

[9] Cloud Native Computing Foundation. Jaeger: open source, distributed tracing platform. https://www.jaegertracing.io.

[10] Cloud Native Computing Foundation. Jaeger tracing repository. https://github.com/jaegertracing/jaeger.

[11] Cloud Native Computing Foundation. Opentelemetry repository. https://github.com/open-telemetry.

[12] Inc. GitHub. Github, version control. https://github.com/.

[13] Google. Production-grade container orchestration. https://kubernetes.io.

[14] Docker inc. Docker, containers. https://docs.docker.com/desktop/.

[15] Maxon. One of the most popular cpu benchmarks available. https://www.maxon.net/en/tech-info-cinebench.

[16] Inc. Meta Platforms. React, frontend programming language. https://pl.legacy.reactjs.org/.

[17] Observable. Visualize transfer time on web app. https://observablehq.com/documentation.

[18] Inc. Observable. D3.js, visualization library for javascript. https://d3js.org/.

[19] Tiso Alvarez Puccinelli. react-flow-smart-edge. https://github.com/tisoap/react-flow-smart-edge.

[20] Armin Ronacher. Flask, backend framework. https://flask.palletsprojects.com/en/3.0.x/.

[21] Python software. Python, programming language. https://pl.wikipedia.org/wiki/Python.

[22] Uber Technologies. React-vis, visualization library for react. https://uber.github.io/react-vis/documentation/welcome-to-react-vis.

[23] TPC. Leading benchmarking and load testing software for the worlds most popular databases. https://www.hammerdb.com.

[24] Cornell University. Social network architecture - an application deployed by dsb. https://github.com/delimitrou/DeathStarBench/blob/master/socialNetwork/figures/socialNet_arch.png.

[25] Cornell University. An open-source benchmark suite for microservices and their hardware-software implications for cloud edge systems.
https://www.csl.cornell.edu/~delimitrou/papers/2019.asplos.microservices.pdf, 2019.

[26] xyflow. reactflow. https://reactflow.dev/.