Degree Programme in Computer Engineering

First Cycle 15 credits

Comparing Virtual Threads and Reactive Webflux in Spring

A Comparative Performance Analysis of Concurrency Solutions in Spring

YO HAN JOO
CARL HANEKLINT

STOCKHOLM, SVERIGE 2023

Comparing Virtual Threads and Reactive Webflux in Spring

A Comparative Performance Analysis of Concurrency Solutions in Spring

Jämförelse av virtuella trådar och Reactive WebFlux i Spring

En jämförande analys av lösningar för samtidighet i Spring

YO HAN JOO
CARL HANEKLINT

Examensarbete inom datateknik
Grundnivå, 15 hp
Handledare på KTH: Anders Lindström
Examinator: Ibrahim Orhan
TRITA-CBH-GRU-2023:086

KTH
Skolan för kemi, bioteknologi och hälsa
141 52 Huddinge, Sverige

Sammanfattning

För att utveckla högpresterande webbtjänster i Java har Reactive WebFlux varit det främsta alternativet. Med introduktionen av virtuella trådar i Java 19 har spekulerats kunna ersätta reaktiva webbtjänster. Detta arbete presenterar prestandaskillnaden mellan Spring-applikationer som drivs av normala trådar, virtuella trådar och dess reaktiva motsvarighet Reactive WebFlux. Testerna utfördes genom att skapa tre prototyper, som användes för att anropa en slutpunkt med en satt fördröjningstid och öka antalet förfrågningar per sekund fram till systemfel. Resultaten visade att prototypen med virtuella trådar presterade något bättre än den reaktiva prototypen. Däremot är det ännu oklart om Reactive WebFlux-applikationen med en optimal webbserver skulle producera ett annat resultat. Dessutom skulle det vara intressant för framtida forskning om hur virtuella trådar fungerar med databasanvändning.

Nyckelord
Virtual threads, Spring Boot, Reactive WebFlux, Java, Concurrency

Abstract

To develop a high performant web service in Java, Reactive WebFlux has previously been utilized as the only alternative. With the introduction of virtual threads in Java 19, it has been considered that it would be an alternative to the current provided option. This paper presents the performance difference between Spring applications powered by normal threads, virtual threads, and its reactive counterpart Reactive WebFlux. The tests were done by creating three prototypes, which were used to call an endpoint with a set delay time and increasing the number of requests per second until system failure. The results show that the virtual threaded prototypes performed slightly better than the reactive prototype. The question remains whether the Reactive WebFlux application using the most optimal web server produces a different result, as well as future research on how virtual threads perform with database usage.

Keywords
Virtual threads, Spring Boot, Reactive WebFlux, Java, Concurrency

Acknowledgements

This paper is a result of a thesis project in the field of applied information technology at the Royal Institute of Technology on behalf of the company Stryda.

We want to extend our gratitude to our supervisor Anders Lindström for his guidance, support and expertise during this project. We also want to thank Sadde Hemani at Stryda for his support and contribution to this project.

Table of Contents

1 Introduction

This chapter presents the problem statement, goals and scope of the thesis.

1.1 Problem Statement

Today, distributed systems face more demand than ever. As more people use more internet services and services grow to be more complex, demands of the underlying technologies also increase by requiring greater performance, availability and scalability.

One of the demands is concurrency, which is a system's capacity to manage many requests at the same time. Concurrency is essential for web services that must handle a high rate of requests in real-time while not sacrificing the quality of the provided service. A way for companies today to combat this problem is to introduce the microservice architecture, where small web services perform specialized tasks to form a complete distributed system.

There are multiple programming languages that can be used to develop web services. With Java being one of the most used programming languages used in the enterprise space and Spring being a widely adopted framework for building web services, there exists two popular frameworks that can be used: Reactive WebFlux and Spring Boot. Reactive WebFlux is a reactive framework designed to handle high concurrency with operations that utilizes blocking operations and the latter being an imperative framework which focuses on simplicity, and ease of use.

Since concurrency is in high demand, Reactive WebFlux has been used extensively for Java web services where performance is important. The caveat of this method has been that the programming paradigm used to create such services has been considered unorthodox by some researchers as well as developers, as the paradigm differs to traditional imperative programming. This can be a challenge to newcomers regarding code readability and in turn increase development time. However, a contender to Reactive WebFlux has been introduced with the release of Java 19.

With the introduction of virtual threads, a more lightweight approach to creating threads in Java 19 as a preview feature, there have been theories surrounding the fact that it might enable non-reactive Spring Boot framework, which is considered by many for being easy-to-use, to be used in large concurrent applications with minimal configuration to achieve the same or better performance achieved with reactive frameworks.

This could, for instance, be a viable option for the company Stryda, the employer of this study, which is a company focused on gamification of e-sports players. The company utilizes the use of microservices and synchronized communication between the web services and the front-end web application. Non-reactive, virtual threaded

spring boot could be a viable option to combat the growing demand for scalability and fast development.

1.2 Goals

The aim of this work is to analyze measurable performance differences between Virtual Thread powered Spring Boot services and Reactive WebFlux services. Three prototypes will be developed and compared to showcase the potential benefits and limitations of each technology. One prototype will use virtual threads, while the other will use Reactive WebFlux. The third one will be a non-virtual thread Spring Boot service.

The task of the prototypes will be to perform data aggregation from two separate underlying web services and return the aggregated data to the user*.

The key metrics measured during testing will be requests per second, latency, CPU- and memory usage. These metrics will be measured in multiple tests with following parameters that can be altered during each test:

- Payload size returned from each underlying microservice.
- Delay from each underlying microservice.
- Number of concurrent requests per second.

The measured metrics in each prototype will be evaluated to determine the strengths and weaknesses of each approach. This evaluation will help identify areas where further improvements could be made. All of this will be made possible using external software that is designed to monitor the above-mentioned metrics.

*More on the purpose of this task in chapter 3

1.3 Scope of the Project and Delimitations

To minimize the number of variables that can impact the result, measures will be taken to limit the library differences that are used on the prototypes. This will ensure that we compare the performance difference in virtual threads and reactive programming instead of libraries that are used.

All tests are conducted on local machines. This will ensure that no external factors such as internet connection affect the tests conducted.

2 Theory and Background

This chapter will present previous work in the area of Java web frameworks with virtual threads and background knowledge required to understand the goals of the project. The first part will convey background knowledge in what concurrency is and how it works in Java. After a short introduction of the Spring environment, previous relevant work regarding Spring Boot, Reactive WebFlux, and Virtual threads will be discussed. This chapter will also discuss how benchmarking can possibly be implemented in the methodology, and the available tools that can be used.

2.1 Concurrency

Concurrency refers to performing multiple tasks at the same time [1]. From a system perspective, concurrency expresses the capacity of an operating system to handle processes at the same time. It can be achieved by a combination of scheduling and parallelism. Scheduling means that a CPU core switches execution between multiple threads quickly to create the illusion of multiple processes executing at the same time. Parallelism refers to the operating system utilizing multiple cores, to literally execute threads at the same time [2]. An example of concurrency on an application level is the listen for user input while performing other complex operations. Similarly, concurrency in web applications can refer to the ability for the application to handle multiple user requests at the same time. For example, user A and user B request a web page from a web service at the same time. The webservice then must process and deliver a response to the two users. Now scale it to 1000 users doing the same request. How well the web service can handle these requests, such as the time it takes to respond to each request is a measurement of the concurrency capacity of the web service.

2.1.1 Reactive Programming

An approach for creating concurrent web services is with reactive programming. Reactive programming is a programming paradigm where data processing is approached in an event-driven and non-blocking fashion with [3]. Event-driven means that software executes whenever events are triggered. Events are any change of state in the application, such as user requests or data updates. The term non-blocking states that a thread moves on to do other tasks while waiting for a resource to finish its execution. Since user requests are similar to events, reactive programming's event-driven and non-blocking approach has been seen as favorable in web services that require thousands of concurrent users.

2.2 Java Concurrency

Concurrency in Java is achieved through the Java Thread API, which is a form of platform thread that wraps around an OS thread [4] which allows Java applications to run concurrent code. Threads are utilized in web services where each thread is created for each new request from a consumer [5].

2.2.1 Reactive Programming in Java

Another way of achieving concurrency in Java is with asynchronous code that utilizes reactive programming methodologies. Reactive libraries such as Mutiny, Project Reactor, and RxJava, are popular choices that provide foundations for introducing asynchronous code into Java applications, especially for creating highly scalable, concurrent web services [5].

2.2.2 Virtual Threads

Virtual Threads were introduced as a preview API in JEP 425 and delivered in Java Development Kit (JDK) 19 as a preview feature [6]. Virtual Threads are lightweight and run Java code on an underlying OS thread but do not capture the OS thread for the code's entire lifetime. This means that many virtual threads can run their Java code on the same OS thread, effectively sharing it. By sharing the same OS thread, a far greater number of virtual threads can be instantiated than normal platform threads [4]. Using virtual threads instead of regular threads in Java web services may enable processing thousands of requests without the need to create additional operating system threads possible.

2.3 Spring Framework

Multiple possible solutions exist for creating a web service in Java. One of the solutions to create web services is the Spring framework [7], which is an open-source framework that is part of the Java ecosystem. The Spring framework is used at Stryda as the framework for creating web services.

2.3.1 Spring Boot

Spring Boot is a framework built upon the Spring Framework that offers quicker development of web services through three core features: autoconfiguration, opinionated approach, and the ability to create standalone applications [7] [8]. By automatically configuring and including starter dependencies, the required work to set up a working web service is reduced. It offers the ability to cherry-pick modules that should be used, such as the normal web module and the reactive module, providing further customization options.

2.3.2 Reactive WebFlux

module and is a framework used for building non-blocking, event-driven web services [9]. It is built upon the Project Reactor library which utilizes Reactive streams, that implements a publisher and subscriber pattern. The pattern allows for asynchronous, non-blocking requests to be possible with the subscribers reacting to publisher events. The result of this is a powerful and efficient alternative that can be used for creating scalable applications which itself also offers high throughput.

2.3.3 Mono and Flux

Mono [10] and Flux [11] are specialized publishers in the Reactive streams API. The publisher Mono takes zero or one data, computes it, and then sends the computed data to a subscriber. Flux is the publisher that takes lists of data and computes all of them in parallel. When all the data in the lists are computed, the list of data is returned to a subscriber. Understanding Mono and Flux is important to understand the Reactive WebFlux module.

2.3.4 Blocking and Non-blocking Web Servers

Two types of servers exist for Reactive WebFlux applications: blocking and non-blocking web servers. The most typical web server is Apache Tomcat, which has been supported by default on Spring Boot applications from version 1.0 [12]. Reactive WebFlux does have support for alternative web servers that support non-blocking operations with the most popular web servers being servers such as Netty and Undertow [13].

2.4 Related work

This chapter presents current knowledge and work done in this field related to this thesis. The aim of this chapter is to summarize current knowledge and discover potential gaps in knowledge on the performance of Java virtual threads in a Spring Environment.

2.4.1 Reactive WebFlux and Spring Boot

In a study on performance differences between reactive and non-reactive Java frameworks conducted at Mid Sweden University by André Nordlund and Niklas Nordström [14], four different prototypes were developed. The prototypes were created using Spring Boot, Reactive WebFlux, non-reactive Quarkus, and reactive Quarkus. The study was performed two times with the performance of each prototype being determined by their average response time, CPU, and heap usage during 250 and 500 concurrent users. The reactive Quarkus outperformed in latency compared to the other frameworks in low transactional sizes but had the highest latency during high transactional sizes. The non-reactive and reactive Quarkus prototypes outperformed the Spring Boot prototypes in CPU and heap usage. Although similar, the non-reactive Spring Boot application outperformed the Reactive Spring Boot application by 10-20% in each category.

In another case study comparing the imperative and reactive approaches in Java web service development by Sebastian Iwanowski and Grzegorz Kozieł [15], a Spring Boot, and a Reactive WebFlux prototype were created to measure their performance, stability, and development time. The latency, CPU usage, and heap usage were measured for 30, 300, and 3000 requests per second. For each test conducted, the reactive Spring application outperformed the non-reactive in high requests per second by having less latency, and less requests that took 10 seconds or more. On a 2CPU and 4GB system, the imperative method consumed 52.8% CPU, and 11.27% RAM usage. The reactive methodology used 33.67% and 10% RAM usage. Iwanowski and Kozieł mention that the reactive method had a longer development time by requiring more lines of code for the entire application.

Karl Dahlin conducted a case study to evaluate the performance of the Reactive WebFlux [16] for database operations. Two prototypes were developed. One using Spring Boot MVC and another using Reactive WebFlux. The database libraries were also different. The CPU usage, heap usage, and execution times were measured by sending 200 000 insert operations on a database table through each prototype. The Reactive WebFlux application had a response time of 6 minutes and 7 seconds, close to 0% CPU usage and 250MB heap usage. The non-reactive had a response time of 1 hour 25 minutes and 6 seconds, close to 0% CPU usage, and peaks of 250MB heap

usage. The study gives an insight on the performance benefits of reactive applications during large transactions of data.

2.4.2 Performance of Java Threads and Virtual Threads

There are several case studies done regarding the performance differences between normal OS threads and Java's new virtual threads. In the paper "Achieving Efficient Structured Concurrency through Lightweight Fibers in Java Virtual Machine" by P. Pufek, D. Beronić, B. Mihaljević and A. Radovan [17], the paper compares the latency per requests between Fiber, which is another name for Java virtual threads, and Java OS Threads. The paper describes that the current Java thread implementation is not scalable in concurrent applications due to the high number of OS threads being created during runs. The HTTP servers were created within the boundary of Java's standard library given in the Java Development Kit (JDK). Each prototype server was given a wait time for each request it received. Each test run had 75 requests per second being sent to the HTTP server. According to the results, when the sleep time per request was set to 1ms, the latency difference between fiber implementation and standard implementation was not that great. However, the 10 milliseconds wait time had a significant difference in latency between OS threads and virtual threads. The OS thread had a mean latency of 25 milliseconds, while the virtual thread implementation had a mean latency of 4 milliseconds.

In the research paper "Comparison of Structured Concurrency Constructs in Java and Kotlin – Virtual Threads and Coroutines" done by D. Beronić, L. Modrić, B. Mihaljević and A. Radovan [18], four HTTP server prototypes were created to measure the number of OS threads started and amount of heap used in Megabytes by each prototype after each test run. Each operation in the test consisted of creating an object, writing it into a file, and then returning it as a response. Normal Java and Kotlin threads started around 100 000 threads and used around 300-400 Megabytes of heap. The Kotlin coroutine started around 23 OS threads with 52-99 Megabytes of heap used. The Java virtual threads started around 35 OS threads and utilized 16-64 Megabytes of heap. The paper discusses that structured concurrent approaches such as Kotlin and Java Virtual Threads had significant performance increases compared to their normal counterparts regarding the usage of OS threads and memory.

2.4.3 Performance of Java Threads and Virtual Threads in Spring Boot

There are some articles covering the difference between normal- and virtual threaded Spring Boot applications. However, these articles should be approached critically since it's not peer-reviewed work and the given methodology has a lot of missing details.

Ghanshyam Verma's article "Virtual thread: performance gain for microservices" [19] displays insight into how virtual threads perform in a Spring environment. A traditional threaded Spring application, and a virtual threaded application was developed. Three tests were done: a thread sleep on one second, one endpoint call, one slept database call. On the first test, traditional threaded application could handle up to 1000 concurrent users before starting to falter with increased latency per request. The virtual threaded application could still handle 2000 concurrent users without

any kind of delays. The second test proved similar results. The third test proved that the virtual threaded application performed worse.

In the medium.com article “Project Loom with Spring Boot: performance tests” by Aleksandr Filichkin [20], four prototypes were created: A normal Spring MVC, Virtual-threaded Spring Boot, Reactive WebFlux, and virtual-threaded Reactive WebFlux. JMeter and VisualVM are used to perform load tests on various numbers of users. From the preliminary results by Filichkin, Virtual-threaded Spring Boot application could at maximum hold, 4 000 users, and then failed. The conclusion given by the author is that virtual threaded Spring Boot can handle the same number of concurrent requests as Reactive WebFlux if it weren’t for Apache Tomcat not being designed to handle several thousands of concurrent users.

2.4.4 Summarizing the Relevance of Previous Works

Performance measurements of Java and Spring framework are plentiful. Studies on the performance differences between Reactive WebFlux and Spring Boot show that the reactive framework often outperforms the non-reactive Java framework.

With the preview release of Java virtual threads in Java 19, performance of virtual threads has been studied against Java’s existing Thread API in a web environment. In both studies where normal threads were compared against virtual threads, multiple advantages were displayed.

Although studies regarding the performance of virtual threads and Java frameworks are plentiful, no peer-reviewed study has been done on measuring performance on a Java Spring application that utilizes virtual threads, especially compared to Reactive WebFlux. Articles have been published on this topic, such as Ghanshyam Verma and Aleksandr Filichkin’s article. As they are not peer-reviewed papers and lack sufficient information for recreating their results, knowledge can be gained on the subject.

2.5 Measuring Performance

How the performance of a web service is measured is important. The following model for measuring the performance of a system is shown in figure 2.1 from the book Systems Performance, 2nd Edition by Brendan Gregg [21]. The system that is being measured is given a workload, and then latency, throughput, CPU and memory usage is measured which in turn is the resulting performance. Interference in a system under test is any variables that can affect the resulting performance. Careful analysis of what could potentially interfere with the system under test is important to ensure that the resulting performance is not skewed.

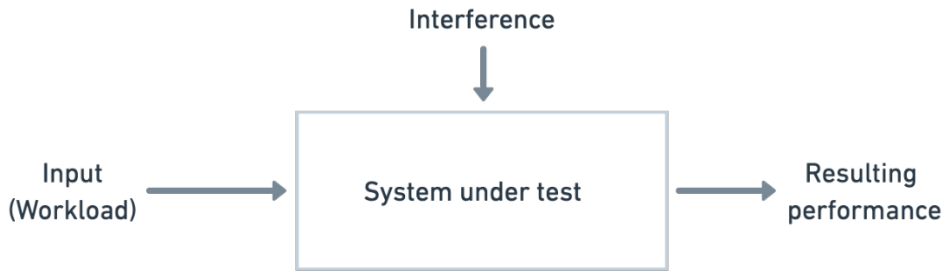

Figure 2.1: Block diagram of a system under test, adapted from Systems Performance, 2nd edition [21]

2.5.1 What to Measure

Gregg mentions the top metrics for web-application environments [21]. One of the mentioned metrics is latency, which is a measure of how long it takes to complete a workload. Latency can be a measurement in multiple scenarios, such as a disk operation or a database query. For web services, it can be a measurement of how fast a website loads. This measurement is important, since it is crucial for the end user experience.

Another key metric is throughput, which is the rate at which work is done. Measuring throughput is key when analyzing the scalability of a system. When comparing the load with throughput, a relationship can be seen. When throughput stops scaling linearly with load the system is often utilizing 100% of its available resources, which is a point when the application cannot do any more work.

Additionally, an important measurement mentioned is resource utilization. It is the measurement of how the system utilizes its available resources, and it is relevant when analyzing throughput. For web-applications, CPU- and memory usage is often examined since they are primary resources required for an application's optimal performance. Several previous works also point out when measuring internal resource utilization. As such, resource utilization is a key metric in evaluating the performance of a web service.

2.5.2 Avoiding Bottlenecks and Influences from Non-relevant sources

For any given performance measurement of a system, it is crucial to realize where the measurement might fail. Brendan Gregg [21] provides a checklist on where a performance measurement might fail, and what to do to prevent these failures. Gregg explains that performance measuring can be a very useful tool if done correctly but could also be misleading if not done carefully. For instance, all tests should be performed with a significant warm up time to ensure that the system is in a ready state when the test is performed. It is also important that the test conditions should stay consistent between tests since a change in system configuration or testing variables can skew the results. Lastly, it is preferred that workloads need to accurately simulate real world conditions to prevent inaccurate conclusions on performance. This is important when performance testing an end user product, but as in the case for this thesis, it is less important.

2.6 Monitoring tools

This section will describe tools used when performing performance testing, such as tools for health monitoring, performance measurement and application isolation. Application isolation will introduce the possible measures that can be taken to reduce the interference from external sources. Health monitoring will introduce the possible tools that can be used to monitor resource utilization of Java applications. Measuring external performance of web frameworks will introduce possible tools to benchmark web services.

2.6.1 Application Isolation

As Gregg mentions on measuring system performance, interference is the influence of external sources that can affect the produced result [21]. Interference could come from processes that run on the same machine as the process that is being measured. Avoiding this type of interference can be done by isolating the application with its own separate execution environment.

Application isolation has been used to some degree to isolate the application from external hardware, or to limit CPU and memory available to the application. In André Nordlund and Niklas Nordströms work [14], containerization platform Docker is used to isolate the application. Docker Desktop – a UI application over Docker [22]– was further utilized by Sebastian Iwanowski and Grzegorz Kozieł [15] by setting a limit on CPU and memory resources available for their test cases.

There are also some works that do not use any kind of containerization. Beronić, Modrić, Mihaljević and Radovan [18] [17] does not mention any kind of application isolation being done in any of their research papers. In Marek Pucek, Michał Błaszczyk, and Piotr Kopniak's study "Comparison of lightweight frameworks for Java by analyzing proprietary web services" [23], bare metal is also utilized when comparing the performance of several Java frameworks. This proves that both bare metal and using Docker are valid alternatives.

An alternative for application isolation is running the application inside a virtual machine. A virtual machine is an emulation of a physical machine, and multiple virtual machines can run on the same physical machine [24]. The difference between using a virtual machine and a docker container is that a docker only virtualizes the operating system, whereas the virtual machine simulates the physical hardware.

All three methods provide some kind of application isolation. Running the tests on bare metal could have affected some of the results since there could be background OS tasks that are not relevant that could be non-controllable variables. Docker's solution on application isolation still involves running small operative systems, [25] but without the overhead of background OS tasks as in running on a virtual machine. However, Docker is still determined by the existing hardware. An additional layer between the application and the operating system should also still be seen as a possible overhead.

2.6.2 Health Monitoring

Health monitoring is the process of measuring the hardware resources a process, application or any kind of program is using. Several tools are mentioned in previous works to monitor the CPU and heap usage of a Spring Boot application. VisualVM – a health monitoring app for JVM applications – is used in Nordlund and Nordstöm's test case [14] VisualVM as an internal health monitoring application in his degree project. Beronić, Modrić, Mihaljević and Radovan also mention the use of VisualVM in their research paper regarding the comparison of Kotlin Coroutines and Virtual Threads.

Another set of tools that can be used to monitor the CPU and heap usage is Spring Boot Actuator, Prometheus, and Grafana. Sebastian Iwanowski and Grzegorz Kozieł used these tools to measure their study case [15]**.** Błaszczyk, and Kopniak's study also mention another set of tools when measuring framework resource usage by using VisualVM, Prometheus, and Grafana [23].

Health monitoring with Spring Boot actuator has a drawback in that it requires Prometheus for scraping data, and Grafana for dashboarding the scraped data. This is a significant overhead compared to VisualVM. However, Spring Boot's actuator records additional data that VisualVM does not. [26] This could show to be beneficial to gather additional information such as types of memory that is being used, number of threads being run and peaks.

2.6.3 Measuring External Performance of Web Frameworks

There are multiple tools to measure the latency and requests per second of web services. Previous works refers to multiple options of high performant software used to examine latency, requests per second, based on configuration given by the user.

André Nordlund and Niklas Nordström used JMeter in their performance comparisons between WebFlux and Spring Boot. Apache JMeter is an open-source application for load testing web services primarily through the means of HTTP. [27] The tool is also favorably used in Marek Pucek, Michał Błaszczyk, Piotr Kopniak's study regarding evaluation of web frameworks.

There are alternatives to JMeter. In "Reactive Programming with RxJava" by authors Tomasz Nurkiewicz and Ben Christensen [3], Wrk is chosen as the benchmarking tool because of its ability to simulate thousands of concurrent users without becoming a bottleneck like traditionally threaded applications such as JMeter.

Vegeta, which is used by Pufek, Beronić, Mihaljević, and Radovan, is also an alternative HTTP benchmarking tool using the Go language for measuring metrics such as latency, and throughput. Vegeta provides extra information compared to Wrk [28] [29] with results displaying different percentiles of latencies, when the test started and ended, requests per second, and throughput. One advantage that Vegeta has over Wrk is that it has been proven in multiple academic studies that it can handle thousands of requests per second. Another is its ability to output benchmarking results into different formats, such as JSON, histograms, and graphs.

3 Methodology

This chapter presents an overview of the methodology for this study including the necessary information and arguments about the test case, development of prototypes, the testing environment, and the evaluation process will be discussed. Section 3.1 will describe literature study and the approach to gaining knowledge on the subject. Section 3.2 will discuss the role of the prototypes that will be developed, and section 3.3, about the technical details and arguments on why specific tools were chosen. Section 3.4 will discuss the details about the testing software that will be used to conduct the test, as well as the specifics about the environment the prototypes will be tested in. Section 3.5 will provide information on how the test data will be examined, and the limitations of the current test case.

3.1 Literature Study

To gain insight into current research and knowledge relating to the problem statement in chapter 1.2 a systemic literature review was done. This process consisted of two parts. Firstly, current knowledge and research was surveyed relating to Java virtual threads. The survey consisted of utilizing database search engines such as Google Scholar, IEEE Xplore, and KTH Primo with keywords such as “Virtual threads”, “Performance of virtual threads” and “Java virtual threads”. This was done to gain an understanding of the subject, and to identify possible gaps in research relating to the performance of virtual threads in web services.

Secondly, the same approach was applied to gain knowledge on methods for performance testing web services. This part consisted of analyzing and reviewing methodology for performance testing web services in related works and reviewing literature on system performance analysis and benchmarking. The purpose of reviewing related works was to provide a basis on how performance testing of web services previously has been done, the results of each study, the methodology used and their respective conclusions. Also relevant for the methodology of this study was reviewing literature on system performance, and how to benchmark applications. This was done to provide knowledge on known pitfalls when conducting application benchmarking, and how to avoid them.

3.2 Test Case

One of the features of the platform Stryda provides is a clan feature. Clans are a type of group that users on the platform can join, which creates a small community of players which can interact with other users. The existing Clan service does not provide user information but holds references to user identification that a consumer can use to call the user data service. Since the front-end web application wants only to perform one call, the data has to be aggregated in a service in between the front-end and the back-end. The current aggregation service uses Reactive WebFlux. An example of the process on how data is aggregated can be seen in figure 3.1.

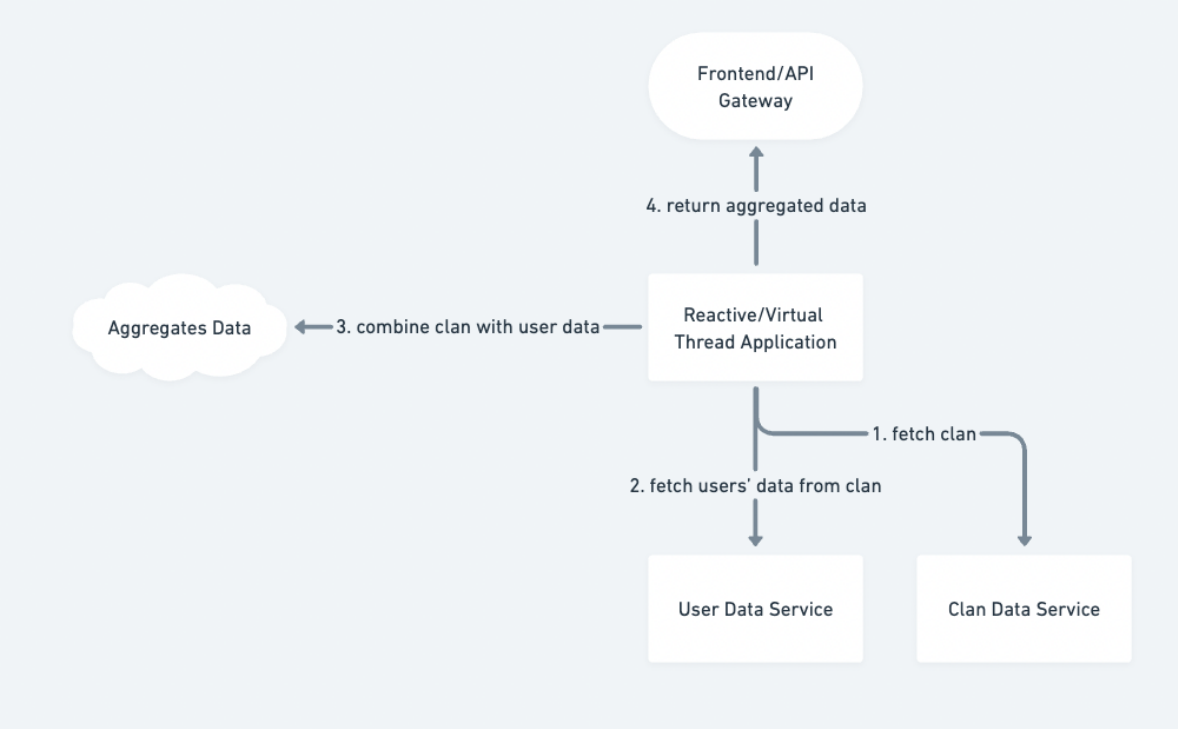

Figure 3.1: Step-by-step example of the Clan API service aggregation endpoint of a get request.

The programming style between the gateways (services that use API composition) and the underlying services with business logic is different, one being reactive and the other being imperative. The difference causes confusion between the developers as well as a large learning curve for new employees at the company.

The test case for this study consists of analyzing the performance of each framework in a situation where the microservice pattern API composition is utilized. The developed prototypes act like the given API gateway responsible for API composition as in figure 3.1. The prototypes will be subjected to a load test, and then the resulting performance will be measured using monitoring software.

3.2.1 Arguments for Using Data Aggregation as a Test Case

There are several key arguments that could be provided on data aggregation being chosen. Although not truly peer reviewed, Ghanshyam Verma's experiment indicated that database operations struggled with virtual threads, which would pose challenges when comparing to Reactive WebFlux. Since virtual threads performed quite well with calling remote endpoints, data aggregation was first seen as a suitable case to compare virtual threads against Reactive WebFlux. It is also important to note that performance comparisons involving databases can be complex. For example, Karl Dahlin's study focused on comparing the database libraries R2DBC and JPA, which differ significantly from the focus of this study.

As highlighted in chapter 2.2, it has been brought up by some that Reactive WebFlux is a high-performance alternative to the traditional imperative programming style, in which the latter relies on creating threads for each request.

If virtual thread-powered web services are shown to perform similarly or even outperform Reactive WebFlux, it may prompt further research into which use cases are better for each approach, as well as if there is now a reason for still using reactive programming in Spring framework-powered applications.

3.2.2 Metrics to Test

Latency was the first chosen key important metric. As noted in chapter 2.5.1, the delay when a service responds to user requests determines whether the user will continue to engage with the application or seek alternatives. Therefore, having low latency in a web service is crucial.

Throughput was selected as another important factor due to the metric representing the rate at which a service can handle workloads and process requests. As a service reaches its threshold, it fails to match the number of requests it can process according to the number of requests it receives, resulting in a decrease in overall service quality and user experience. The metric was therefore seen as an essential factor to evaluate to measure a web service's performance.

As for internal performance metrics, CPU and memory usage as primary metric. These primary resources are required to run a web service. Nowadays running on the cloud means the cost of a service scales with the rate of CPU and memory usage [30]. This means that a service that utilizes less CPU and memory is more cost-effective as well as having the possibility of being more environmentally friendly.

3.3 Development of the Prototypes

This chapter will describe the implementation process for each prototype. First, the base design of what the prototypes will do will be introduced. Also, the base code library that will be used on all prototypes will be introduced with arguments on why it's being used. Common tools such as HTTP client will also be discussed and chosen. Finally, the prototype implementation for normal threaded, virtual threaded, and reactive will be described.

3.3.1 Base Design

The objective of the prototypes was to imitate the clan aggregation service, with a specific focus on the HTTP endpoint that served the aggregated information to the benchmarking tool due to the aggregation of clan and user being the primary focus of such service. For the prototypes, two designs were considered: either to separate the two endpoints into their own underlying services or to include them both in each prototype.

Even though the latter strategy would have been more realistic, it would have required extensive testing on different frameworks to find one that would perform better than the actual prototypes to prevent the underlying services from becoming bottlenecks that would affect the results. Due to time constraints and desire to reduce the number of uncontrollable variables, it was decided that each service part would be simulated via a stub endpoint that existed within the prototypes.

There were concerns regarding the accuracy and realism of incorporating each endpoint into the prototype, even though this method was more efficient in terms of hardware usage. Particularly, the prototype's inclusion of both stub endpoints would have meant that a single request would technically be regarded as multiple requests, which could influence the results. It was still deemed an acceptable compromise considering the practical constraints.

As seen in figure 3.2, the benchmarking tool calls an /aggregated endpoint which in turn calls stub endpoints /clans and /users. The proper implementation of these stub endpoints can be seen in listing A.1 in appendix A.

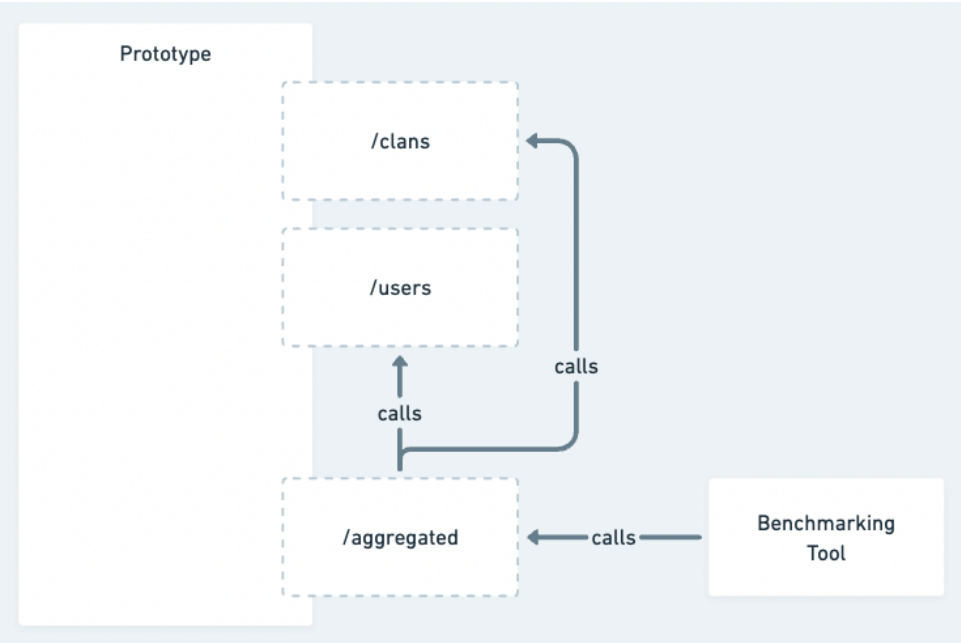

Figure 3.2: Step-by-step example of the prototype process.

3.3.2 Base Code Library

With the information brought up at 2.5.2 to avoid factors that could be problematic to the results, a decision to utilize a base code library, a set of methods and classes whose purpose was to be shared between the prototypes, was determined due to the argument that reducing of potential for variations in the code would produce a much more focused result. This concern could be highlighted in Aleksandr Filichkin's article, where the code for imperative- and reactive services differed, and it was unclear whether the differences in methods affected the results.

To develop the base code library, critical key components that would be shared across the prototypes were established. First was the data format used by the clan data service and user data service. Secondly, the format of the aggregated data was selected for sharing. Thirdly, a custom written object mapper was decided to be included in the base code library to facilitate the aggregation of data.

A stub was also integrated into the prototypes to generate a clan and a list of users, which would be exposed to the endpoints that could be called by each prototype recursively, with the list size for the clans and its members, as well as the sleep timer,

being controllable parameters that could be adjusted. The implementation of the method for building a clan can be seen in listing A.1 in appendix A.

As seen in figure 3.3, the base library provides all the required stub data and the method to aggregate.

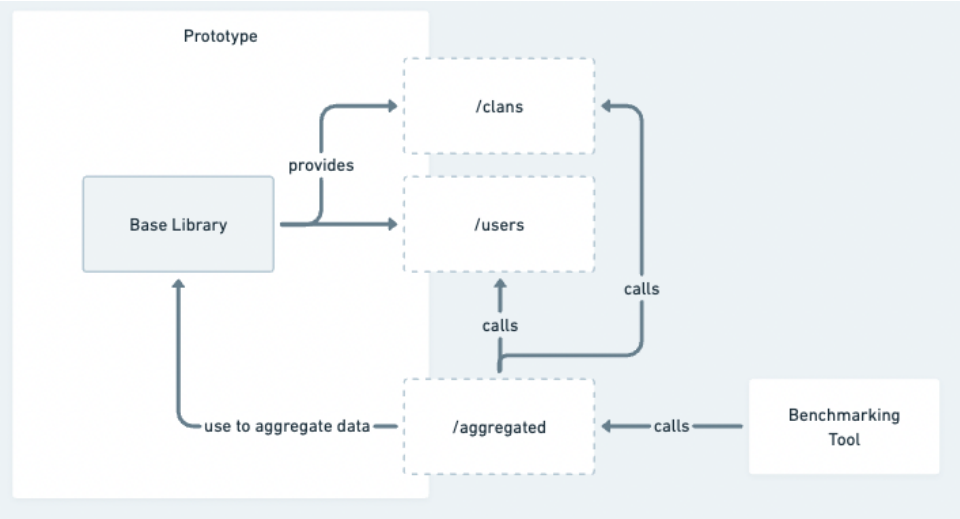

Figure 3.3: Step-by-step example of the prototype process with the role of base library

3.3.3 HTTP Clients

To ensure that the results of the study were not affected by differences in HTTP client libraries, the selection of an appropriate client was crucial. Several options were considered with their own advantages and disadvantages.

Java's HttpClient [31], introduced in JDK 11, was one possibility due to its ability to perform synchronous and asynchronous HTTP requests, and being very compelling for being a part of Java's standard library.

Apache's HttpComponents library also offers an HTTP client class [32] that, according to Apache, was developed due to Java's standard HttpClient lacking certain features. The main advantage of using Apache's client would be that it supports a reactive style, which could be proven to be useful for the study.

Spring WebClient [33], built on top of Reactor Netty's HTTP client, was the third suitable option considered. This client supports both synchronous and reactive programming styles, making it friendly to Reactive WebFlux and Spring Boot applications.

Spring WebClient was chosen as the most suitable HTTP client for the study due to its compatibility with both synchronous and reactive programming styles and its strong function inside the Spring environment. This ensured that the HTTP client library did not become the focus of the study, but rather that the results reflected the virtual thread and reactive designs being tested.

3.3.4 Reactive WebFlux

The Reactive WebFlux prototype utilized the base code library to implement the stub endpoints to be called. As for the inner implementation of Reactive WebFlux, it was achieved using the publisher subscriber pattern utilizing Flux and Mono wrappers [34], which allowed the code to respond to events when the work was completed. As described in 2.3.3, due to clans list residing within the Flux wrapper, each invocation was executed in parallel and returned upon completion of each aggregation.

As described in chapter 2.3.4, It is worth noting that Reactive WebFlux is compatible with both Tomcat and Netty web servers, but Tomcat was selected for this study as Spring Boot only supports Tomcat. The usage of different servers had the potential to produce varying results, hence standardization was deemed necessary to avoid this issue.

3.3.5 Non-virtual Threaded Spring Boot

In order to achieve parallelization that is possible in Reactive WebFlux with its Flux publisher, it was necessary to implement a similar asynchronous pattern, in order to not produce different results. This was accomplished using the Future and ExecutorService classes [35] [36] which allowed parallelization and simulated the way Reactive WebFlux was implemented. As seen in figure 3.4, the approach involved placing a list of asynchronous tasks in an array and returning the list once all tasks were done with the cachedThreadPool method being utilized to execute parallel tasks when fetching users for each clan.

```
Input:
  - type of executor (virtual thread, cached thread pool)
  - list of items
  - function to execute on each item

Tasks = list of async tasks

For every item in list
    Create a task that executes the function on the item
    Add the task to the list of tasks

For every task in the list of tasks
    Wait for the task to complete
Return the modified list of items
```

Figure 3.4: Pseudocode of asynchronous list operations.

3.3.6 Virtual Threaded Spring Boot

The distinction between a virtual-threaded Spring Boot application and a non-virtual-threaded Spring Boot application is simply configuration class. As stated by the Spring Boot team, minimal configuration is required to enable virtual threads on an existing Spring Boot service [37]. The configuration class can be viewed in listing A.1 appendix A.

After implementing the necessary configuration, Spring Boot could utilize virtual threads for request processing. To create threads when fetching users for each clan in parallel, a parallel executor that creates virtual threads was employed.

3.4 Testing Environment

This section is going to introduce the chosen technologies for application isolation, health monitoring, and benchmarking tools, as well as argue on why they were chosen.

3.4.1 Application Isolation

Chapter 2 presented three alternatives for isolating applications to run tests on: Virtual machine, Docker, and bare metal. After considering various factors, Docker was chosen as the preferred option. One of the primary reasons for this decision was that according to IBM, Docker provides a consistent environment for different hardware and operating systems [24]. This means that it is easier to reproduce the tests and achieve consistent results. Furthermore, previous studies, such as the study conducted by André Nordlund, Niklas Nordström, Sebastian Iwanowski, and Grzegorz Kozieł, have demonstrated that Docker fits well for performance comparisons.

IBM also introduced another advantage of Docker, which is that an isolated environment for the application that is being tested on can be provided to prevent any possible interference from the machine it is running on. In addition, since Docker containers only contain the OS processes and dependencies necessary to execute the application, rather than an entire OS instance, making it a more lightweight solution than a virtual machine. Docker was hence deemed to be the most appropriate software for application isolation as it provides application isolation without any unnecessary load on the physical machine.

To facilitate the setup and teardown of collections of applications, such as the underlying microservices and sidecar applications used for collecting internal health information, docker-compose was utilized [38].

3.4.2 Health Monitoring

In previous studies, VisualVM was frequently used as an internal health monitoring tool to collect information about an application's CPU- and heap usage. While VisualVM is a popular tool in studying health metrics, Spring Boot Actuator was selected for this study as it offers the ability to analyze additional data [26]. However, Spring Boot Actuator only exposes an endpoint that can be scraped via HTTP request which means that external scraping tools must be utilized to collect information. Brian Brazil, in his book Prometheus: Up & Running [39], explains that Prometheus requires a specific format on the metrics endpoint to be able to interpret the data.

To enable Prometheus to scrape health metrics, the micrometer-registry-prometheus library [40] was used to create a Prometheus-friendly metrics endpoint. With the endpoint in place, Prometheus could effortlessly gather health information. While Prometheus provided alerts and basic graphs, it lacked a graphical dashboard capability. Brian Brazil also suggest that Grafana can be quickly set up with docker and can utilize Prometheus as a data source. This could then be used to create various

types of dashboards containing time-series data for each metric with the capability of exporting the scraped data into a time-series CSV file to be used for analysis. An example of how it was set up can be seen in figure 3.5.

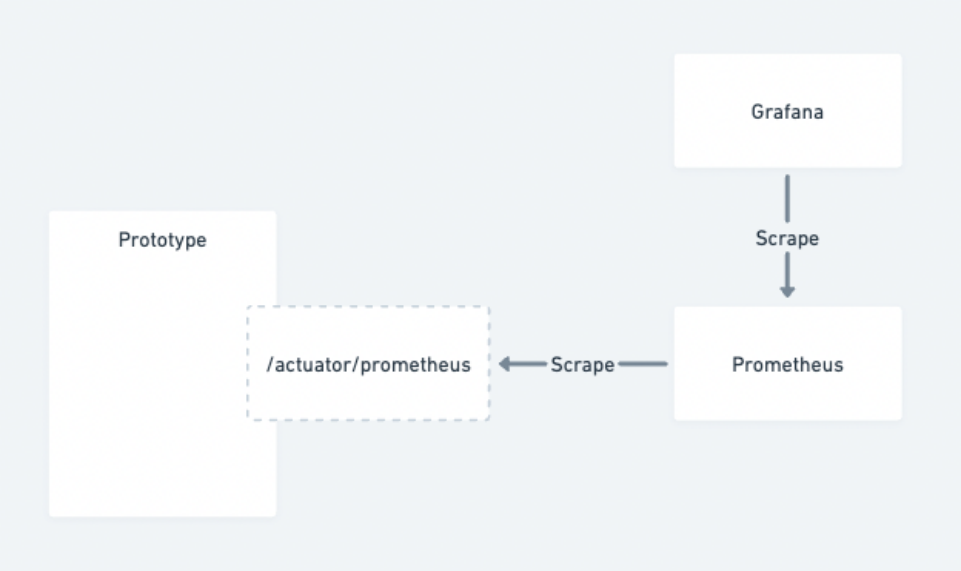

Figure 3.5: Process of Prometheus scraping the actuator endpoint and Grafana getting information from Prometheus.

In conclusion, Spring Boot Actuator was preferred over VisualVM due to its ability to provide more comprehensive information, Prometheus was utilized for scraping health information, and Grafana was chosen for its capacity to display dashboards of the scraped data. Listing B.1 in appendix B has the configuration for setting up scraping configuration of a spring boot actuator enabled web service.

3.4.3 Benchmarking Tool

Since it is referenced by Tomasz Nurkiewicz and Ben Christensen that JMeter is suffering from performance issues when it comes to simulating thousands of users due to its reliance on threads, it was out of the scope for this study. Wrk and Vegeta are two other mentioned tools. Vegeta was chosen over wrk due to its ability to provide information about latencies, throughput, and response timings. It also supports different formats for producing attack reports, making it easy to export the results for analysis. It has also been tested and proven to work with up to several thousand concurrent requests without any hindrances, which is more than enough for producing load to the prototypes.

3.5 Testing Methodology

The test process was done per prototype. Before running the prototypes, the tunable parameters were set in the code. Thereafter, Docker Compose was used to set up the prototype, Prometheus, and Grafana. As discussed previously in chapter 2, a warm-up was done via Vegeta that called the testing endpoint for 30 seconds, one request per second. After the warm-up was completed, the actual attack was performed.

3.5.1 Testing Parameters

For each prototype version, multiple tests were performed, and three times for each test to check that the values were consistent for each test. Two tunable parameters were used, requests per second and artificial delay from endpoints that simulate the underlying services.

There was the mention of payload being used as a parameter in chapter 1.2, but it was exempted due to the time constraints. It was also targeted as non-valuable since it was not considered to provide valuable information about a prototype's behavior. However, since the stub required clan count and member count, it was both set to five clans and five members.

The delay on each request was an artificial time for the stubs to serve each request. In a real-life scenario, this delay would vary depending on the required work the service must do as the work could consist of performing database queries or requesting data from other services. Therefore, it was considered valuable to perform tests with different delays. With consultation from engineers at Stryda, 100- and 500 milliseconds of delay were chosen as test values due to it being typical delay at which their services take for any given request.

The lower limit of 10 requests per second was selected as a representative sample of the number of requests a typical web-service at Stryda receives during peak hours. The number of requests per second was then increased in steps of 25 requests per second until the prototype had a system failure or became unresponsive. Doing gradual steps until unresponsiveness was the chosen process as it could display the potential strengths and weaknesses of each prototype at each stage.

3.5.2 Testing Hardware

The hardware that the prototypes were run on was an Intel-chip based MacBook Pro running on Quad-Core Intel Core i5 2.4GHz with 16 gigabytes of memory, since it was provided by the company. This specific operating system was chosen due to it providing UNIX-based syntax making it easy to install software such as Vegeta, Docker, etc.

3.5.3 Collecting Test Results

After running a test, the result was extracted from Vegeta's results to a JSON file and then evaluated. Through the start- and end time provided by Vegeta, the internal CPU- and memory usage was taken at each time interval, which was then aggregated to give values such as minimum, average, and max usage of each metric. To collect the internal health metrics, Grafana provided an export of a time series CSV file which would row each resource utilization during 1 second delay between each row.

4 Results

This chapter will present the statistical results obtained from benchmarking the three prototypes on their CPU and memory utilization, as well as their average latency and throughput. As previously described in Chapter 3.5, the number of requests per second for each fixed delay was increased until all three prototypes failed. For each prototype, a grouped bar plot was generated, with each group representing a specific number of requests per second used for benchmarking. The mean value of CPU and memory utilization is calculated from 120 values from each case of requests per second in all cases except for the maximum number of requests where the prototypes were unresponsive. In those cases, the number of values used in calculating the mean was significantly lower since Grafana only received data when the prototype was responsive. The virtual prototype is represented by the blue bar, the reactive prototype by the red bar, and the normal threaded prototype by the green bar. The black line in each bar is the confidence interval with a confidence level of 95 percent.

4.1 CPU Usage

In Figure 4.1, the average CPU usage of the prototypes is presented in relation to the number of requests per second when the fixed delay was set to 100 milliseconds. The normal prototype stopped responding around 100 requests per second, the reactive at 600, and the virtual at 700 requests per second.

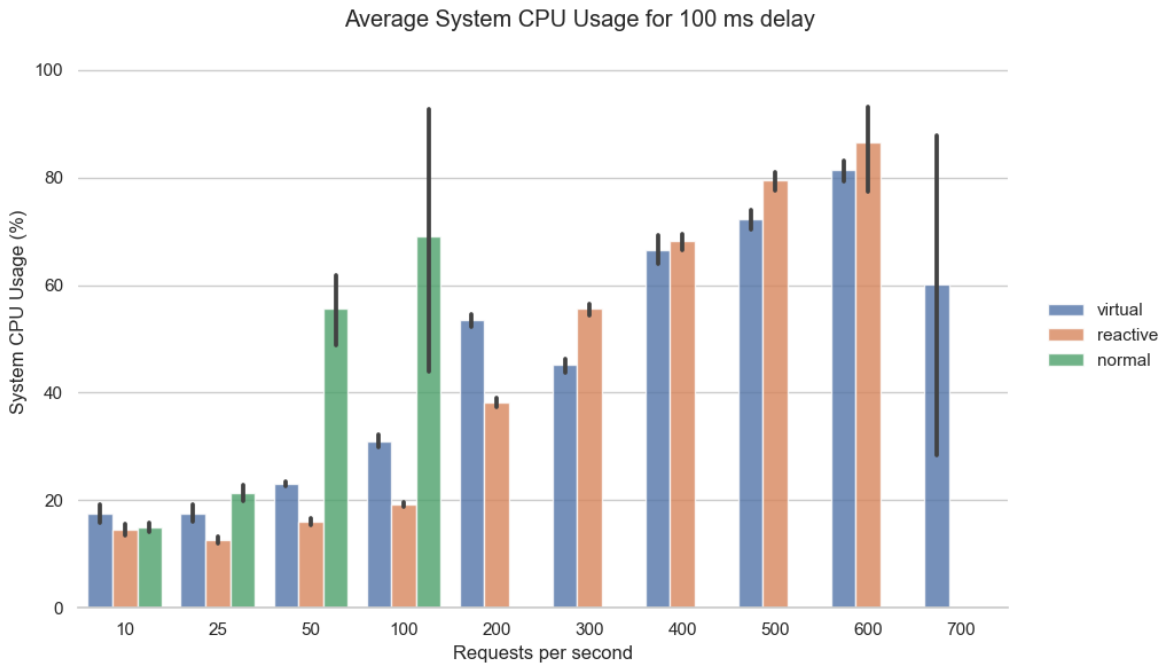

Figure 4.1 Average CPU usage for each prototype, during 10 to 700 requests per second for 100 milliseconds of delay on the underlying services. Lower is better. X-axis represents the number of requests that were done for the prototype represented in the grouped bar.

Figure 4.2 displays the CPU usage for the prototypes when the fixed delay was set to 500 milliseconds. The normal prototype stopped responding around 100 requests per second, the reactive at 225, and the virtual at 250 requests per second.

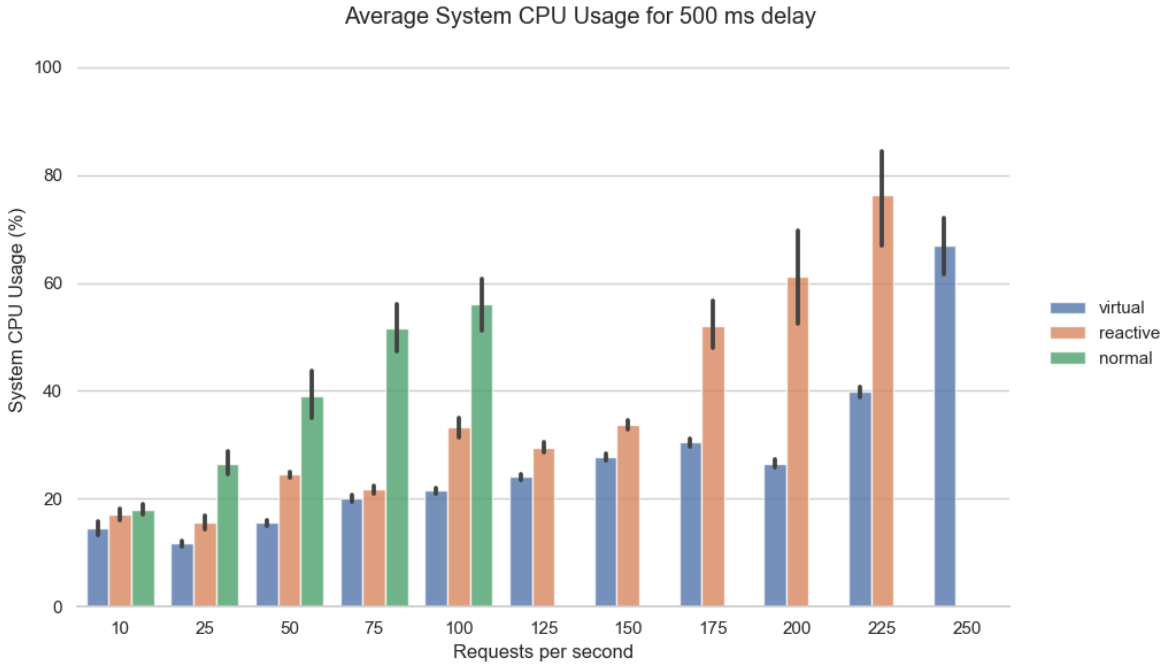

Figure 4.2: Average CPU usage for each prototype, during 10 to 250 requests per second with 500 millisecond delay for the underlying service. Lower is better. X-axis represents the number of requests that were done for the prototype represented in the grouped bar.

4.2 Memory Usage

Figure 4.3 illustrates the memory usage of the prototypes at 100 milliseconds of delay. The total number of measurements in this figure is the same as figure 4.1, due to both graphs coming from the same source.

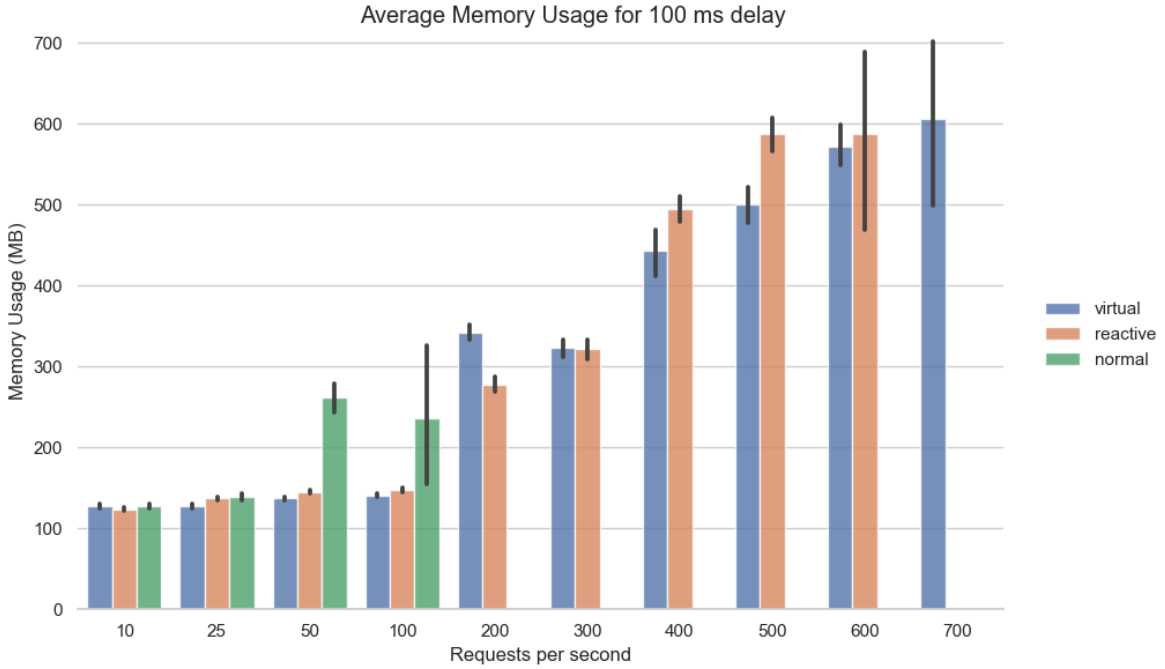

Figure 4.3: Average memory usage in megabytes for each test case and prototype at 100 milliseconds of delay. Lower is better. X-axis represents the number of requests that were done for the prototype represented in the grouped bar.

Figure 4.4 illustrates the memory usage of the prototypes at 500 milliseconds of delay.

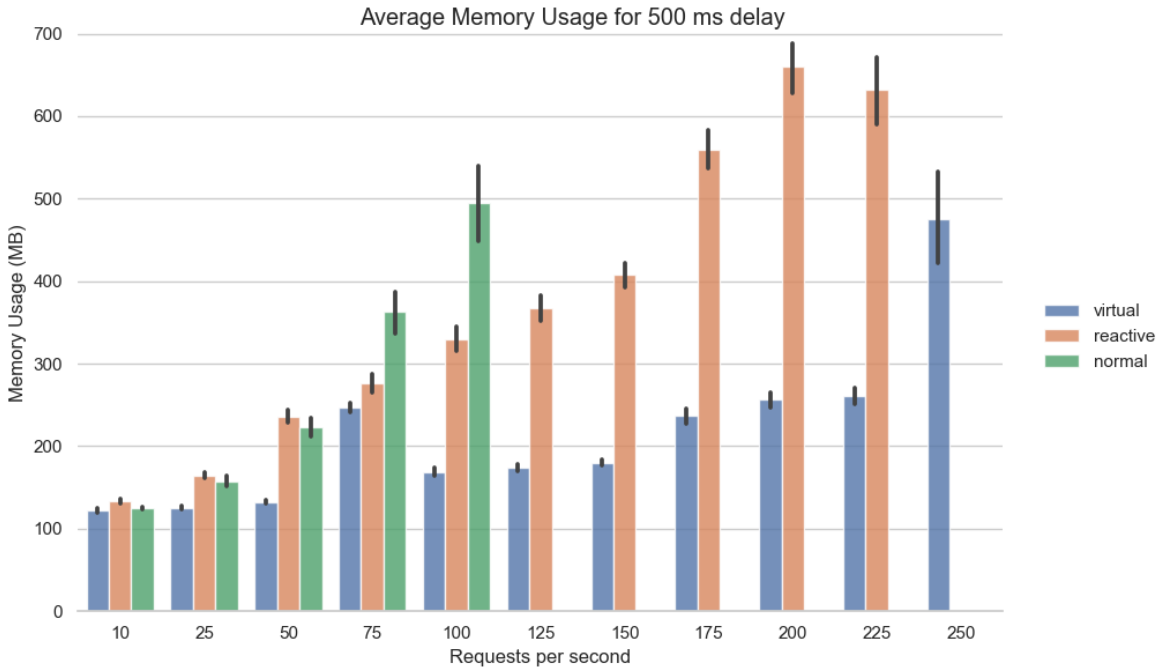

Figure 4.4: Average memory usage in megabytes for each test case and prototype at 500 milliseconds of delay. Lower is better. X-axis represents the number of requests that were done for the prototype represented in the grouped bar.

4.3 Latency

Figure 4.5, shown below, illustrates the average latency measurements for each prototype at various intervals. The total number of measurements for each bar is the number of requests per second times the time taken for each test run. The number of measurements could be different for each prototype due to failed requests not being included.

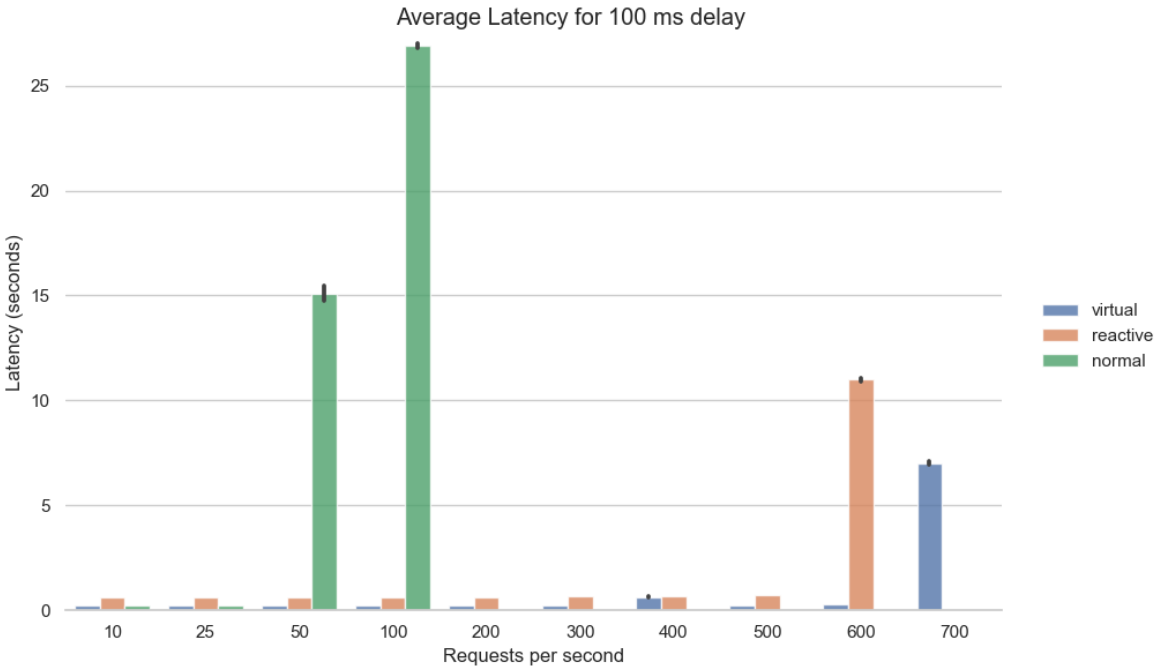

Figure 4.5. Average latency for each test case and prototype is at 100 milliseconds of delay. Lower is better. X-axis represents the number of requests that were done for the prototype represented in the grouped bar. The Y-axis represents the average latency in seconds.

Figure 4.6 depicts as described above the average latency measurements for various intervals, but for 500 milliseconds of delay.

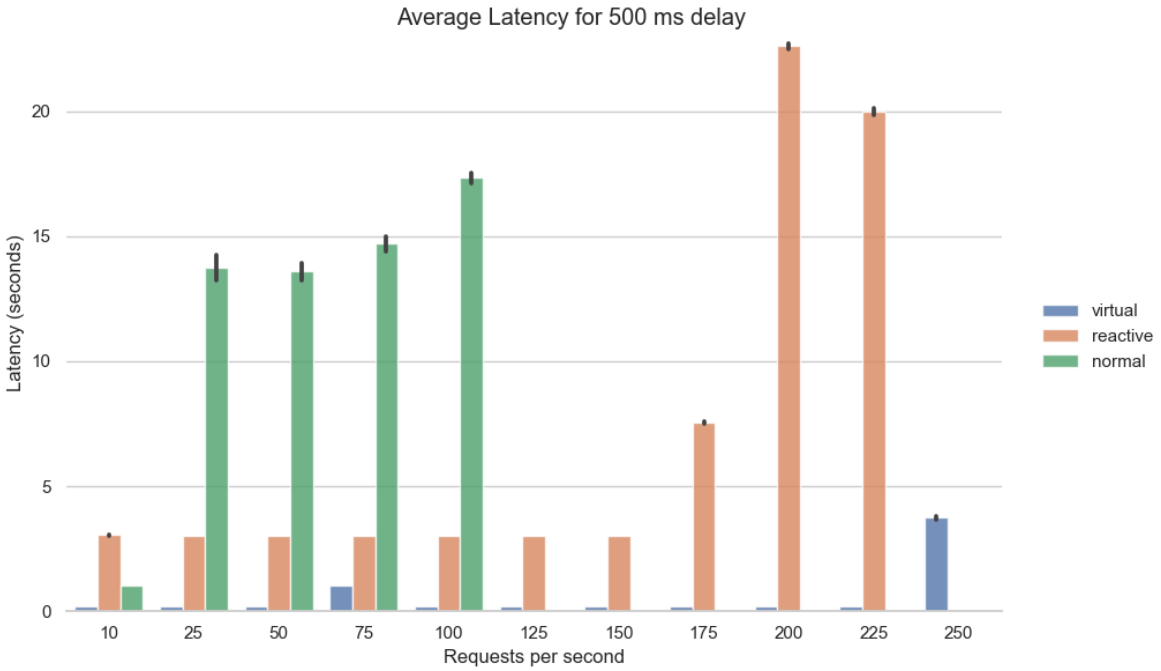

Figure 4.6. Average latency for each test case and prototype is at 500 milliseconds of delay. Lower is better. X-axis represents the number of requests that were done for the prototype represented in the grouped bar. The Y-axis represents the average latency in seconds.

4.4 Throughput

Figure 4.7 illustrates the throughput (requests per second) made for fixed delay of 100 milliseconds. Throughput does not have a confidence interval due to having a single value available from the results given by Vegeta.

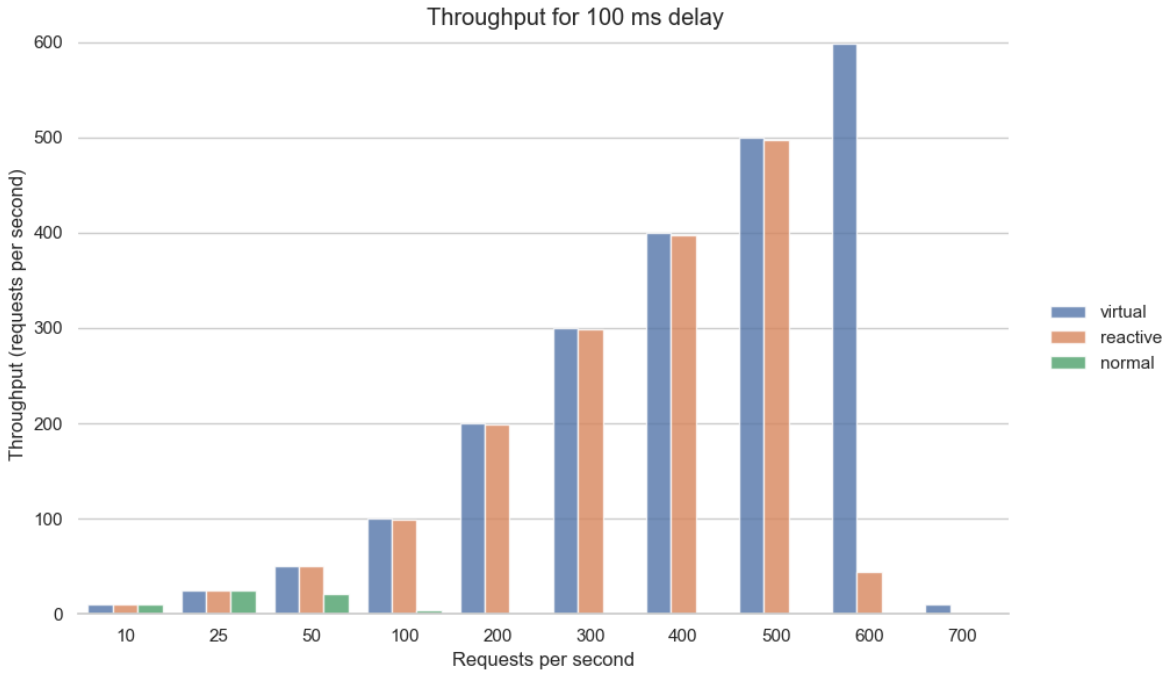

Figure 4.7: Throughput (requests served per second) for each test case and prototype at 100 milliseconds of delay. Higher is better. X-axis represents the number of requests that were done for the prototype represented in the grouped bar.

Figure 4.8 illustrates the throughput (requests per second) made for fixed delay of 500 milliseconds.

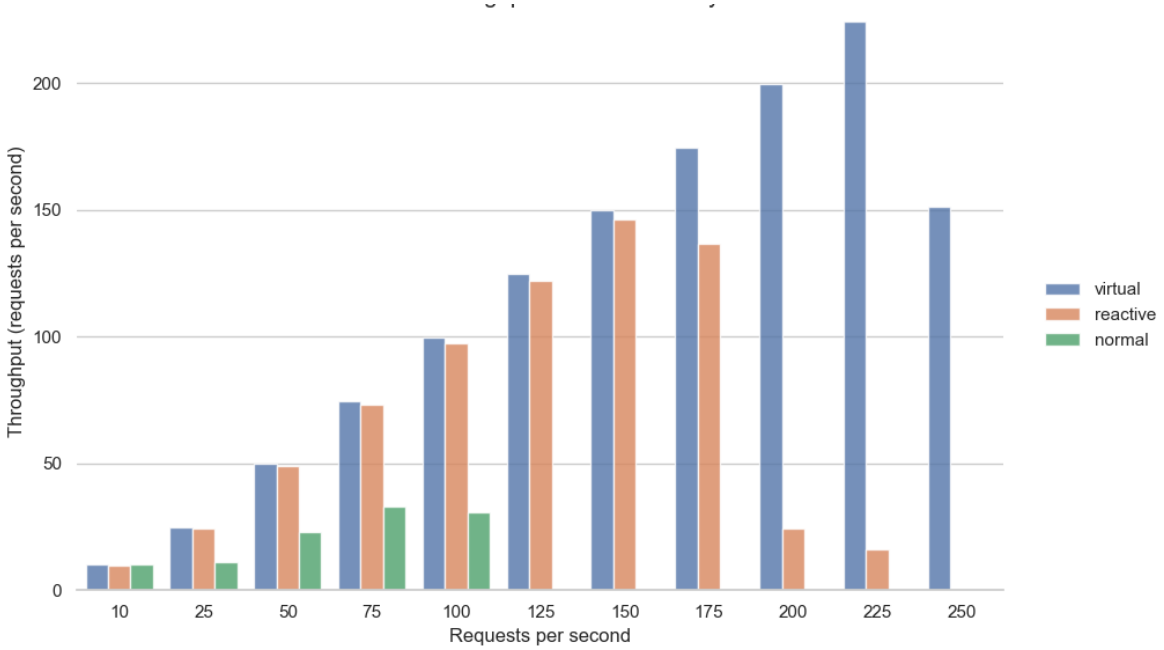

Figure 4.8: Throughput (requests served per second) for each test case and prototype at 500 milliseconds of delay. Higher is better. X-axis represents the number of requests that were done for the prototype represented in the grouped bar.

4.5 Successful Requests

Figure 4.9 describes the percentage of successful requests where the fixed delay was set to 100 milliseconds. A successful request is determined by the response status code 200, which stands for success in REST terminologies. Y-axis represents the percentage of successful requests out of the sent requests.

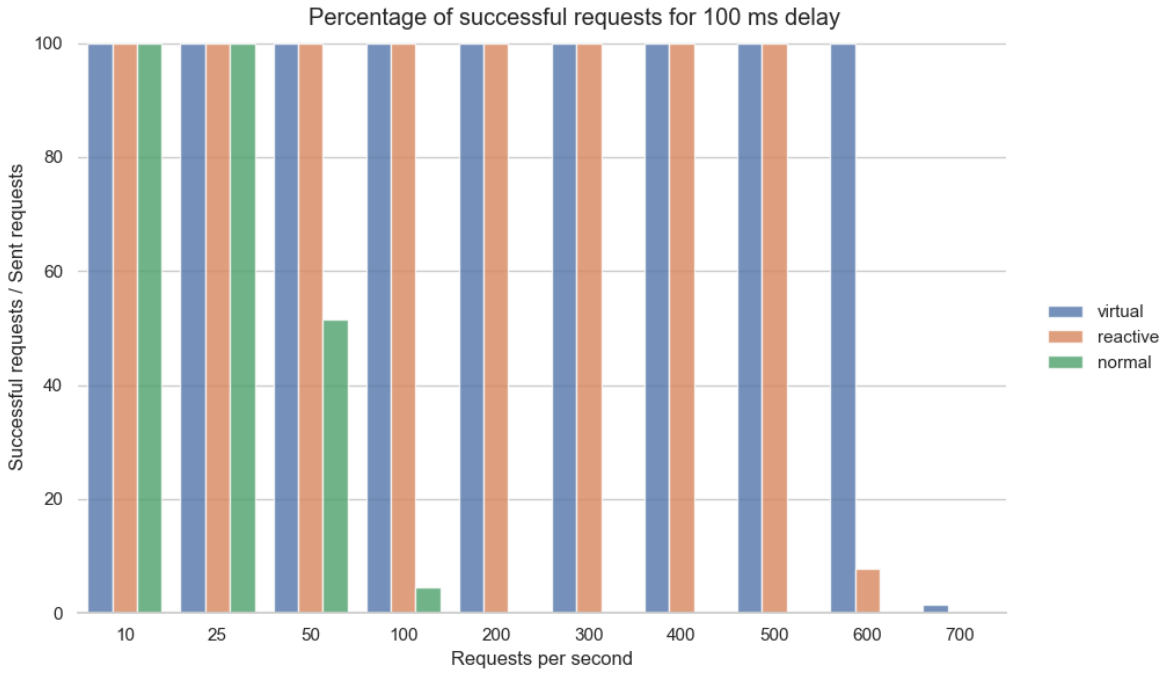

Figure 4.9: Percentage of successful requests for 100 milliseconds of delay. Higher is better. X-axis represents the number of requests that were done for the prototype represented in the grouped bar. Y-axis represents the percentage of successful requests out of the sent requests.

Figure 4.10 describes the percentage of successful requests where the fixed delay was set to 500 milliseconds.

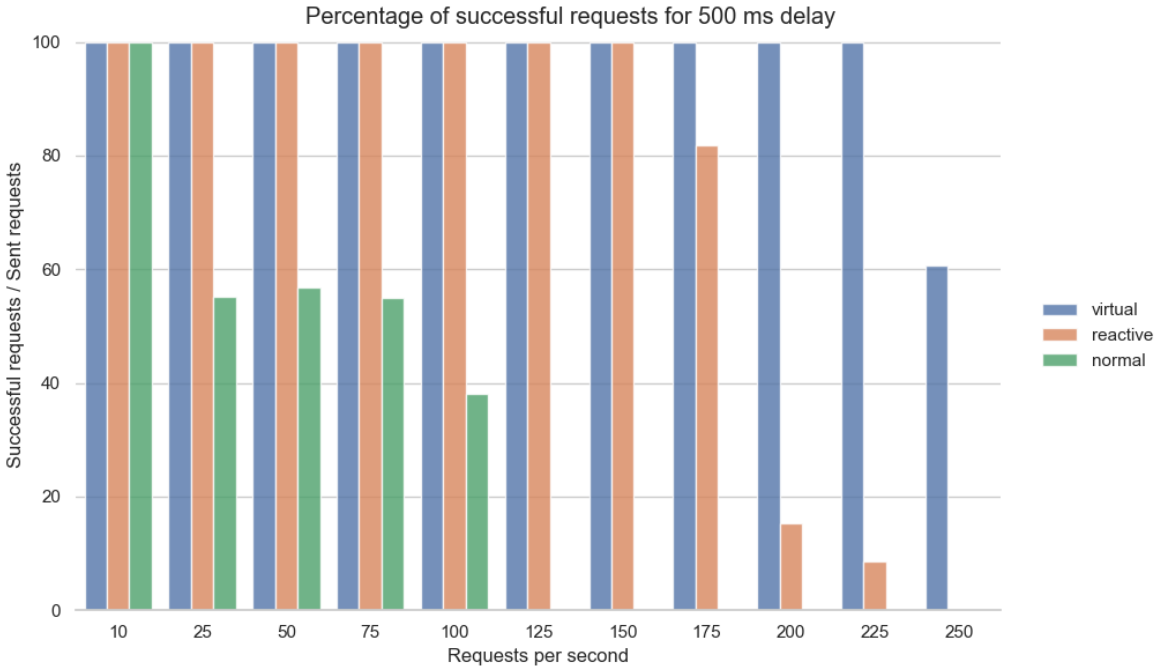

Figure 4.10: Percentage of successful requests for 500 milliseconds of delay. Higher is better. X-axis represents the number of requests that were done for the prototype represented in the grouped bar. Y-axis represents the percentage of successful requests out of the sent requests.

5 Analysis and Discussion

This chapter will focus on analyzing and discussing the performance differences between the two prototypes, as well as exploring the possible sources of error that could affect the results. The impact of the choices that were made in the methodology chapter will also be discussed. In addition to technical factors, economic, societal, ethical, and environmental aspects will be considered and discussed.

5.1 Performance Differences Between the Prototypes

The most common pattern occurring in each prototype from the initial results is the sudden surge of variation in resource utilization before non-responsiveness for the next interval. As seen in CPU-usage from figures 4.1 and 4.2, there is a sudden increased variation in each prototype, although the reactive prototype seems to be more stable.

The results of the study have indicated that the virtual thread prototype performed better than the reactive prototype in most of the scenarios, especially regarding the situation of having 500 milliseconds delay on the stub endpoints. Whereas the reactive prototype no longer responds, the virtual threaded prototype could still serve requests without sacrificing the service quality by keeping up the throughput on the number of requests per second that were made, especially in the category of latency where the reactive prototype performed worse than the normal threaded prototype at the lower range of requests per second.

A possible behavior that could be attributed regarding the reactive prototype's performance could be the result of the method for delaying data, which happens in the data return inside each stub endpoint. When performing artificial delay in the in the Reactive WebFlux library, each return data gets put inside a single-threaded method, which in this study, used normal OS threads [41]. According to Dora Beronić [42], mentions that OS threads utilize more memory than lightweight threads as well has a higher delay for construction and destruction of threads. This could be the reason for the increased memory consumption of the reactive prototype.

Although virtual threads outperformed the other prototypes, there were some questionable results on some of the virtual threaded prototype's performance such as the CPU- and the memory usage spikes, which in subsequent tests dropped instead of continuously increasing. The spikes were also applicable in latency and throughput. At first, it was believed that the spikes existed due to Java's garbage collector running during the test, but the theory was quickly shutdown due to how the behavior could be repeated for each test run. The strange behavior raises some questions regarding how stable the behavior of virtual thread is, due to the feature still being in its preview feature as of Java version 19.

5.2 Possible Sources of Error

Even though possible sources that could affect the test were taken into consideration during the development of the prototypes as well as the test setup, there are still some decisions that could have possibly skewed some parts of the results. The first

discussion point in this section is the overhead from the use of Prometheus and Grafana for health monitoring. The second is the use of WebClient as the HTTP library, and the third is the use of Tomcat of Netty as the web server for Reactive WebFlux. Lastly, the overhead of the abstraction layer of Docker is also discussed.

5.2.1 Overhead of Prometheus and Grafana

The use of Prometheus and Grafana may have had an influence on the results by possibly introducing a slight overhead to the prototypes. Prometheus' call to fetch data from the endpoint provided by spring-boot-actuator has resulted in each prototype being subjected to one extra call per second. Since the exact performance impact of these calls on the system is still unclear, there is a possibility that they may have contributed to deviations in the results. Moreover, the benchmarking tool's utilization of millisecond-level precision for start and end times, small variations in the number of measurements recorded could have happened.

5.2.2 The Use of WebClient

The use of the WebClient library as a HTTP client may have skewed the results by being a factor that introduced a certain degree of overhead, although the extent of this effect is still unclear. While both reactive and imperative programming is supported by the WebClient library, there is still a bit of uncertainties regarding the performance of the library for each programming style. Another uncertainty is the compatibility of WebClient and virtual threads. Given the new nature of virtual threads and their limited adoption in the industry, there is a requirement for further research being done to better understand the compatibility with different libraries.

5.2.3 Using Tomcat over Netty for Reactive WebFlux

As noted in chapter 3, Reactive WebFlux is designed to work optimally on a non-blocking server such as Netty. Since the blocking server Tomcat was chosen to run with Reactive WebFlux, there is a slight possibility of the results being different due to the reactive library running worse than it would do on a non-blocking server.

5.2.4 Possible Overhead of Docker and Hardware

Another possible impact and source of error is that the main overhead introduced by Docker and the additional abstraction layer, as discussed in chapter 2, may have potentially impacted the hardware resource utilization resulting in suboptimal performance for the prototypes. It is possible that the hardware utilized in our study was limited in terms of how much the servers could utilize, which could have further exacerbated the impact of the abstraction layer introduced by Docker.

Beyond the overhead of Docker, an additional reason for the outcome could have been attributed to the use of the chosen operating system. The Mac operating system by Apple may have affected how technologies such as virtual threads and Reactive WebFlux are handled on an operative system level, due to the magic that JVM performs on each operating system.

5.3 Data Aggregation and Databases

The use of a simulated data aggregation API, as opposed to a database for this study may limit the generalizability and applicability of the results to real-world scenarios

as incorporating a database test case could have been a more realistic approach to evaluating the performance of the prototypes and their suitability for various businesses that utilize Spring.

While the HTTP client with its self-exposed endpoints is a much more realistic approach, the use of thread sleep method with internal method calls could have provided with a much more focused study to differentiate on the performance between virtual threaded Spring Boot and Reactive WebFlux. However, the chosen approach is still justified by providing an initial comparison of the prototypes' performance by using similar libraries and same code base.

5.4 Possible Alternatives for Future Studies

5.4.1 Possible Use of Netty as Web Server for Reactive WebFlux

As results show and mentioned in chapter 3, it is mentioned that Reactive WebFlux runs optimally on non-blocking web servers such as Netty. The results display a clear advantage to virtual threads, which could have been due to Reactive WebFlux running on Tomcat. Considering the preference for non-blocking web servers, a subsequent exploration of Netty's integration with Reactive WebFlux would prove to be an intriguing future study.

5.4.2 Possible Use of VisualVM to Collect Health Metrics

Given that the only required health metrics were CPU and memory usage, VisualVM could be seen as an alternative option for health metrics software. Prior studies have indicated that VisualVM offers CPU- and memory usage data as a time series graph, which is like how Prometheus and Grafana were utilized in this study. Despite the potential benefits of using VisualVM, the familiarity of Prometheus and Grafana as widely used health metrics software packages in production environments in Stryda led to their selection over VisualVM. However, they may have posed additional overheads during the testing process, but their familiarity and established use were considered more advantageous. A prospective study employing a more comprehensive set of metrics provided by Prometheus, or the use of VisualVM, would incite interest in future studies.

5.5 Economic, Social, Ethical and Environmental Aspects

With the number of connections on the internet, the demand for high-concurrent web services is also apparent. Such high energy consumption can be resulted due to high CPU- and memory usage. With finding the best alternative energy-efficient solutions, there is a high chance that a more environmental approach exists. Since low performant services require more CPU- and memory usage, the higher energy consumption from those services has led to greater negative environmental impact. This is why there has been a need for energy-efficient solutions that also provide high concurrency.

Additionally, the widespread adoption of the Spring ecosystem can also bring additional benefits in the social and the economical aspect. There is a possibility that high-performing web services can be created without the need for horizontal scaling and reduce the number of machines required to run the server, thereby decreasing

the financial burden of such businesses which often run web services in cloud environments. With the introduction of virtual threads being a possible alternative to reactive, the possibility of fostering social collaboration and innovation can be a possible positive factor.

Even though some environmental and societal aspects were taken into consideration when conducting this study, the ethical aspects were not taken into action.

6 Conclusion

This thesis has presented a comparative analysis of the difference between normal threaded- and virtual threaded Spring Boot applications and Reactive WebFlux applications. These prototypes were built with a purpose to aggregate data given by its own separate stub endpoints and were used to conduct performance tests by requesting data at a given rate that was increased until the service failed to conduct the requests or eventual system failure. The stub endpoints were configured to provide data after a given fixed delay to simulate a realistic situation and these values were given 100 milliseconds and 500 milliseconds.

From the results, conducted at each fixed delay, the virtual threaded prototype outperformed the reactive and normal threaded prototypes with its ability to continuously provide successful responses and its low latency until its eventual failure. There were, however, some tests where the virtual threaded prototype displayed spikes of resource utilization and increased latency, which later decreased for subsequent tests. It is believed that it could be due to the virtual threads being in their early stage of development.

The use of Tomcat as the Reactive WebFlux might have affected the results as well as the creation of underlying services could have reduced the real-world practical insight that would be interesting to evaluate. However, the results show that there are possibilities that there are more options to Reactive WebFlux in the future for designing high performant web services.

6.1 Future Work

The results show that the performance between the virtual thread and reactive prototypes are similar. As such, it would be interesting to do further research in how these prototypes handle environments that support better hardware to evaluate exactly one or the other prototype fails. Additionally, further research into how prototypes behave in real-world practical environments such as in cloud services or implementing underlying services would be a good alternative.

Since Reactive WebFlux did not run on an optimal web server that supports non-blocking approaches, a future study on the performance difference where Netty is used instead of Tomcat on Reactive WebFlux would be an interesting evaluation to be conducted.

Lastly, with virtual threads getting more and more adopted by libraries, a future study on how virtual threads handle database operations compared to Reactive WebFlux could be an interesting study, which could be much more applicable to current business solutions where databases are often utilized in web services.

References

[1] Oracle, "Lesson: Concurrency," [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/. [Accessed 1 April 2023].

[2] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, "Operating Systems: Three Easy Pieces," Arpaci-Dusseau Books, 2018.

[3] B. Christensen and T. Nurkiewicz, "Reactive Programming with RxJava," O'Reilly Media, Inc., 2016.

[4] R. Pressler and A. Bateman, "JEP 425: Virtual Threads (Preview)," 18 January 2023. [Online]. Available: https://openjdk.org/jeps/425. [Accessed 1 April 2023].

[5] C. Escoffier and K. Finnigan, "Reactive Systems in Java," O'Reilly Media, Inc., 2021.

[6] R. Pressler and A. Bateman, "JEP 436: Virtual Threads (Second Preview)," 2 March 2023. [Online]. Available: https://openjdk.org/jeps/436. [Accessed 1 April 2023].

[7] IBM, " What is Java Spring Boot?," 1 April 2023. [Online]. Available: https://www.ibm.com/topics/java-spring-boot#anchor-1000848595. [Accessed 1 April 2023].

[8] A. L. Davis, "Overview," in *Spring Quick Reference Guide*, Berkeley, California, Apress, 2020, pp. 5-8.

[9] VMware, Inc, "Web on Reactive Stack," 20 March 2023. [Online]. Available: https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html. [Accessed 1 April 2023].

[10] S. Baslé, S. Deleuze, D. Karnok and S. Maldini, "Class Mono<T>," [Online]. Available: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html. [Accessed 1 April 2023].

[11] S. Baslé, S. Deleuze, D. Karnok and S. Maldini, "Class Flux<T>," [Online]. Available: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html. [Accessed 1 April 2023].

[12] J. Long, "Deploying Spring Boot Applications," 7 March 2014. [Online]. AvaiGhanshyamlable: https://spring.io/blog/2014/03/07/deploying-spring-boot-applications. [Accessed 1 April 2023].

[13] F. Gutierrez, "WebFlux and Reactive Data with Spring Boot," in *Pro Spring Boot 2*, Berkeley, CA, Apress, 2018, pp. 173-206.

[14] A. Nordlund and N. Nordström, "Reactive vs Non-Reactive Java framework," MID Sweden University, Sundsvall, 2022.

[15] S. Iwanowski and G. Kozieł, "Comparative analysis of reactive and imperative approach in Java web application development," *Journal of Computer Science Institute, 24,* pp. 242-249, 30 September 2022.

[16] K. Dahlin, "An Evaluation of Spring WebFlux," MID Sweden University, Sundsvall, 2020.

[17] D. Beronić, B. Mihaljević, P. Pufek and A. Radovan, "Achieving Efficient Structured Concurrency through Lightweight Fibers in Java Virtual Machine," Communication and Electronic Technology (MIPRO), Opatija, Croatia, 2020.

[18] D. Beronić, B. Mihaljević, L. Modrić and A. Radovan, "Comparison of Structured Concurrency Constructs in Java and Kotlin – Virtual Threads and Coroutines," Communication and Electronic Technology (MIPRO), Opatija, Croatia, 2022.

[19] G. Verma, "Virtual thread: performance gain for microservices," 23 December 2022. [Online]. Available: https://medium.com/naukri-engineering/virtual-thread-performance-gain-for-microservices-760a08f0b8f3. [Accessed 1 April 2023].

[20] A. Filichkin, "Project Loom with Spring boot: performance tests," 4 December 2022. [Online]. Available: https://filia-aleks.medium.com/project-loom-with-spring-boot-performance-tests-c007e0e411c8. [Accessed 1 April 2023].

[21] B. Gregg, "Systems Performance, 2nd Edition," Pearson, 2020.

[22] N. Poulton, "3: Installing Docker," Packt Publishing, 2020.

[23] M. Błaszczyk, M. Pucek and P. Kopniak, " Comparison of lightweight frameworks for Java by analyzing proprietary web applications," *Journal of Computer Science Institute,* pp. 159-164, 30 June 2021.

[24] IBM, "What are virtual machines (VMs)?," [Online]. Available: https://www.ibm.com/topics/virtual-machines. [Accessed 1 April 2023].

[25] IBM, "What is Docker?," [Online]. Available: https://www.ibm.com/topics/docker. [Accessed 1 April 2023].

[26] VMware, Inc., "49. Metrics," [Online]. Available: https://docs.spring.io/spring-boot/docs/1.3.8.RELEASE/reference/html/production-ready-metrics.html. [Accessed 1 April 2023].

[27] Apache Software Foundation, "Apache JMeter™," [Online]. Available: https://jmeter.apache.org/. [Accessed 1 April 2023].

[28] W. Glozer, "Wrk (Github)," 2012. [Online]. Available: https://github.com/wg/wrk. [Accessed 1 April 2023].

[29] T. Senart, "Vegeta (Github)," 2013. [Online]. Available: https://github.com/tsenart/vegeta. [Accessed 1 April 2023].

[30] Microsoft, "Microsoft Virtial Machines Pricing," [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/#pricing. [Accessed 1 April 2023].

[31] Oracle, "Class HttpClient," [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.net.http/java/net/http/HttpClient.html. [Accessed 1 April 2023].

[32] The Apache Software Foundation, "HttpClient Overview," 26 February 2023. [Online]. Available: https://hc.apache.org/httpcomponents-client-5.2.x/. [Accessed 1 April 2023].

[33] B. Clozel, S. Deleuze, A. Poutsma and R. Stoyanchev, "Interface WebClient," 28 September 2017. [Online]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/reactive/function/client/WebClient.html. [Accessed 1 April 2023].

[34] S. Baslé, S. Deleuze, D. Karnok and S. Maldini, "Class Mono<T>," [Online]. Available: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html. [Accessed 1 April 2023].

[35] Oracle, "Interface Future<V>," 30 September 2004. [Online]. Available: https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/Future.html. [Accessed 1 April 2023].

[36] Oracle, "Interface ExecutorService," 30 September 2004. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html. [Accessed 1 April 2023].

[37] M. Paluch, "Embracing Virtual Threads," 11 October 2022. [Online]. Available: https://spring.io/blog/2022/10/11/embracing-virtual-threads. [Accessed 1 April 2023].

[38] Docker Inc., *Docker Compose,* 2013.

[39] B. Brazil, "Prometheus: Up & Running," O'Reilly Media, Inc., 2018.

[40] VMware, Inc., *Micrometer,* 2017.

[41] S. Maldini, "Class Schedulers," [Online]. Available: https://projectreactor.io/docs/core/release/api/reactor/core/scheduler/Schedulers.html#parallel--. [Accessed 15 May 2023].

[42] D. Beronić and B. Mihaljević, "Performance Enhancements using Virtual Threads and Modern Memory Management," 17 May 2022. [Online]. Available: https://2022spring.javacro.hr/eng/Program/Performance-Enhancements-using-Virtual-Threads-and-Modern-Memory-Management. [Accessed 15 May 2023].

Appendix

Appendix A – Java Code Snippets

Listing A.1 shows the build clan method for Flux and thread sleeping. The method returns a list of clans depending on the clanCount and memberCount given during the creation of the object instantiation.

```
/** Builds a random list of clans. Will also perform a simulated sleep time. ...*/
1 usage
public Flux<Clan> getClansFlux() {
  return Flux.fromIterable(buildClanWithoutSleep())
      .delayElements(Duration.of( amount: threadSleep / clanCount, ChronoUnit.MILLIS));
}

/** Builds a random list of clans given by the clan count and member count. Will also perform a ...*/
1 usage
public List<Clan> buildClanBlocking() {
  try {
    Thread.sleep(threadSleep);
    return buildClanWithoutSleep();
  } catch (InterruptedException e) {
    e.printStackTrace();
    return null;
  }
}
```

Listing A.1 Stub method for building cla12n with configurable sleep duration.

Listing A.2 shows how to set up beans to override how threading is done in Tomcat and Spring Boot's task executor.

```
@Slf4j
@Configuration
public class VirtualThreadConfig {

  no usages
  @Bean(TaskExecutionAutoConfiguration.APPLICATION_TASK_EXECUTOR_BEAN_NAME)
  public AsyncTaskExecutor asyncTaskExecutor() {
    log.info("Creating asyncTaskExecutor with virtual thread");
    return new TaskExecutorAdapter(Executors.newVirtualThreadPerTaskExecutor());
  }

  no usages
  @Bean
  public TomcatProtocolHandlerCustomizer<?> protocolHandlerVirtualThreadExecutorCustomizer() {
    log.info("Creating protocolHandlerVirtualThreadExecutorCustomizer with virtual thread");
    return protocolHandler ->
        protocolHandler.setExecutor(Executors.newVirtualThreadPerTaskExecutor());
  }
}
```

Listing A.2 Configuration beans for enabling virtual threads.

Appendix B – Configuration for Prometheus

Scrape configuration for Prometheus. The scrape interval is set to one second, which means that the endpoint is requested every second.

```
scrape_configs:
  - job_name: 'spring-actuator'
    metrics_path: '/actuator/prometheus'
    scrape_interval: 1s
    static_configs:
      - targets: [ 'virtual-backend:8080' ]
```

Listing B.1 Scrape configuration for Prometheus.

TRITA – CBH-GRU-2023:086