Master's thesis

Master of Engineering, Software Engineering and ICT

2021

Heikki Kesa

# PERFORMANCE CHARACTERISTICS OF CURRENT MICROSERVICE FRAMEWORKS

MASTER'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Master of Engineering, Software Engineering and ICT

2021 | 54 pages, 1 page in appendices

Heikki Kesa

# PERFORMANCE CHARACTERISTICS OF CURRENT MICROSERVICE FRAMEWORKS

This thesis compares performance of microservice development frameworks. There were no studies found that directly compare the performance of microservice frameworks, although the subject is covered to a degree in studies that focus on other aspects of microservice development.

The results were gathered by creating two applications with each framework chosen for this study, which were Nest, Spring and Micro. First application was created to measure the overhead the framework adds, by only returning string "Hello World!" upon requests. Second was a real-life simulating application which authorizes requests and returns list of basic info about universities for the requested country. Each applications performance was then measured by firing requests to the application endpoints using stress testing tool. Request response times were recorded, together with memory and CPU usage values. Two types of tests were executed: one with a low amount of requests and another with high amount of simultaneous requests with shorter time in between. Each test was executed three times per application and the average values were used for the final results.

Nest was by far the fastest framework across all tests, followed by Spring and Micro. Springs median response times were close to Nests but its initial responses were the slowest of all three, attributed to the Java Virtual Machines slow warm up time. Micros median response times were consistently slowest. Resource utilization results showed Nest using the least amount of memory and CPU while producing the best results. Micro used the most CPU in all tests and Spring required the highest amount of memory in all but one test.

KEYWORDS:

microservice, framework, comparison, performance

OPINNÄYTETYÖ (YAMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Ohjelmistotekniikka ja ICT

2021 | 54 sivua, 1 liitesivu

Heikki Kesa

# NYKYISTEN MIKROPALVELUKEHYSTEN SUORITUSKYKYOMINAISUUDET

Tämä opinnäytetyö vertailee mikropalvelujen kehittämiseen soveltuvien ohjelmistokehysten suorituskykyä. Vastaavia aikaisempia tutkimuksia ei löytynyt, mutta on olemassa mikropalvelujen kehittämistä koskevia tutkimuksia jotka sivuavat suorituskykyä.

Tulokset kerättiin luomalla kaksi sovellusta kullekkin tutkimuksessa käytetylle kehykselle, jotka olivat Nest, Spring ja Micro. Ensimmäinen sovellus luotiin mittaamaan kehysten aiheuttamaa lisäkuormaa, palauttamalla vain merkkijonon ”Hello World!”. Toinen sovellus luotiin paremmin mallintamaan tosielämän käyttökohdetta, tässä sovellus valtuuttaa pyynnön pääsyoikeuden ja palauttaa listan korkeakoulujen perustietoja halutun valtion perusteella. Jokaisen sovelluksen suorituskyky mitattiin lähettämällä kutsuja sovelluksen ulkoiseen rajapintaan rasituskoetyökalulla. Rasituskokeesta taltioitiin pyyntöjen vasteajat sekä sovelluksen muistin- ja prosessorinkäyttö. Kaksi erilaista rasituskoetta suoritettiin: toinen pienemmällä pyyntöjen määrällä ja toinen suurella määrällä samanaikaisia pyyntöjä joiden välissä oli lyhyempi tauko. Jokainen koe suoritettiin kolme kertaa sovellusta kohti ja tulosten keskiarvoja käytettiin lopullisia tuloksia koostettaessa.

Selkeästi tehokkain kehys oli Nest, olemalla nopein kaikissa kokeissa. Toiseksi sijoittui Spring ja viimeiseki jäi Micro. Springin pyyntöjen vasteaikojen mediaani oli lähellä Nestiä, mutta sen ensimmäisten vastausten nopeus oli joukon hitain, joka oli seurausta Java Virtual Machine -virtuaalikoneen hitaasta käynnistymisestä. Micron vasteaikojen mediaani oli kaikissa testeissä hitain. Sovellusten resurssikäyttö osoitti Nestin käyttävän vähiten muistia sekä prosessoritehoa, vaikka se tuotti nopeimpia vastauksia. Micro käytti eniten prosessoritehoa kaikissa kokeissa, Spring taas käytti joukon eniten muistia kaikissa paitsi yhdessä kokeessa.

ASIASANAT:

mikropalvelu, ohjelmistokehys, vertailu, suorituskyky

# CONTENT


**1 INTRODUCTION ..... 1**
1.1 Motivation ..... 1
1.2 Research questions ..... 1
1.3 Methodology ..... 2
1.4 Previous studies ..... 2
1.5 Structure ..... 3

**2 BACKGROUND ..... 4**
2.1 History of microservices ..... 4
2.2 Microservice architecture ..... 5
2.2.1 Monolithic services ..... 5
2.2.2 Characteristics of modern microservices ..... 6
2.2.3 Advantages and disadvantages of microservice architecture ..... 8
2.3 Microservice frameworks ..... 9
2.3.1 Spring ..... 9
2.3.2 Nest ..... 10
2.3.3 Micro ..... 10
2.3.4 Comparison ..... 11
2.3.5 Other frameworks ..... 12

**3 METHODOLOGY ..... 13**
3.1 Hello World application ..... 13
3.1.1 Spring ..... 14
3.1.2 Nest ..... 15
3.1.3 Micro ..... 16
3.2 Universities application ..... 18
3.2.1 Authorization ..... 18
3.2.2 Universities service ..... 19
3.3 Orchestration ..... 20
3.4 Measuring the performance ..... 22

**4 RESULTS ..... 24**
4.1 Hello-World application ..... 24
4.1.1 Response times ..... 25

4.1.2 Resource utilization ..... 29
4.1.3 Success rate ..... 32
4.2 Universities application ..... 33
4.2.1 Response times ..... 33
4.2.2 Resource utilization ..... 37
4.2.3 Success rate ..... 41

**5 DISCUSSION ..... 43**

**6 CONCLUSION ..... 46**

**REFERENCES ..... 47**

# APPENDICES

Appendix 1. Source codes

# LIST OF ABBREVIATIONS (OR) SYMBOLS

| | |
|---|---|
| API | Application programming interface |
| HTTP | Hypertext Transfer Protocol |
| JSON | JavaScript Object Notation |
| JVM | Java Virtual Machine |
| JWT | JSON Web Token |
| RPC | Remote Procedure Call |
| TCP | Transmission Control Protocol |

# 1 INTRODUCTION

This chapter covers the motivation for this study, along with research questions and the methodology used to achieve the results. Also previous related studies will be covered. The last part will describe the structure of this thesis.

## 1.1 Motivation

Modern software projects are often designed following microservice architecture, meaning that applications are made up with multiple small services instead of one large one. Microservices that consist the application can be built with different languages and tools if needed, but when creating applications from scratch it might be beneficial to use same methods across the microservices, especially if the development team stays the same. Microservices can be created with custom implementations from the ground up, but using available frameworks can significantly speed up the development. Many established general software development frameworks, like Spring that is one of the frameworks compared in this thesis, offer tools to build microservices and companies might choose the frameworks they are most familiar with, to minimize the learning phase and risks. In some cases performance can be a critical requirement for the application but the performance differences between the frameworks are not easily available. This study aims to find out if there are significant performance differences between the microservice development frameworks.

## 1.2 Research questions

There are studies made about comparing performance across different programming languages and runtimes, described in chapter 1.4, but no studies that directly compare the performance of the microservice frameworks. Therefore the research questions for this study are:

1. Are there significant performance differences between microservice frameworks?
2. Do the performance differences correlate with memory and CPU utilization between the frameworks?

## 1.3 Methodology

This thesis is a quantitative study using combination of descriptive and correlational research methods to answer the research questions.

Data is gathered by creating two applications for each framework (Nest, Spring and Micro) in this study, one which is very minimalistic program to measure the overhead the framework adds and another which more closely emulates real-life scenario. The frameworks were chosen because of their differences, each frameworks uses different programming language, also the maturity and how widely they are used played a role, to see if more established frameworks have better performance than newer ones. Each application is ran in Docker containers on a same machine to ensure consistency. Performance is measured using stress testing tool which fires desired amount of requests to the application endpoints. Two different stress tests are used for each applications, a low load one with low amount of requests and a high load version with high amount of simultaneous requests with shorter times in between the requests. Request response times, memory usage and CPU utilization values are recorded for comparing the framework performance and analyze the resource utilization correlation. Each test is executed three times and the average values are used for the final results.

## 1.4 Previous studies

Scientific studies about microservice framework performance could not be found. Most performance related studies were comparing programming languages or other software development frameworks, but not from the microservice perspective.

One study was found where microservice development frameworks were evaluated, which included a chapter about performance. This study from Tuan, Beierle, Garzon and Mora in 2020 compared four microservice frameworks: Spring, Lagom, Moleculer and Go Micro. The Spring framework is also compared in this thesis and Go Micro is a predecessor for another framework compared in this thesis: Micro. The study shows that Go Micro uses by far the less resources compared to the other frameworks, memory and CPU usage are 8 to 10 times lower than the Spring implementation. Request response times were comparable across all the frameworks, but Spring showed slow results for initial requests. This was attributed to Java Virtual Machines slow warm up phase. [1]

Two individual online performance comparisons were found that included frameworks from this thesis, although both included only Java Virtual Machine based frameworks. A comparison from 2016 by Cyril Delmas showed that Spring framework placed second out of five frameworks, only losing to the fastest framework vertx by around 15% measured in response time. [2] A comparison made by Auryn Engel in 2019 also included Spring framework. Here Spring was the slowest of four Java Virtual Machine frameworks compared when producing response with large amount of data but was second fastest with small response. It was noted that Spring had clearly the slowest start time. [3]

A benchmarking of 32 microservice development frameworks was done by Steve Hu and other collaborators from 2016 to present day. The comparison only measures the overhead of the frameworks, all of the sample applications only return string “Hello World!” as a response. Results include Spring which is fourth to last with average latency of 83 ms per request. In the middle of the field in response times with 22 ms is Node.js framework which the Nest framework used in this thesis is based on. [4]

## 1.5 Structure

This thesis is divided into six chapters. The chapter following this introduction chapter covers the background information required to understand the main aspects of microservice architecture and microservice development frameworks. Third chapter covers the methodology of gathering the research data, this includes describing the created applications, running them in consistent fashion and getting the performance values using stress testing. Fourth chapter presents the results with informative charts and explanations. Fifth chapter discusses the results in more detail and explains possible effects and limitations the results have, with recommendations for future research. The last chapter ends this thesis with the final conclusions about the results.

# 2 BACKGROUND

This chapter aims to give readers adequate background information about concepts and technologies used and evaluated in this study. First, the microservice architecture is explained, with comparisons to monolithic architecture. Then the need for microservice frameworks is described. All of the microservice frameworks used in this study will be described in enough detail for readers to get proper understanding in their differences. All the frameworks are written in different programming language, which are detailed in the last chapter.

## 2.1 History of microservices

Microservice architecture has been a very popular topic in software development for past couple of years, but in order to describe it's current role, it is good to understand the history. The earliest implementations closely related to microservices have been done in the 1980s. Back then, the biggest limiting factors in computing were the processing power and the amount of available memory. Remote Procedure Calls (RPC) concept was developed to combat this issue. This enabled developers to run procedures and methods on other computers and build large machine-crossing systems to circumvent the physical limitations of the computers. This worked well for the time, but when more memory became available, it was discovered that distributing the processing logic too much introduced so much networking overhead that it heavily lowered the overall processing power. For that reason, the Facade pattern was discovered. The idea was to identify interfaces of the system and to only expose a limited amount of them to the wider network. This naturally guided developers to build subsystems that communicated with each other, but did not expose all the interfaces across the system. Later this evolved into each subsystem having their own external API. The principle of the APIs was to be business driven, or what is more commonly known as Domain Driven Design. The aim to design systems in domain driven way is to identify the actual business concepts the system is designed to fulfill and create the microservices for each concept. [5]

## 2.2 Microservice architecture

Although microservice architecture principles were defined by many influential programmers and innovators throughout the years, the most influential one is considered to be Martin Fowler. He has laid out general guidelines to follow when developing microservices and this chapter will explain the main points.

The main idea of microservice architecture is to develop an application that is made of multiple small services that communicate with each other. These services are created to fulfill business concepts and are independently deployed. There is ideally very little common management between the services, so that they could be written in different programming languages and have different database technologies. [6] A side-by-side comparison between microservice- and monolithic architecture is shown in Figure 1.

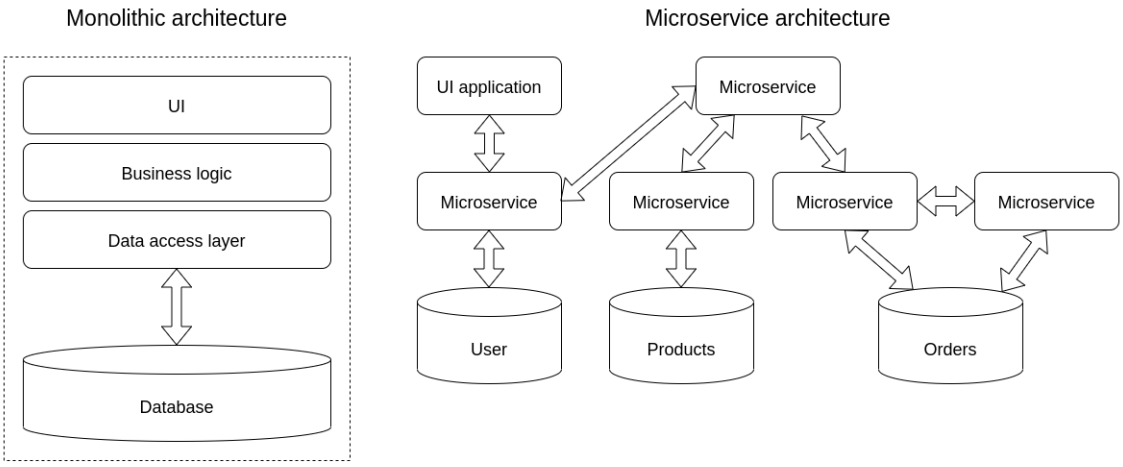


**Figure 1** Comparison between monolithic and microservice architecture. Monolithic architecture uses a single large database here, and all its logic resides in a single app. Microservice architecture has data divided into separate databases which are retrieved and handled by separate services.

### 2.2.1 Monolithic services

To explain the advantages of microservice architecture, it's required to understand the basics of monolithic architecture. A monolithic application is an application that is built as a single item. Common enterprise application usually consists of user interface, a database and a server-side application. Generally the server-side part is a monolith, a

single service that handles all the business logic of the application: HTTP requests, database manipulation, building and sending data to the front end. Any changes in this application would require rebuilding and deploying a new instance of the application on server. Benefits of this architecture is its modular nature, it is convenient for developers to use common code throughout the application. Monolithic applications can work well in smaller projects with few developers, or in cases where the application isn't being developed and doesn't need changes. But often when application grows larger, it increases developer frustration because all the small changes require predetermined release windows and testing workflows. Also the modularity becomes hard to maintain. Often developers are afraid to remove or refactor old code, as it might cause errors in somewhere completely different part of the application and opt to write new classes just in case. Also scaling is not optimal, as it would require to scale the resources of the whole application, even when only one part of the application requires more computation power. [6, 7]

### 2.2.2 Characteristics of modern microservices

One benefit of microservices is to think of them as components to build applications. Programs are commonly built using libraries, components that are designed to solve a certain problem. Libraries are run in-memory when application needs to call them. Applications could call and use the microservices like library components, but instead of in-memory calls they will be invoked over web service requests or RPC calls. This will result in slower response times, but also provides benefits. One of the benefits is being able to separately deploy changes to these services, as updates to libraries would require rebuilding and deploying the app using the library. Also, more processing power requiring services could be deployed on larger instances on cloud and the rest of the application could run on lighter instance. [6]

Other benefit is to organize teams around business capabilities. Traditionally software development teams have been organized by technologies, for example large application project could have separate teams for: front end, server logic and database. This could result in having some of the application logic in sub optimal places, caused by financial or time related issues. Microservice focused teams would be organized so that each team would aim to fulfill the required business capability, meaning that teams could have

front end, server side and database specialists working together. The teams would be responsible for their creations, instead of being forced to handle the whole monolith and all the related contexts. In addition to clearer responsibility areas, this relieves pressure of the individual developers. [6]

Generally software projects are developed by a developer team, then shipped when they are ready. After that a different team often handles running and maintaining it and the developer team is disbanded or moved on to next project. In monolithic applications, this is often reasonable way to manage product lifetime, as the applications and teams are large. When developing microservices, it is possible and even encouraged for the developer team to also run and maintain it, as the services and teams are smaller. Also the teams might already have maintenance members on board. [6, 7]

Microservice architecture gives developers freedom to choose technologies that are best for the service they are working on. In monolithic applications, developers are often using the same programming languages and databases throughout the entire application. This might be suitable option when the application is either small enough or very similarly structured. As microservices are not coupled with each other, this gives teams opportunity to use programming languages and other tools which best suit their needs. The different teams need to communicate about their service endpoints between other teams, but they do not need to know the internal intricacies of the services. [6, 7]

Microservices make it easier to decentralize the data management. In traditional monolithic applications, there is generally one large database that contains all the required data. This makes it cumbersome to make changes to database structure and often tables would have too much content that might be needed only by single component of the application. Microservice architecture enables teams to separate content into separate databases. For example user info management service needs to only know about users and product management service needs to have a list of all the products. Common misconception is that each microservice needs to have their own database, instead multiple microservices could use the same database if it suits their needs. Drawback of using separate databases is that it prevents using transactions when updating data. Services need to coordinate the changes between each other to make sure the data between databases stay consistent. [6, 7]

An important emphasis on developing microservices, is to make sure that the services can handle failures. When microservices are used as components, it is important for

them to recover from failures as smoothly as possible. When monolithic applications fail, often the whole application would stop working and continue on when recovered. But in microservice architecture, extra care needs to be placed on monitoring the services and making sure separate services would handle situation where other services fail. On the other hand, when designed correctly, single service failing would not be noticed by users at all. Also failing services could be fixed and deployed independently, without having to re-deploy or take down the whole application. [6]

Microservices are not limited to be used only in applications that are designed in microservice architecture in mind. It is common practice to develop new functionalities or rewrite existing ones in monolithic application using microservices. Old but actively used monolithic application could be modernized so that existing parts of the application would be replaced with microservices. The core of the application could stay monolithic, but all the new and refactored parts could be microservices. This generally eases the transition to new application as parts of it could be migrated one by one, instead of planning longer down times and release schedule for the whole monolith. [6]

### 2.2.3 Advantages and disadvantages of microservice architecture

Microservice architecture is relatively new concept in software development. It is already widely used by large companies, such as Netflix, LinkedIn and Amazon, but this doesn’t mean it is preferable to monolithic architecture in every situation. [7] One guideline for building applications is to first start the project by building a monolith and then convert parts of it to microservices when needed. [7] This is encouraged when it’s not clear what the application is going to end up like. It might introduce too much overhead and complexity to try to create microservice architecture for smaller projects. Also the team experience matters, teams that are experienced in creating successful monolithic applications might be slowed down by forcing them to create microservices and the end product might be worse. One major concern when creating applications with microservice architecture is that dependencies between services became too complex. Aim for microservices is to contain complexity inside services, but when the application becomes large enough, the services will be too tightly coupled together and keeping track of their connections becomes hard to maintain. All in all, microservice architectural style is still not mature enough to draw conclusions on if it is better than monolithic style or will there be major issues that are not discovered yet. [6]

## 2.3 Microservice frameworks

Microservices can be built with almost any of the technologies that are used to build monolithic applications. After all microservices are small programs that handle data and store or return it for other programs to use. When developing microservices, teams aim to map the business components into microservices and build them to fulfill that role. But a lot of development time is used on implementing technical features that are required for the service to function, but do not directly advance the business requirements. [8] For that reason, frameworks to help and speed up microservice development have emerged. Some are existing monolithic application development frameworks that have received microservice related modules in recent years, others are ones that have been built as a microservice frameworks from the ground up. This chapter will introduce three frameworks that have been selected for the comparisons in this study.

### 2.3.1 Spring

Spring is an enterprise grade open-source software development framework for Java programming language. It was first released in 2003 by Rod Johnson and is currently maintained by VMware, Inc. [9] It is the most popular Java framework by a large margin, over two thirds of all Java web applications are created using Spring. [10] The Spring frameworks most praised feature is its Dependency Injection (DI) implementation. The core of the framework uses its Inversion of Control (IoC) to manage object life cycles, initializing them and wiring them together when needed, so that the developers don’t need to do it manually. Dependency Injection is used to manage the connections between objects. To put it simply, DI injects the required dependencies for objects automatically when they are created.

The Spring ecosystem is divided into projects that are designed to fulfill specific needs. The most popular one and the one used in this research is Spring Boot. It is a collection of “starter” dependencies and conventions to speed up the creation of Spring projects. In addition to minimal libraries required to create Spring application, it also includes monitoring and external configuration features to make the application production-ready. Notable core features of Spring Boot: Internationalization, JSON library integrations, graceful shutdown, message streaming, security implementations, wide array of database technologies, caching, method validation features and testing utilities. [11]

The Spring projects are modular, meaning that developers are not limited to features provided by a single project. In this research, microservices created with Spring framework mainly contain the modules from Spring Boot, but also utilize modules from Spring Cloud project. The Spring Cloud project is a collection of tools to help managing connections and service discovery between microservices, providing API gateways and defining configurations for cloud deployments. [12]

### 2.3.2 Nest

Nest (also called NestJS) is an open-source JavaScript framework created by Kamil Mysliwiec, built on Node.js. Node.js is a JavaScript runtime that uses Google Chrome's V8 engine to enable running JavaScript applications server-side. [13] Node.js is the de-facto technology for building JavaScript back end applications, but it could be considered as a collection of tools instead of a complete framework. Nest has been created to solve that, it provides an architecture to speed up and ease the development of Node.js based applications. The architecture is heavily inspired by Angular, a popular front end JavaScript framework. Nest is a layer on top of Node.js but developers can also use Node.js directly if needed. Nest provides developers with a list of technologies and libraries, most notably: database integrations, external configurations, caching, scheduling, logging, security technologies, GraphQL integration, websocket integration and microservice implementations. In addition to these provided technologies, developers can utilize general JavaScript and Node.js libraries as well. [14, 18]

### 2.3.3 Micro

Micro is a framework for building cloud apps using microservice architecture, written in Go language. It is created by Asim Aslam and the first version was released in the August of 2020, although it is based on Go-Micro which was released in 2017. Micro is a direct successor to the Go-Micro, which is not actively developed anymore. Go-Micro was an array of services and utilities for creating microservice applications, but Micro is more of a platform that includes all the general components of the microservice architecture. Its goal is to abstract away the infrastructure that make up the microservice applications, like authentication, service discovery, event streaming, api gateway, etc. and enable

developers to focus on developing the actual microservices. In Table 1 is a list of services the Micro server consists of, as described by the official documentation.

**Table 1** Services of which the Micro server is composed of. [11]

| | |
|---|---|
| API | HTTP Gateway which dynamically maps http/json requests to RPC using path based resolution |
| Auth | Authentication and authorization out of the box using jwt tokens and rule based access control |
| Broker | Ephemeral pubsub messaging for asynchronous communication and distributing notifications |
| Config | Dynamic configuration and secrets management for service level config without the need to restart |
| Events | Event streaming with ordered messaging, replay from offsets and persistent storage |
| Network | Inter-service networking, isolation and routing plane for all internal request traffic |
| Proxy | An identity aware proxy used for remote access and any external grpc request traffic |
| Runtime | Service lifecycle and process management with support for source to running auto build |
| Registry | Centralised service discovery and API endpoint explorer with feature rich metadata |
| Store | Key-Value storage with TTL expiry and persistent crud to keep microservices stateless |

Micro also comes with tools to easily deploy the services into Kubernetes cluster, by automatically creating configurations for the whole architecture. Micro is heavily opinionated, to make microservice development as fast as possible, but this also makes it very difficult to use different technologies or architecture designs than what it is designed for. [15]

### 2.3.4 Comparison

The three frameworks in this study are all somewhat different. Spring is the most used and very widely known one with a long history, Nest is newer but has been rapidly gaining propularity. Both of them are general use frameworks that have capabilities to be used

as a microservice framework. Micro is clearly the youngest and still in very active development, it is also the only one that has been designed as a tool for developing microservices for cloud architecture. In Table 2 is a comparison of the three frameworks, but it is recommended to keep in mind these differences when comparing the values. All the values are from writing of this document.

**Table 2** General comparison between the frameworks used in this study. [14, 15, 16, 17, 18]

| | Spring | Nest | Micro |
|---|---|---|---|
| Production release | 2004 | 2017 | 2020 |
| GitHub stars | 54500 (Spring Boot)<br>42200 (Spring Framework) | 36100 | 9900 |
| Language | Java | JavaScript | Go |
| License | Apache-2.0 | MIT | PolyForm Shield License |
| Notable users | *Very widely used*<br>Netflix<br>Amazon<br>Ebay<br>Visa<br>Twitter<br>Zalando | Adidas<br>Autodesk<br>Capgemini<br>Sanofi | Imgur<br>Tencent<br>Vimeo<br>Volvo |

### 2.3.5 Other frameworks

The frameworks chosen for this study were because of their popularity, usage in the company that ordered this study and the different programming languages used. There are other frameworks that are gaining popularity or are specially tailored for microservice use. Very popular choice for running microservices on Java Virtual Machine is Vert.x, it is designed to be very lightweight and fast, making it a good choice for creating microservices. It is more of a toolkit instead of full fledged framework. Other Java Virtual Machine based option is Micronaut, it is built as a microservice framework from the ground up. The main advantages of it are ease of unit testing and perfromance, having a fast start up time and small memory footprint. [19] Popular framework for Python language is Flask, it is a general web application framework but emphasizes on using a microservice architecture when building applications. It is developed to be lightweight and simple, still having the tools needed for modern applications. [20]

# 3 METHODOLOGY

The aim for this study was to measure and evaluate real life performance of the frameworks. Two separate sample application were developed for each framework. Th first application was minimalistic “hello world” program, for measuring how much the core of the frameworks themselves affect the performance. The second application was emulating more realistic real-world scenario, in order to find out if the performance difference will multiply or stay near the hello-world level when the program gets more complicated. The full codebase of the applications and the performance testing orchestration can be found from link in Appendix 1. Below both of these applications are explained in more detail, together with differences the different frameworks cause for the architecture of each application. Also the performance measuring orchestration is explained.

## 3.1 Hello World application

The first application was created for measuring the difference in performance each framework would cause in the simplest possible scenario. Hello World application consisted of a gateway program which acted as a HTTP-endpoint that requested a response from the hello-world service and returned it to requester. The hello-world service only returned the text “Hello World!”, without doing anything more special. When implementing the applications for each of the framework, it was decided to implement them as developers would generally program them, utilizing the official documentation of each framework. That caused the applications to differ somewhat from each other, like Spring implementation having a service discovery service to provide info about the location of the hello-world service for the gateway and the Micro implementation having other supporting services that are always running by default. This meant that the applications were not forced to be implemented as carbon copies of each other, but would instead be more realistic versions and would give more useful results from the measurements.

### 3.1.1 Spring

The Spring implementation consisted of three services: gateway service for handling the requests, hello-world service for returning the “Hello World!” string for the gateway and an Eureka server to provide service discovery between those services. When running the application, first the Eureka server was started, then the gateway and hello-world services registered themselves for Eureka. When all the services were running and registered the gateway listened for HTTP GET-requests, upon receiving the request it retrieved a list of hello-world services from Eureka and requested a response from the only hello-world service available, finally returning the response for the GET-request. See the visualization of the architecture in Figure 2. All the services were running on Apache Tomcat servers in separate Docker containers. All of the services were built on Spring Boot starter package extended with Netflix Eureka libraries from Spring Cloud project. In Table 3 are listed the packages and versions used in the Spring implementation.

**Table 3** Versions of software used in Spring hello-world implementation

| | Version | Info |
|---|---|---|
| Java | openjdk 11.0.10 | |
| Tomcat | 9.0.41 | |
| Spring Boot | 2.3.9 | Real-life application |
| Spring Boot | 2.4.2 | Hello-World application |
| Spring Cloud | Hoxton SR.10 | Real-life application |
| Spring Cloud | 2020.0.1 | Hello-World application |
| Eureka | 3.0.1 | Server used only in eureka service<br>Client used in gateway and hello-world services |
| io.jsonwebtoken | 0.11.2 | Only in real-life application |
| Docker | 19.03.13 | |
| Docker image | adoptopenjdk/openjdk11:alpine | |

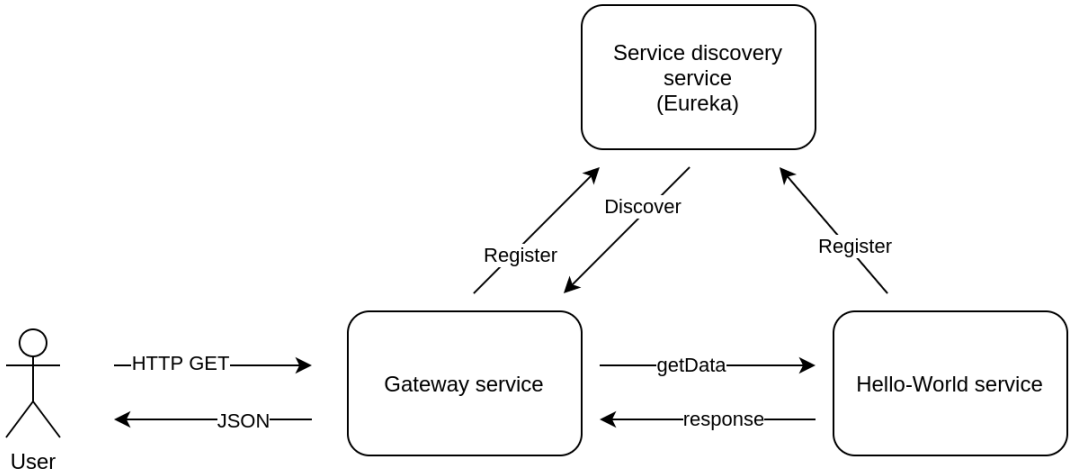


**Figure 2** Service architecture of the Spring hello-world application

### 3.1.2 Nest

The Nest application was more straightforward than the other two, as it didn't have service discovery server but instead used configured service destinations. Other than that, the architecture was very similar. There were two services, gateway and hello-world. The gateway listened to the HTTP GET-requests and requested hello-world service which returned the "Hello World!" string, then the gateway returned this as a response to the GET-request. The hello-world service was created using Nests "Microservices" library, data transfers between the gateway and hello-world services were handled using TCP -transmissions. See the architecture visualization in Figure 3. Nest "Microservices" library also supports other transport layers, but TCP is used as a default. In Table 4 are the packages and versions used in the Nest implementation.

**Table 4** Versions of software used in Nest hello-world implementation

| | Version | Info |
|---|---|---|
| Nest | 7.5.1 | |
| Nestjs/microservices | 7.6.5 | |
| Nestjs/passport | 7.1.5 | Only in real-life application |
| Node.js | 14.15 | |
| Express | 4.17.1 | |
| Typescript | 4.0.5 | |
| Docker | 19.03.13 | |
| Docker image | node:14.15-alpine | |

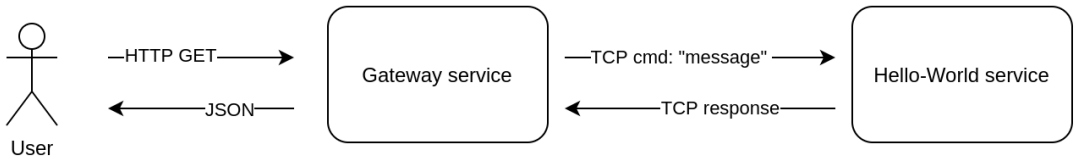


**Figure 3** Service architecture of the Nest hello-world application

### 3.1.3 Micro

The Micro application differed the most of all three, as Micro is more of a full architecture development platform, instead of a framework for building the services by hand. That meant that even the simple hello-world implementation required to have all the core services running, which were described in chapter 2.3.3. There was also difference in orchestrating and running the application, applications made with other frameworks in this study had their services running in their own Docker containers, but this wasn't possible with Micro. Micro is designed so that it is easy to deploy the services into Kubernetes cluster, with each service being automatically run in their own pods. As of writing this document Micro architecture used mDNS (Multicast DNS) protocol for resolving hostnames between services, which works fine in Kubernetes environment but that protocol wasn't supported by Docker. Between adding support for some other method for connecting between containers or running the services in one container, the latter was chosen. Mainly because it more closely matches the developer experience

when developing Micro application using the official documentation. It should be noted that this could have a small positive effect on performance, as the data is transferred inside one container instead of across multiple containers. See the architecture visualization in Figure 4.

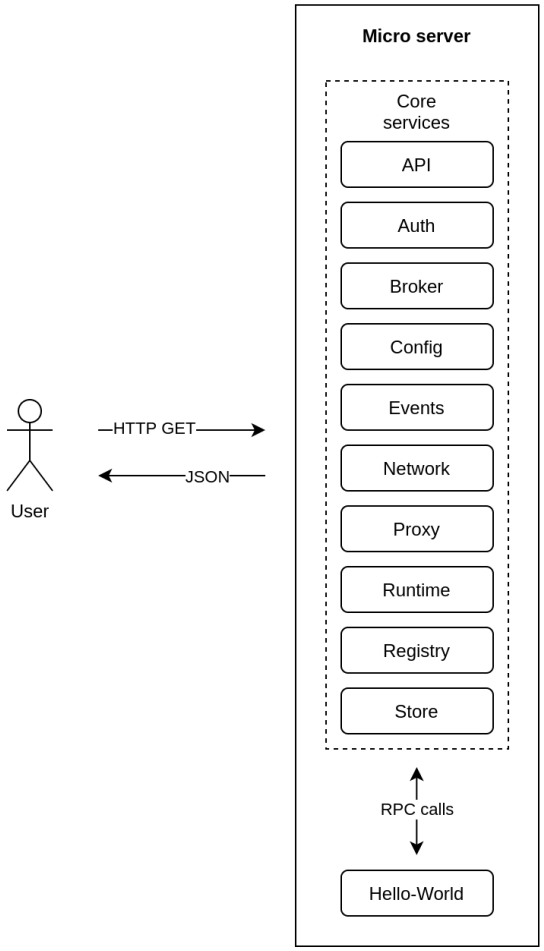


**Figure 4** Service architecture of the Micro hello-world application

The application itself was created by creating a service that returns the “Hello World!” string and that service was registered to the Micro server. The servers API service automatically created an endpoint for this service which was callable with a GET request. The messaging between the API gateway and the service was automatically handled by messaging service. Messaging between Micro services is handled using gRPC system which uses RPC (Remote Procedure Call) protocol. This requires proto -files that define the structure of the data which is transferred. [21] For the hello-world service, Micros

service generation CLI command automatically created the required proto file. In Table 5 are the packages and versions used in the Micro implementation.

**Table 5** Versions of software used in Micro hello-world implementation

| | | |
|---|---|---|
| Micro | 3.0.0 | |
| Go | go1.16 | |
| libprotoc | 3.14.0 | For compiling proto files |
| Docker | 19.03.13 | |
| Docker image | micro/micro | Contains the Micro server |

## 3.2 Universities application

In addition to measuring framework overhead performance by creating simplified hello-world applications, also more realistic real-life emulating applications were created, to see how much additional complexity affects the overall performance. The core of the applications were similar to the hello-world implementations, as the applications still had the gateway service for handling requests and retrieving data from other services. But instead of returning just a “Hello World” string the applications returned list of universities, filtered by country parameter given in the request. Applications also authorized the requests by verifying JSON Web Token (JWT) of the request header. Next chapters describe how the authorization and university retrieval was handled in the different applications.

### 3.2.1 Authorization

Increasingly common way to handle user authorization is using JWT. JWT is a JSON object that is digitally signed using a symmetric HMAC (hash-based message authentication code) method or asymmetric public/private key pair method, utilizing RSA (Rivest-Shamir-Adleman) or ECDSA (Elliptic Curve Digital Signature Algorithm) algorithms. [22] JWT is usually returned to the client after authenticating user, so that it could then be used in future requests that might need authorization for reaching certain parts of the application. For this study, JWTs with very long expiration dates were created

for each application. The JWT was added as an HTTP header for each request, in the most commonly used scheme: “Authorization: Bearer eyJh...”. The Spring and Nest applications handled the token authorization in the gateway service before allowing access to the universities service. Micros Auth service handled authorization semi-automatically.

Handling authorization in Spring application required the most custom code of all three. JWT authentication filter was added for the gateway service. The filter parsed the authentication part from the request header, verifying the signature and reading the contents of the object. Upon successful verification, it added new authentication metadata for the request, enabling the request to be forwarded to the universities service, as all requests required authorization in the Spring application.

Nest application also handled authorization in the gateway service. An authorization guard was added for all the endpoints in the application. In Nest, guards are used as barriers to allow or prevent access to endpoints. A custom JWT authorization guard was created that extended the authorization guard class from “nestjs/passport” library. Guard validated the request by automatically reading the JWT from the request header and verifying its signature, then allowing access to the universities endpoint.

Micro used its Auth service to automatically authorize access to endpoints it was configured to authorize. When setting up the Micro server, a CLI command was run that set authorization requirement rule for the universities service and removed the default rule which enabled free access to all resources, meaning that all the available endpoints required authorization. Implementing the authorization required the least work compared to other frameworks, but this again makes it harder to implement custom authorization methods if Micros default method isn’t sufficient for the application in development. For this study, it was sufficient as it also uses JWT from request header to authorize requests.

### 3.2.2 Universities service

For measuring the frameworks performance when performing more demanding tasks and handling larger amounts of data, a JSON processing service was developed. An open-source project “University Domains and Names Data List & API” was used, mainly its JSON -file that includes basic info of most of the universities in the world. The data

included for each university were: web page, name, domains, country two letter code, country and state/province if available. [23] See a snippet of the data in Figure 5.

```
...
{
  "web_pages": ["http://www.seikei.ac.jp/"],
  "name": "Seikei University",
  "domains": ["seikei.ac.jp"],
  "alpha_two_code": "JP",
  "country": "Japan",
  "state-province": null
},
...
```

**Figure 5** Snippet from the university list JSON -file.

The file was 77553 lines long at the time of building the applications. Aim for the universities service was to filter the contents of this file based on the country name parameter which was given in the query string parameter of the request and return a JSON -response with the filtered content. The file was located in the services static file location and was read when it was requested. The filtering itself was programmed using the best practice method of each language. In Spring implementation a stream was created from the file and it was filtered using “java.util.stream.Stream.filter” function, with string comparison predicate to match the countries. Nests universities service was the most straightforward one, as the JSON file is a JavaScript object, it only required importing the file contents and filtering it with the standard “filter” method, using string comparison as a predicate. Micro being a Go language framework, meaning that a bit more work was required to implement the university service. First the file was read into memory using “io/ioutil” library and then converted to a list of Go structures using “encoding/json” library, both of these are core libraries of Go language. Finally a new list was built by iterating through the previously created one and adding the universities that match the country string.

## 3.3 Orchestration

In order to make running the applications and performance measurements as consistent and repeatable as possible, a Makefile that handled the whole process was created. GNU Make is a widely used tool for in Unix systems, generally used for building and installing programs. [24] Make is essentially a tool to execute series of commands defined in a Makefile, making it a good way to orchestrate running certain list of tasks when needed. The application startup and measurement taking workflow was run separately for each application. The workflow consisted of 5 parts: creating and starting the Docker containers, reading and storing the CPU usage values before running the tests, running the performance tests, reading and storing the after test CPU and memory usage values, shutting down the containers.

The Docker container configurations were defined for each service separately in Dockerfiles. As each application, except the Micro implementation consisted of multiple services that ran in their own containers, a Docker Compose was used to define the relationships and networking between the services. Docker Compose was also used for building and starting/stopping the containers in Makefile.

CPU and memory values were read from files where Docker stores them for each container. In Unix systems the location for CPU usage was "/sys/fs/cgroup/cpu,cpuacct/docker/<container-id>/cpuacct.stat", the container ID was retrieved using Dockers "ps" command. Memory usage file was located in "/sys/fs/cgroup/memory/docker/<container-id>/memory.max_usage_in_bytes". The CPU file contained the amount of CPU usage the container processes had accumulated in total, meaning that it had to be read before and after running the performance tests and then calculate the difference to get the total usage. The CPU usage value was in a unit called a "jiffy", which has historically been ticks of $1/100^{th}$ of a second. Nowadays with variable frequencies and tickless kernels, the value isn't useful in itself but suits for comparing processes ran on the same machine, as in this study. The memory usage was easier to read as the file contained a value for maximum memory usage that had occurred during the lifetime of the container, the file was read after running the tests. [25]

After recording the CPU and memory values, the performance tests were run. The testing was done using k6 load testing software and will be described in more detail in the next

chapter. In short, a separate script was created for each application that included the testing options and output location.

Right after the performance tests were finished, the CPU usage values were recorded and the containers were shut down.

## 3.4 Measuring the performance

For measuring the responsiveness of the applications, some load testing method was required. Getting the response time of a single request to application endpoint would be simple, but in order to better evaluate real-life performance a proper load test with large amount of simultaneous requests was needed. Original idea for load testing was to write a script that would fire off requests to the endpoint and record the response times. But upon researching the subject, it became obvious that the traffic generation capability of this method was very low compared to specialized load testing tools, in some cases over 100 times slower. [26] Also, the load testing tools offered other benefits such as better accuracy and more detailed measurement outcomes. An open-source load testing tool k6 (version 0.31.1) was chosen for this study. Other testing tools were evaluated, but k6 was chosen because its usability when running from command line interface, meaning it suited well for the Make orchestrated workflow. Also its traffic generation capabilities were good compared to other tools with similar usability abilities. [26] In addition to the request execution time, the output the k6 generated also included more detailed statistics such as how many of the requests succeeded, how long the requests were in blocked state in average and how much data was transferred. The way k6 generates load is by creating what they call Virtual Users (VUs) and executing a custom function, which is defined by the tester. The function is repeated until the tests timeout is reached. For this study, this custom function calls the endpoint of the application and waits for the response, after that it verifies that the response has an HTTP status code 200, which is a standard response for successful HTTP request. Finally it waits for a defined time before ending the function, to simulate an actual user or application reading the content of the response. k6 is configured to run multiple VUs simultaneously and to repeat their functions for a preconfigured time. All the applications are configured to have same amount of VUs, wait times and total iterations, meaning that all applications receive the exact same amount of total requests. This is crucial, as we compare the total CPU and memory usage between the frameworks.

Initial plan was to run the applications and the measurements in a cloud instance, for example AWS (Amazon Web Services) or GCP (Google Cloud Platform), to match the common real-life scenario, as most applications are deployed to cloud. But upon researching the best ways to handle performance testing in cloud, it became clear that it will produce inaccurate results. There are several causes for this. Cloud server instances are generally multiple virtual servers on a physical server. The actual physical server details are not disclosed for the end user, as providers might move the virtual servers around onto other physical servers, for maintenance or other reasons. Also the processing power sharing between virtual server instances makes it impossible to get reliable performance testing results. Even the time of day affects the available processing power the instance has available. [27] For those reasons the measurements were decided to be executed on single computer, that way we could ensure that the available processing power stays the same between the tests and applications. Before executing the tests, the computer was rebooted to minimize the performance loss caused by other processes. As described in the previous chapter, the Docker containers were always shut down after running the tests, this combined with running each application separately ensured that the applications didn't affect the performance of each other. Also the tests were ran multiple times per application and the result averages were used. In Table 6 is a table of the computer used for running the performance tests.

**Table 6** Details of the computer used for running the performance tests

| Computer | Lenovo ThinkPad T490 | Manufactured 7/2019 |
|---|---|---|
| CPU | Intel i7-8565U @ 1.8GHz | |
| Memory | 40GB DDR4 @ 2.4GHz | |
| Operating System | Ubuntu 20.04.2 LTS | |
| Kernel | Linux 5.4.0-72-generic | |
| Bios | N2IET83W (1.61) | |

Each application was tested with two different load testing configuration. First was a low load scenario with 10 VUs and ran for 100 iterations, with each VU waiting for one second between the requests, which means 10 requests per VU. Second scenario included 100 VUs with half a second wait time and ran for 10000 iterations, producing 100 requests per VU. The reason for using two different scenarios with light and heavy load was to find out how much the requests affect the performance and hardware usage, otherwise

the resources used by just starting the applications would play too large role in the results.

# 4 RESULTS

This chapter presents all the results that were recorded by running set of tests for each framework, more in detail description is in the previous chapter. Here is a quick rundown of the testing and measuring process.

Each framework had two sample applications created: the “Hello-World” application and a more complex “Universities” application that simulates more realistic real-life use case. For both of the applications two separate load testing runs were executed: one with low amount of simultaneous requests and with longer break in between the requests and other more demanding one with more requests for longer time and with shorter breaks between the requests. Each test was ran three times per framework and average of the measured values were used for the final results. The main measured value was the time taken for the application to return a result, both median and 95$^{th}$ percentile values were recorded. Some requests took significantly longer to respond than others, most probably the initial ones, as the applications didn’t have any caching or other optimizations set up at that point. For that reason the 95$^{th}$ percentile results would be most informative when interested to know how fast the applications respond for majority of the requests as the slowest ones are left out, the value shows the highest request duration for those of 95 percent of the shortest request times. Also total memory and CPU usages were recorded. In addition to comparing resource utilization between the frameworks, it also gives an idea how much the frameworks itself use the resources and how much the actual functionalities of the applications raise it. The performance tests verified that the responses were successful, by checking that the response status was HTML status code 200. Most of the applications managed to always respond with successful response, but the more demanding “Universities” application failed to do so with some frameworks when running the high-load stress tests.

## 4.1 Hello-World application

The aim for the Hello-World application was to measure the overhead the frameworks add, as the service itself was as light as possible, only returning string “Hello World!”.

### 4.1.1 Response times

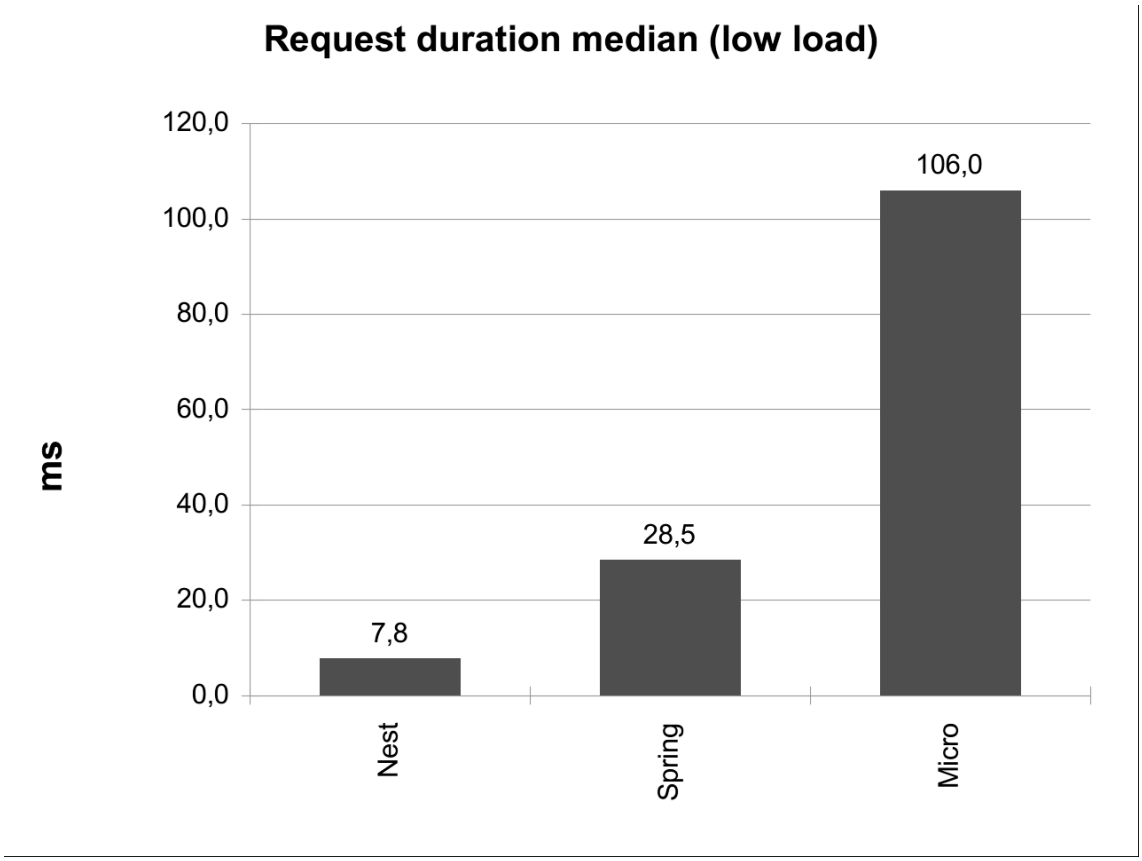


**Figure 6** Median of request duration for Hello-World application in low load stress test

As seen in Figure 6, the low load stress test showed that Nest responded to request much quicker than the other two frameworks, although the Spring was quite close. Micros median for request duration was 3.7 times slower than Spring and 13.5 times slower than Nest, showing that Micro being more of a full fledged all-in-one platform, it has much more overhead for handling simple requests than the other two frameworks.

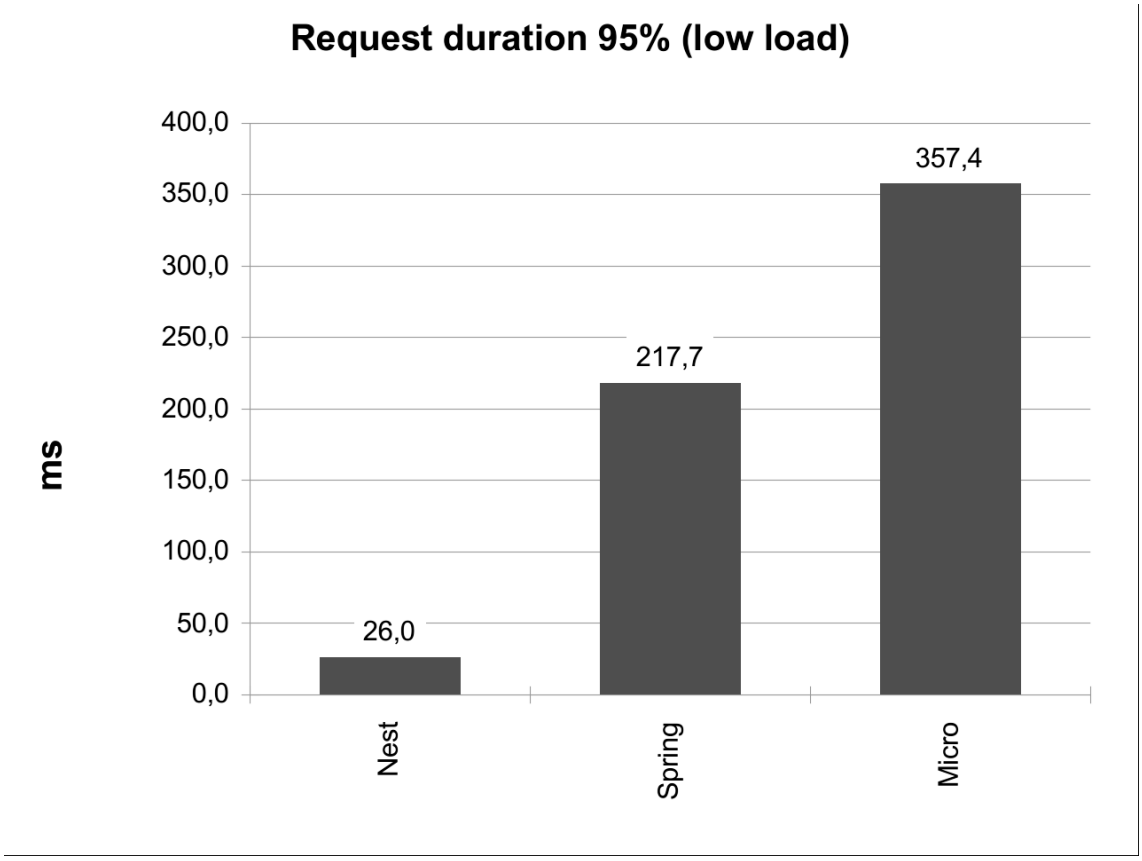


**Figure 7** 95th percentile request duration for Hello-World application in low load stress test

The 95th percentile values compared to median show that Spring requires more warming up to handle the first requests it receives, as its median response times are nearly 8 times faster than the 95th percentile ones. Shown on Figure 7.

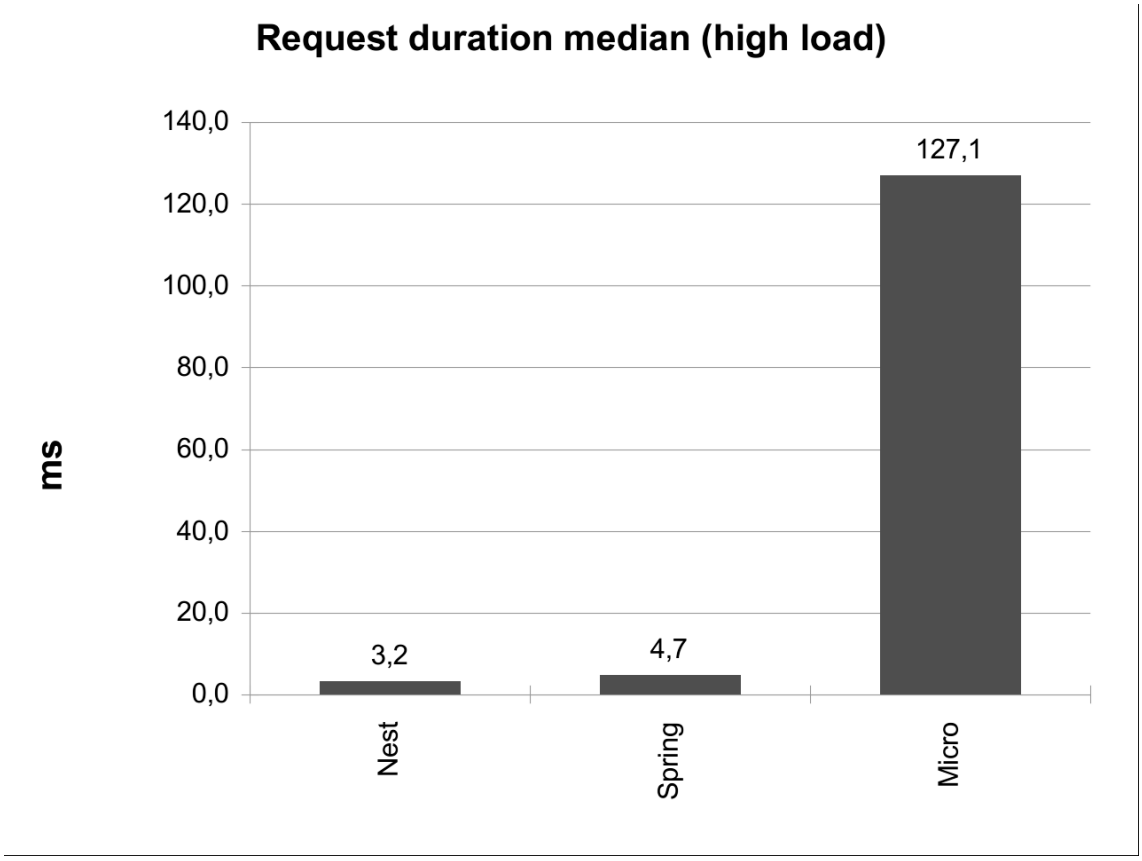


**Figure 8** Median of request duration for Hello-World application in high load stress test

High load stress tests show that Nest and Spring are very close in handling the simple requests, Micro being again clearly slower. Figure 8 shows that the median request durations between Nest and Spring are very low and close, but Micro is over 40 times slower.

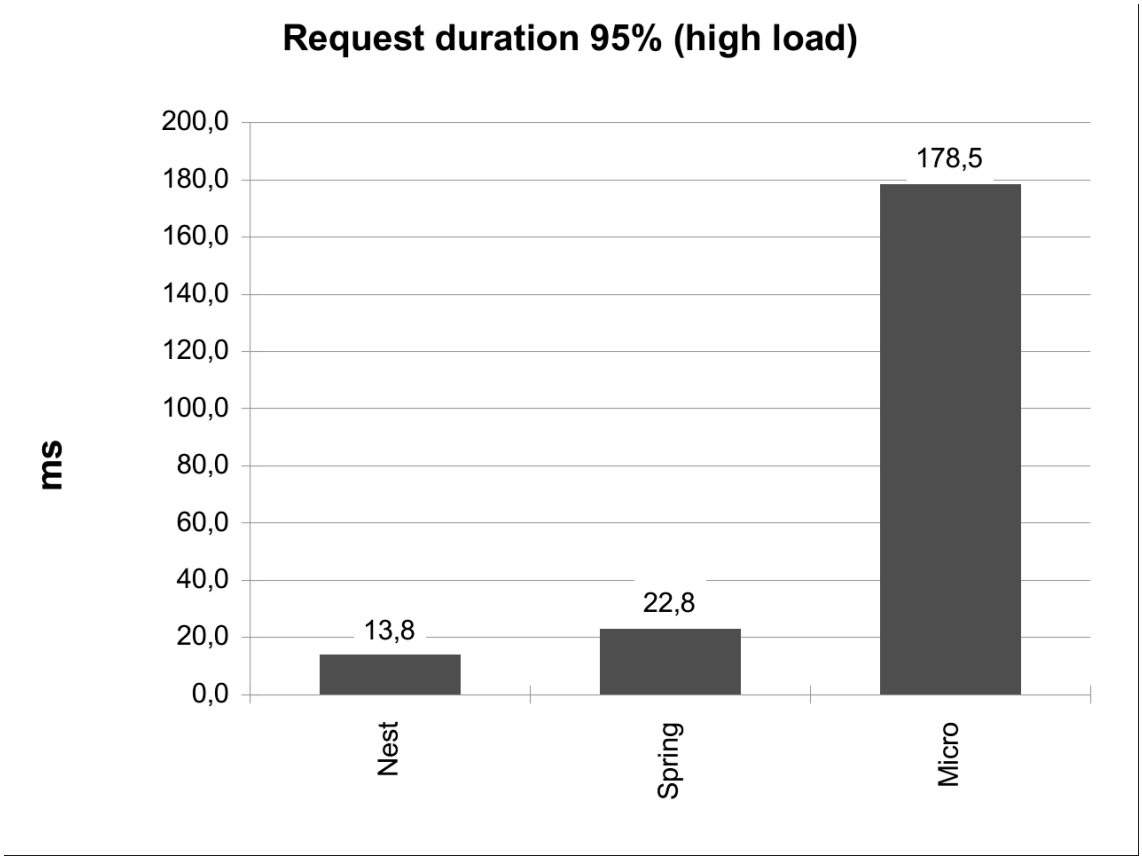


**Figure 9** 95th percentile request duration for Hello-World application in high load stress test

All the values show that initial requests take longer time to respond compared to the subsequent ones, but as shown in Figure 9 the 95$^{th}$ percentile values are noticeably lower than in the low load test as 95$^{th}$ percentile includes less of the initial responses, explained by high load test having 10000 total requests compared to low loads 100.

4.1.2 Resource utilization

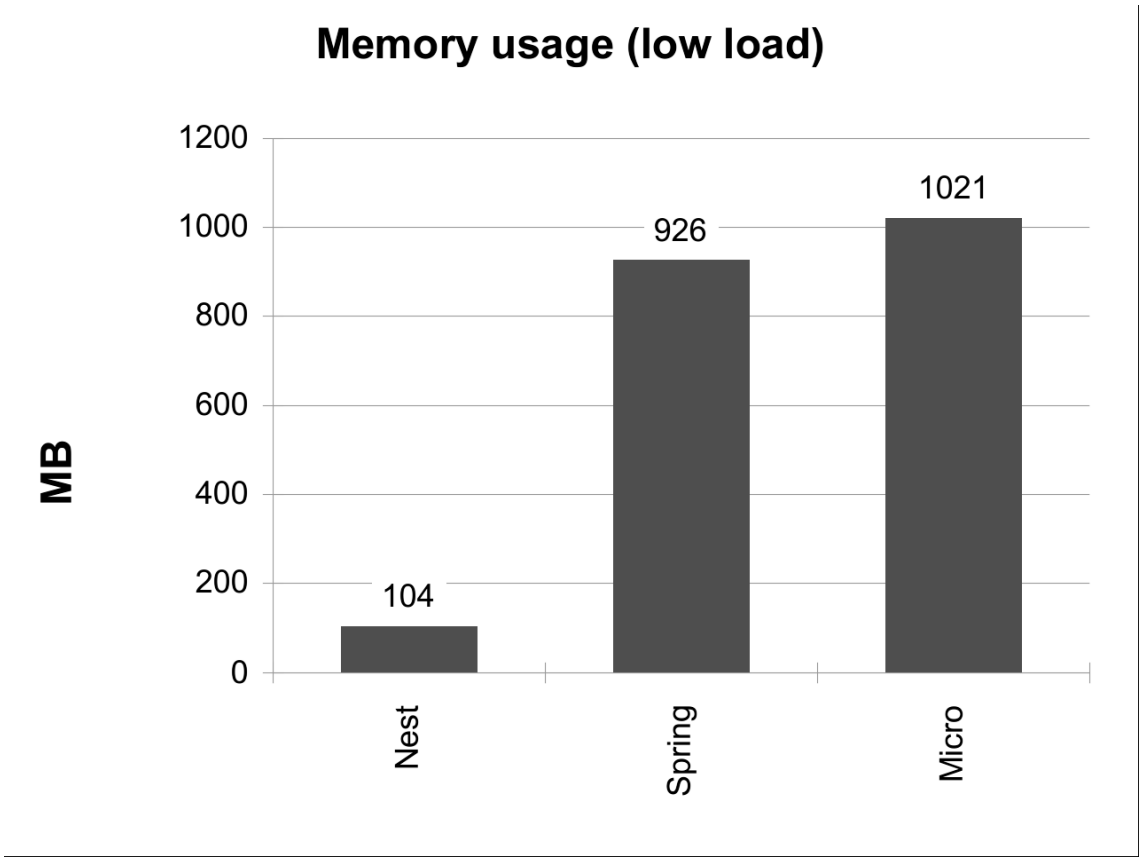


**Figure 10** Memory usage of Hello-World application in low load stress test

Memory and CPU usage in low load test showed that Nest used noticeably less resources than Spring and Micro. As shown in Figure 10 Spring and Micro used somewhat same amount of memory with only around 10% difference, but Nest used only approximately only 11% of memory compared to the other two.

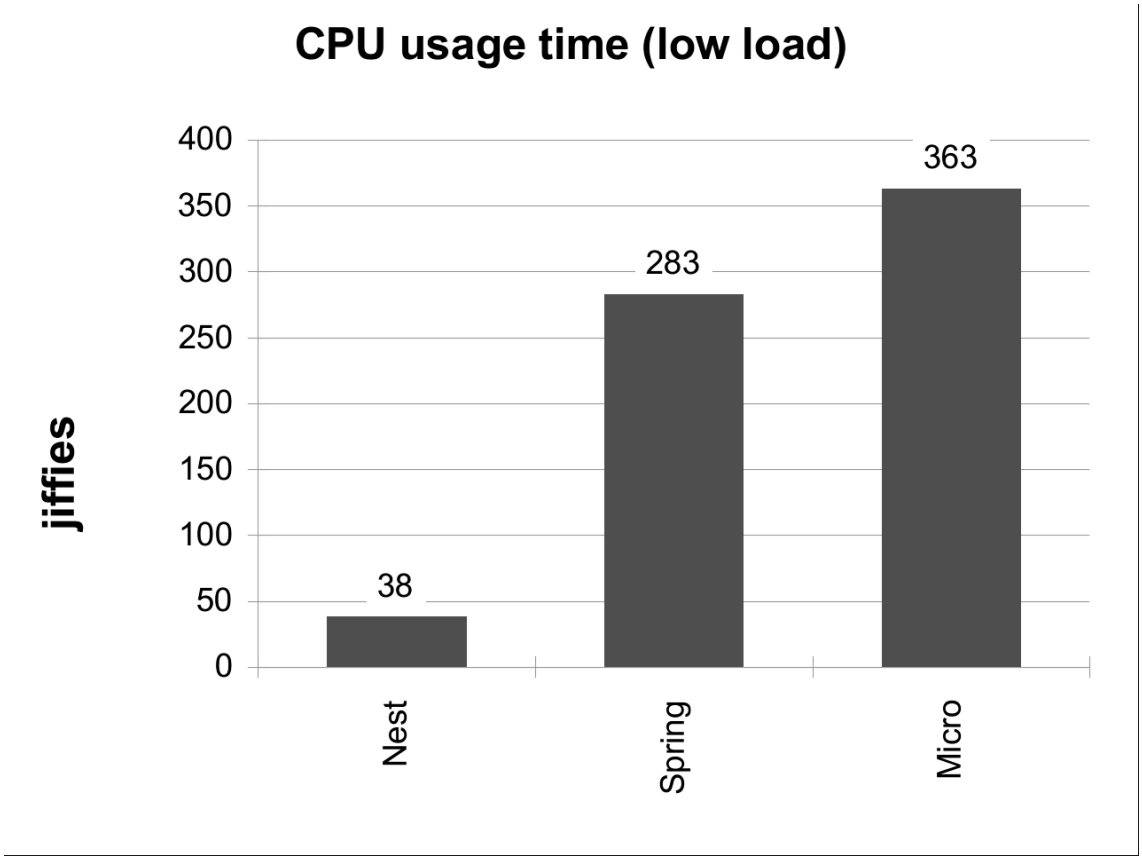


**Figure 11** CPU usage time of Hello-World application in low load stress test

CPU utilization showed similar results to memory utilization, with Spring utilizing CPU 7.5 times more than Nest and Micro being the most demanding with 9.5 times more CPU time than Nest. As Figure 11 shows, Nest utilized clearly the least amount of CPU, which raised doubts that if the starting sequence might make that big difference instead of producing the responses. But the testing code was reviewed to make sure that the values were recorded only for the time the actual stress tests were ran, eliminating this worry.

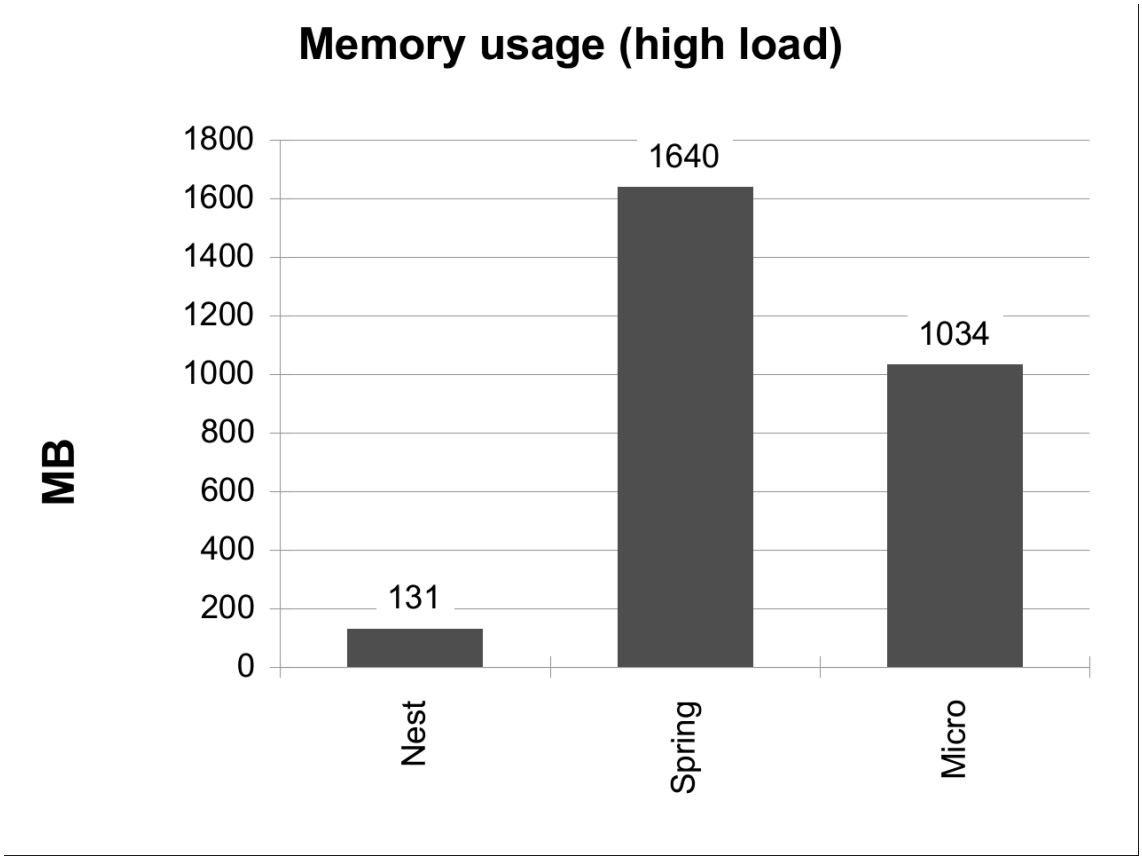


**Figure 12** Memory usage of Hello-World application in high load stress test

High load test produced clearly different results from the low load one. As seen in Figure 12, Spring used the most memory of all three, requiring 1.6 times more than Micro. Nest being again the most lean with requiring less than 8% of the memory the Spring used.

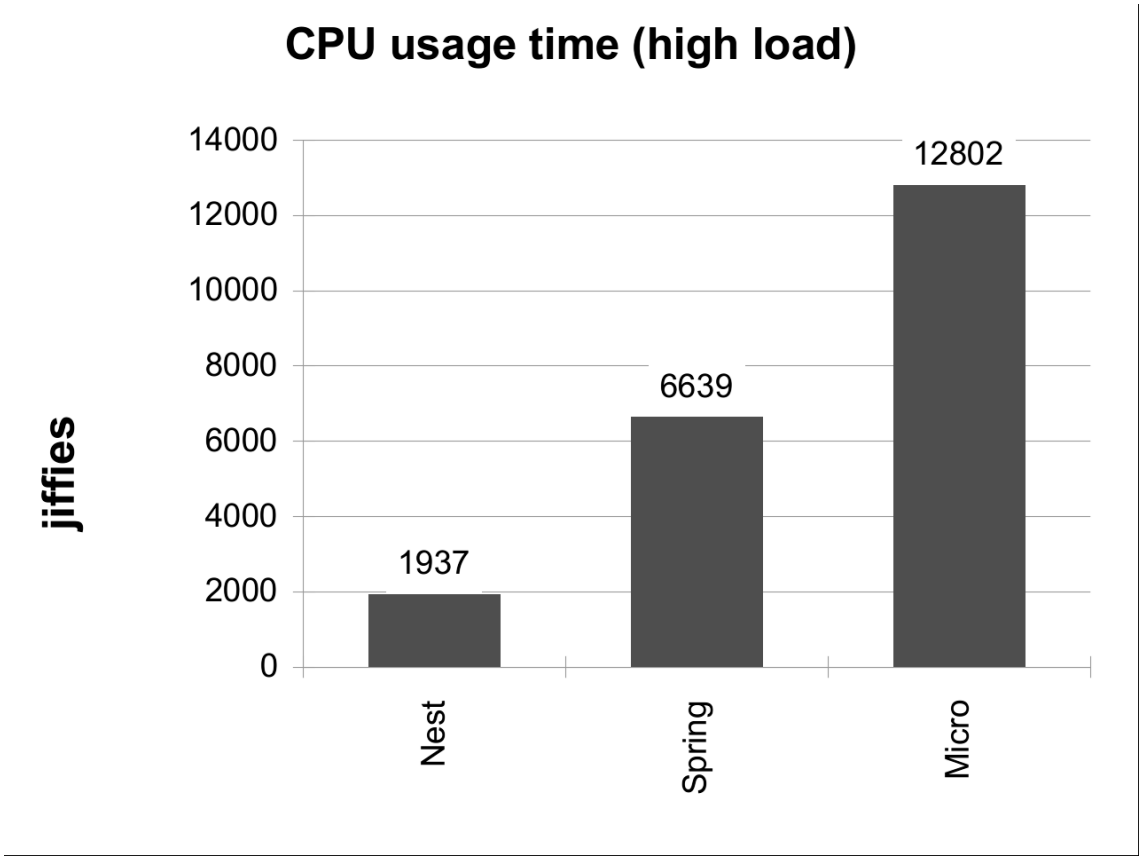


**Figure 13** CPU usage time of Hello-World application in high load stress test

CPU usage showed Nest being the most efficient again, but here Micro utilized CPU almost twice as much as Spring. Every measurable value up to this point was placing Nest first, Spring second and Micro third. But Figure 13 shows Spring having the highest memory usage, being over 12 times higher than Nest and over 1.5 times higher than Micro. Programs running on Java Virtual Machine are generally known for their higher than average memory usage levels.

### 4.1.3 Success rate

Successful response rate was perfect 100% for all frameworks except Micro. Micro failed to produce correct result from 1 to 8 times out of 10000 requests per test in high load test, resulting in 99.98% success rate. In low load, also Micro managed to respond successfully for all the request.

## 4.2 Universities application

The more real-life use-case simulating ”Universities” application was created to measure how the frameworks and languages affect performance and resource usage beyond the minimal headroom displayed by the “Hello-World” application. This application first authorized the user by JWT in request header and then returned filtered list of basic university info by country, which was defined in request query string.

### 4.2.1 Response times

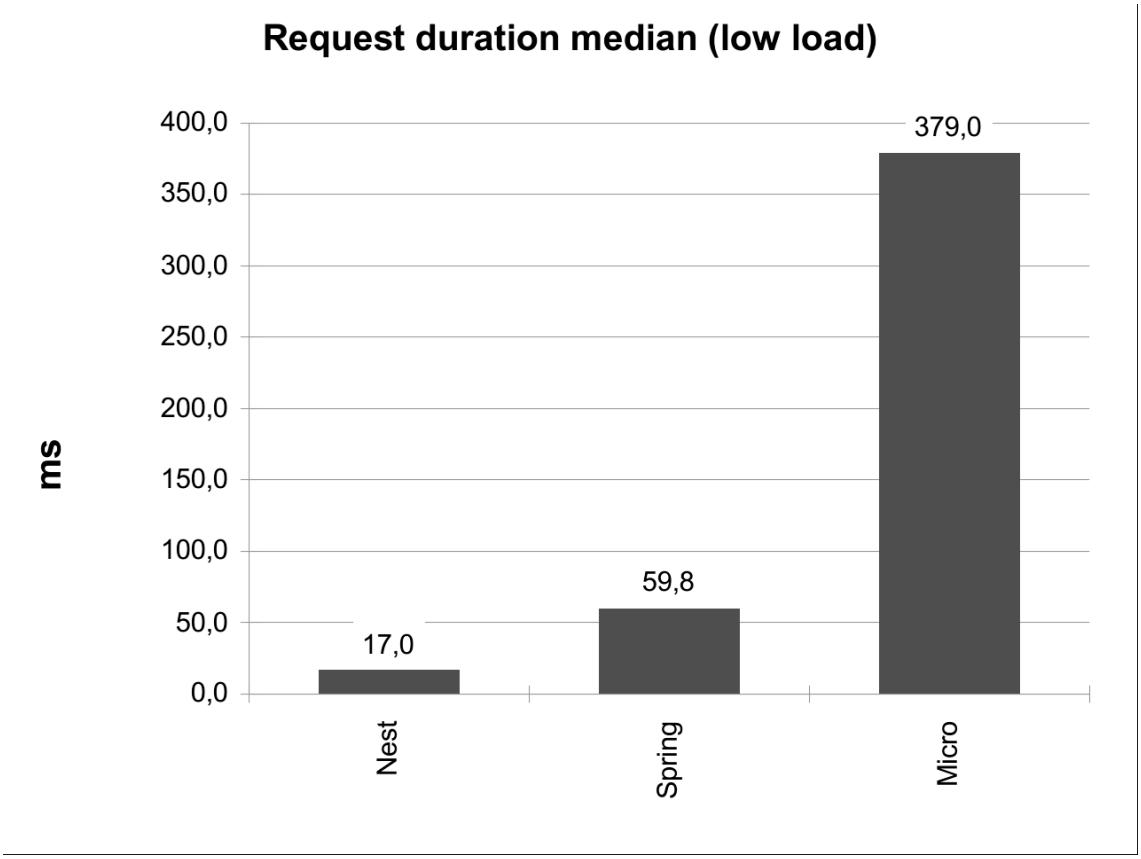


**Figure 14** Median of request duration for Universities application in low load stress test

The Universities application produced slower response times than the Hello-World one, as was expected, as all frameworks were at least twice as slow to respond. As shown in Figure 14, again Nest was the fastest one, Spring second and Micro last. But difference between the slowest and other two were greater than in Hello-World, as Micro required over 6 times more time to respond than Spring.

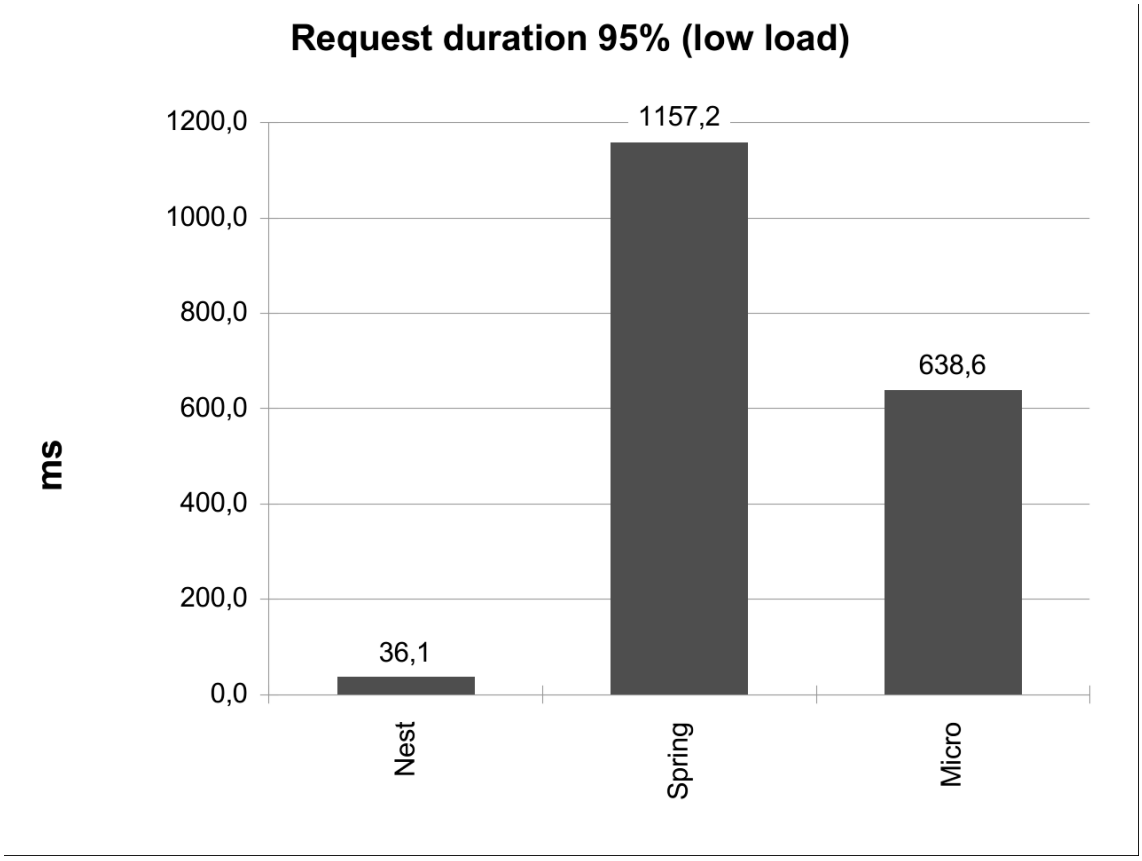


**Figure 15** 95th percentile request duration for Universities application in low load stress test

The 95th percentile results in Figure 15 showed that Spring required a lot more time to handle the first requests compared to the rest, its 95th percentile result was almost 20 times higher than median. This shows that Spring is very efficient in optimizing the subsequent requests, as the initial ones were twice as slow compared to Micro but median was significantly lower.

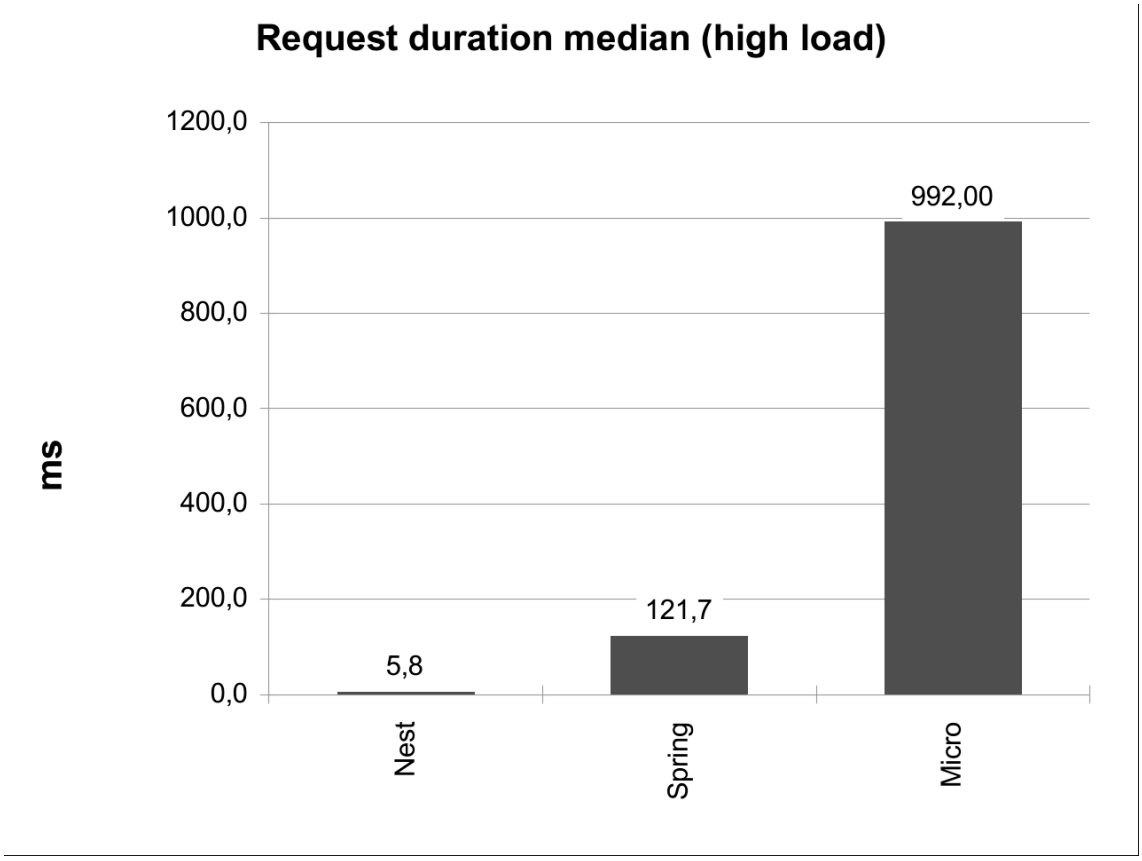


**Figure 16** Median of request duration for Universities application in high load stress test

In Figure 16 the high load test showed Micro noticeably slowing down when stressed with a lot of requests, median being almost one second per request. Spring also slowed down compared to low load with around twice as long response times. Nest in the other hand responded faster than in low load. Median of the response times between the fastest Nest and slowest Micro was significant, with Micro being 171 times slower.

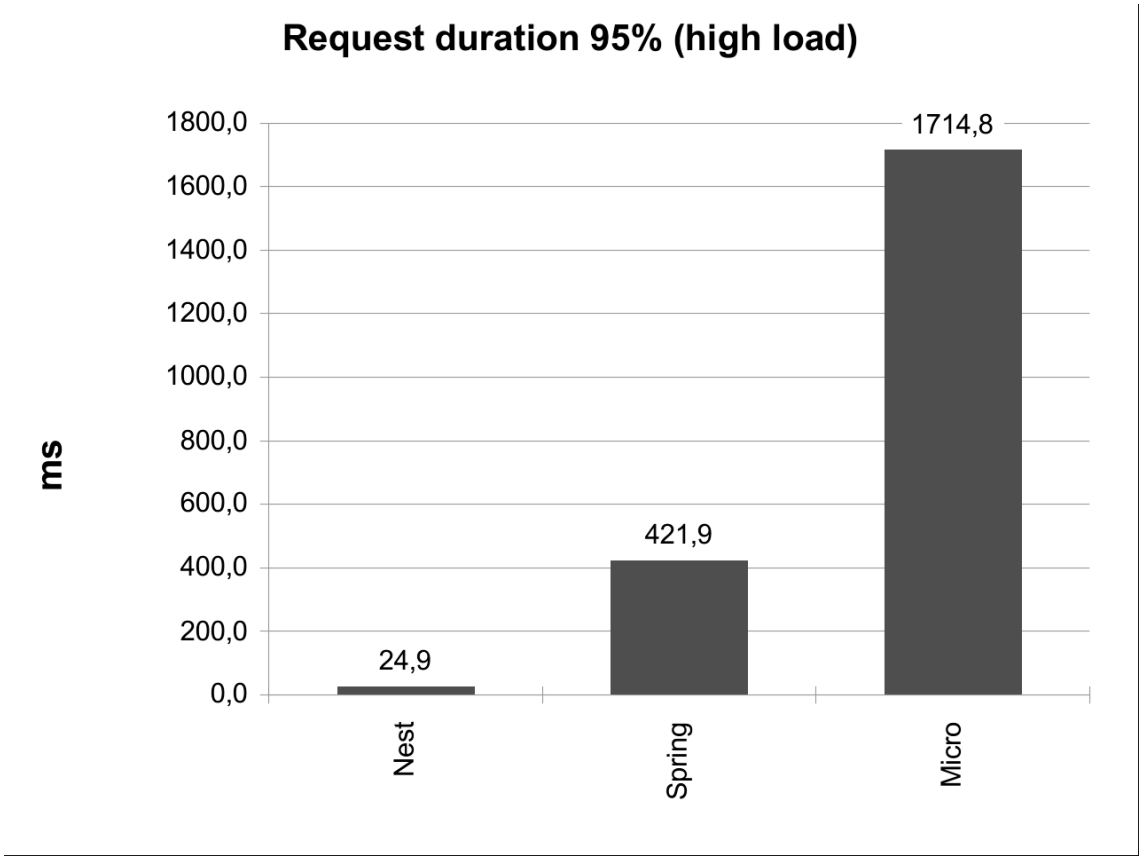


**Figure 17** 95th percentile request duration for Universities application in high load stress test

95th percentile results in Figure 17 again showed higher response times for Spring compared to medians, but here the wasn't nearly as big as in low load scenario, that's because the 95th percentile values include more of the responses after initial ones. One factor that might result in Nest being the fastest is its way to handle JSON -files, being able to read them very efficiently as the file contents are in JavaScript object notation format. The other frameworks have to read the files and convert the contents into their respective language data mapping format. This could be improved in the future by using CSV -format file instead.

4.2.2 Resource utilization

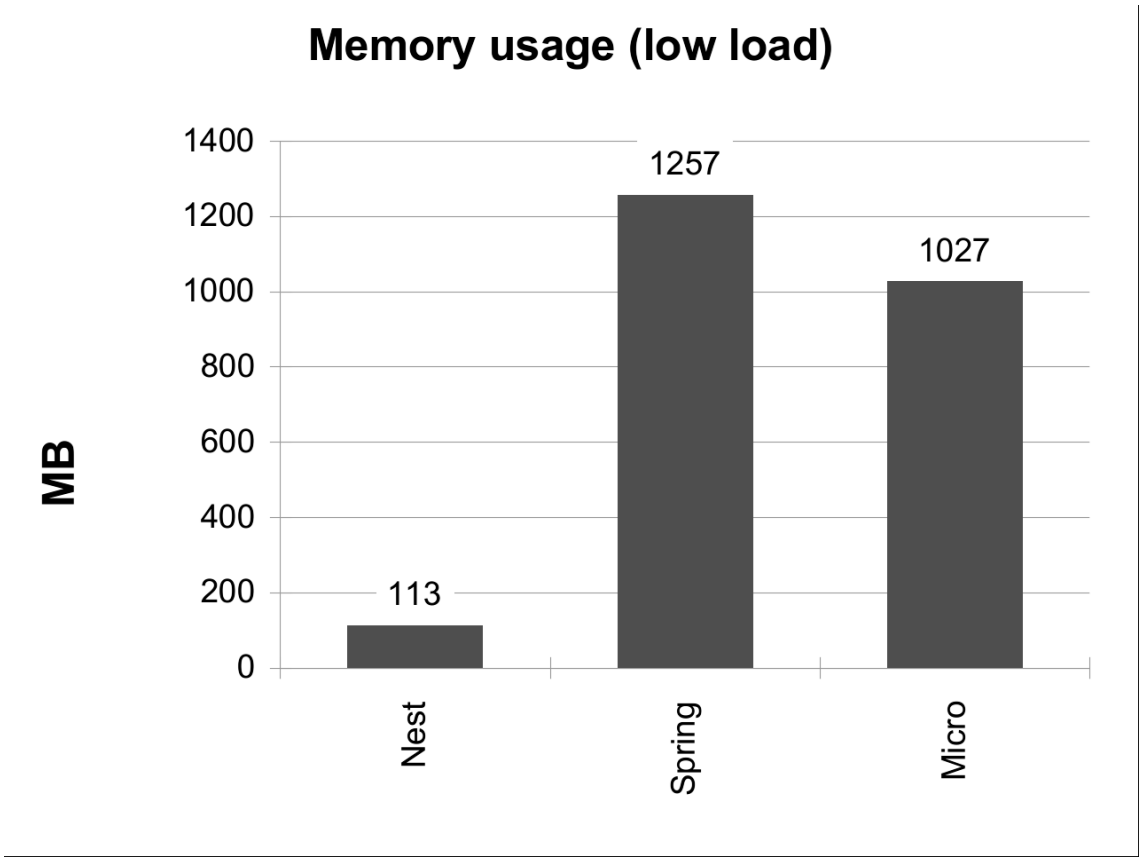


**Figure 18** Memory usage of Universities application in low load stress test

Resource usage of Universities application reflected the results of Hello-World application. As shown in Figure 18 and 19, here too Nest was the lightest one in both the memory and CPU usage, Spring used the most memory and Micro the most CPU. Although the low load scenario in Hello-World application as seen in Figure 10, showed Micro having the highest memory usage, here in the more complex Universities application the Spring uses the most memory.

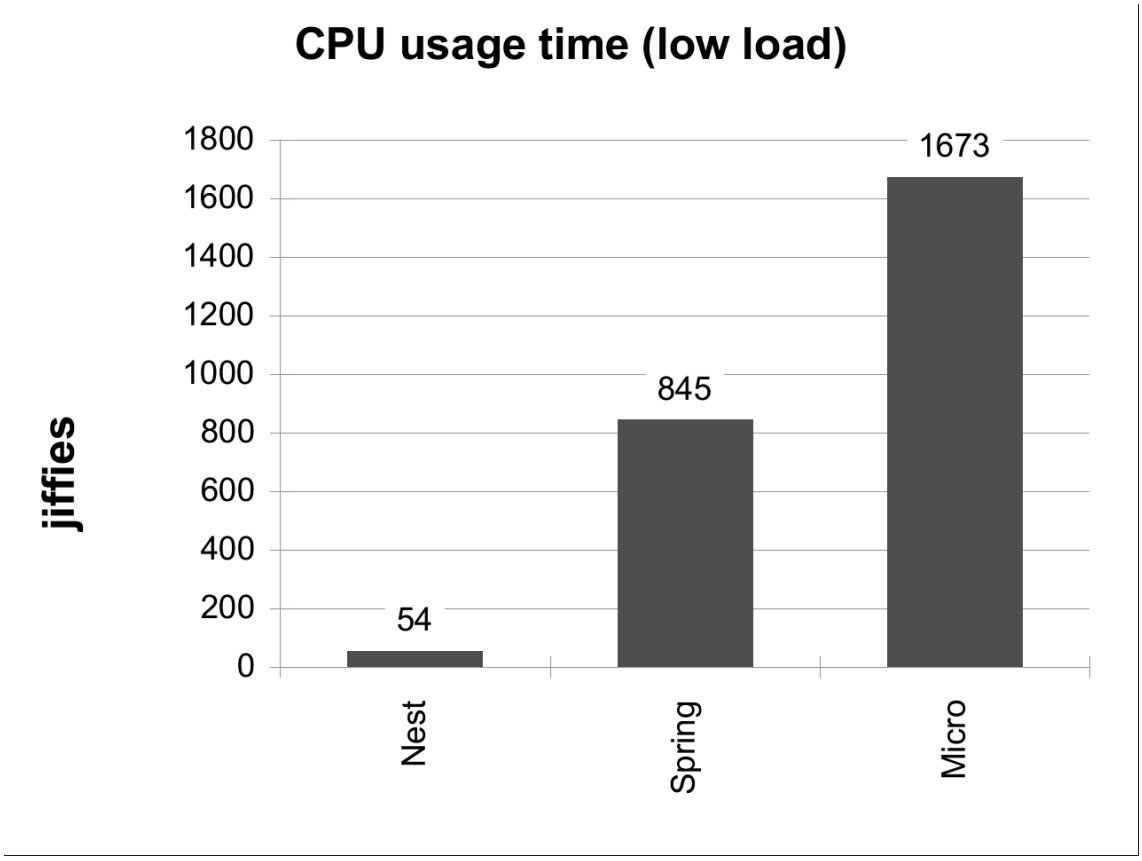


**Figure 19** CPU usage of Universities application in low load stress test

As shown in Figure 19, the most noticeable difference was that Micros CPU usage was nearly twice as high as Springs, although they were closer in Hello-World application. Difference between Nest and Spring was similar to Hello-World, but Nest was even leaner in comparison, utilizing around 11 times less memory and over 15 times less CPU than Spring.

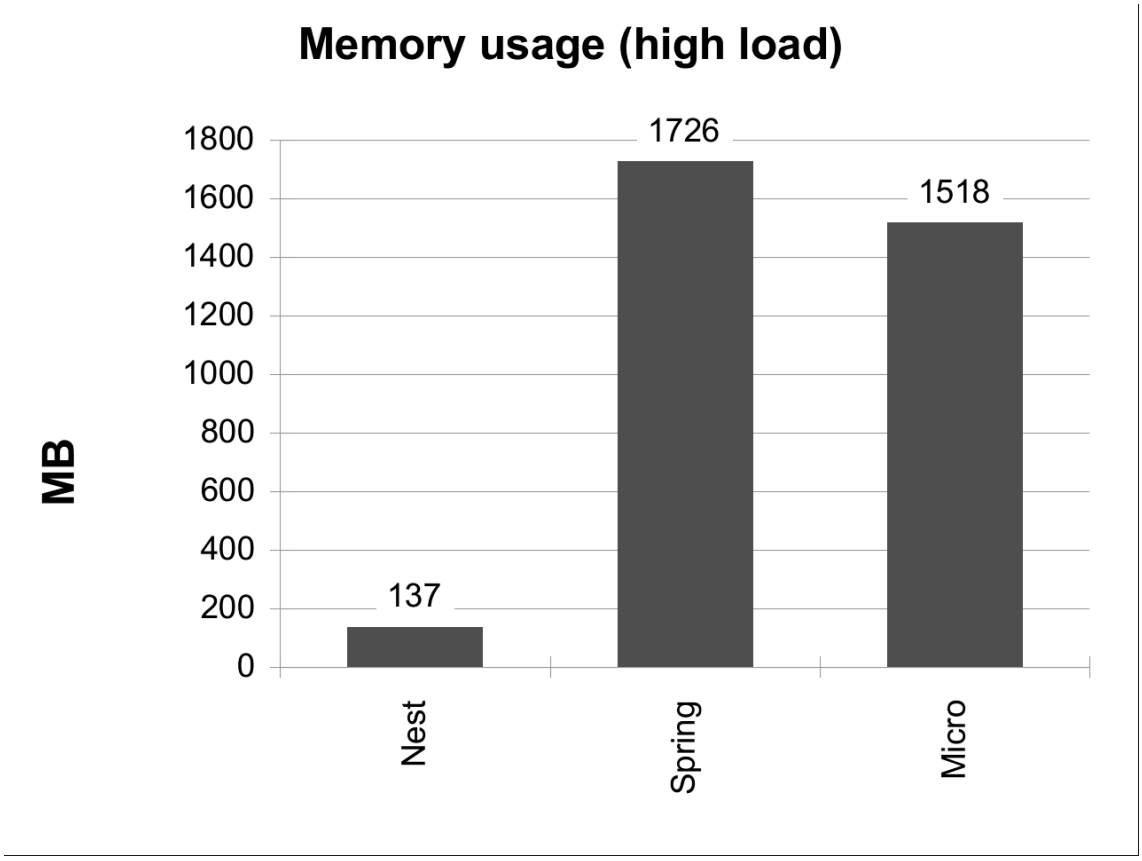


**Figure 20** Memory usage of Universities application in high load stress test

Memory usage in Figure 20 was comparable to low load stress tests in Figure 18, all the frameworks saw approximately 50% raise in values but the differences between the frameworks stayed similar. The low raise in memory usage was interesting, as the difference between low and high load scenario of CPU usage, as shown in Figure 21 and Figure 19 were significantly higher. The probable cause is that the application need to store the same values into memory over and over again, but being able to garbage collect the unused values away, making the memory consumption stay relatively low as the values only show maximum usage. Whereas the CPU usage values show the accumulated total processing time, making the values go up as more responses are produced.

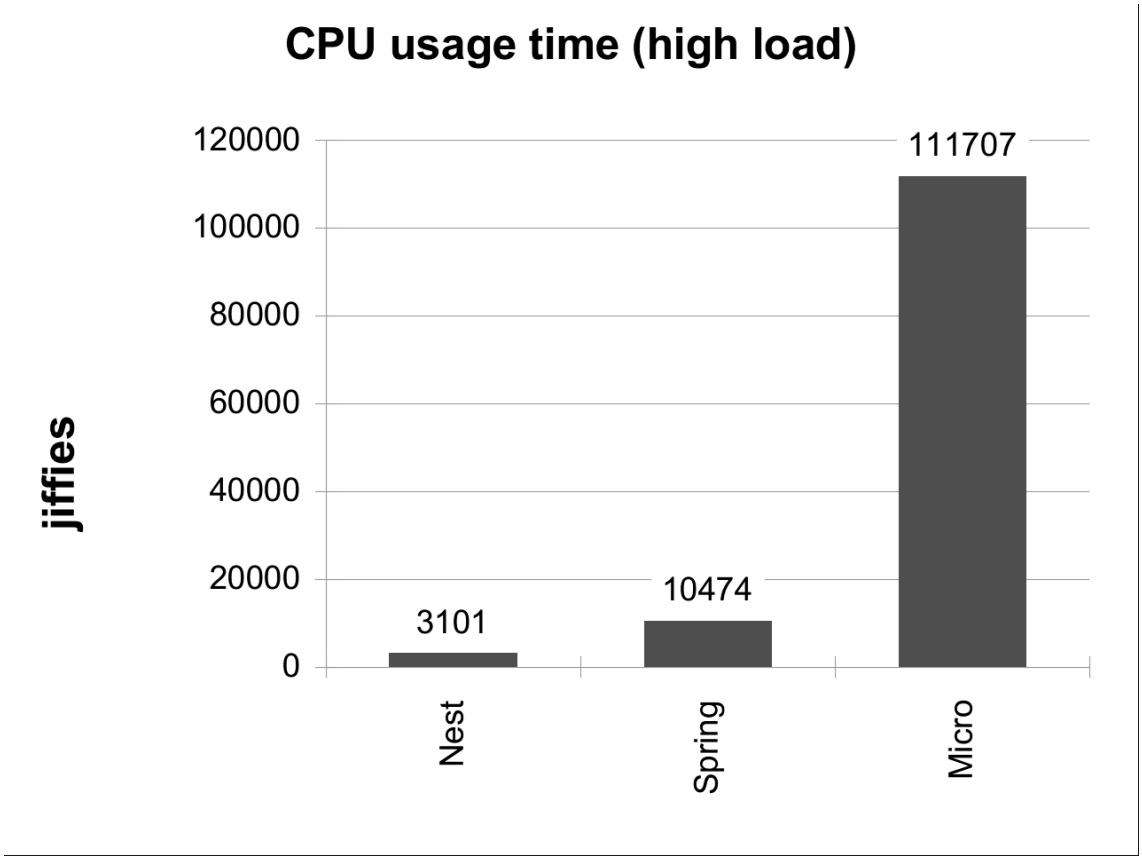


**Figure 21** CPU usage of Universities application in high load stress test

High load results were again similar to Hello-World application, except as shown in Figure 21 Micro produced very high CPU usage times. Micro utilized CPU over 10 times more than Spring, showing that the CPU was clearly bottlenecking the performance in high load scenario and causing the long response times. This hints that Micro might not use any caching to produce the responses as the CPU usage raises significantly compared to low load scenario.

### 4.2.3 Success rate

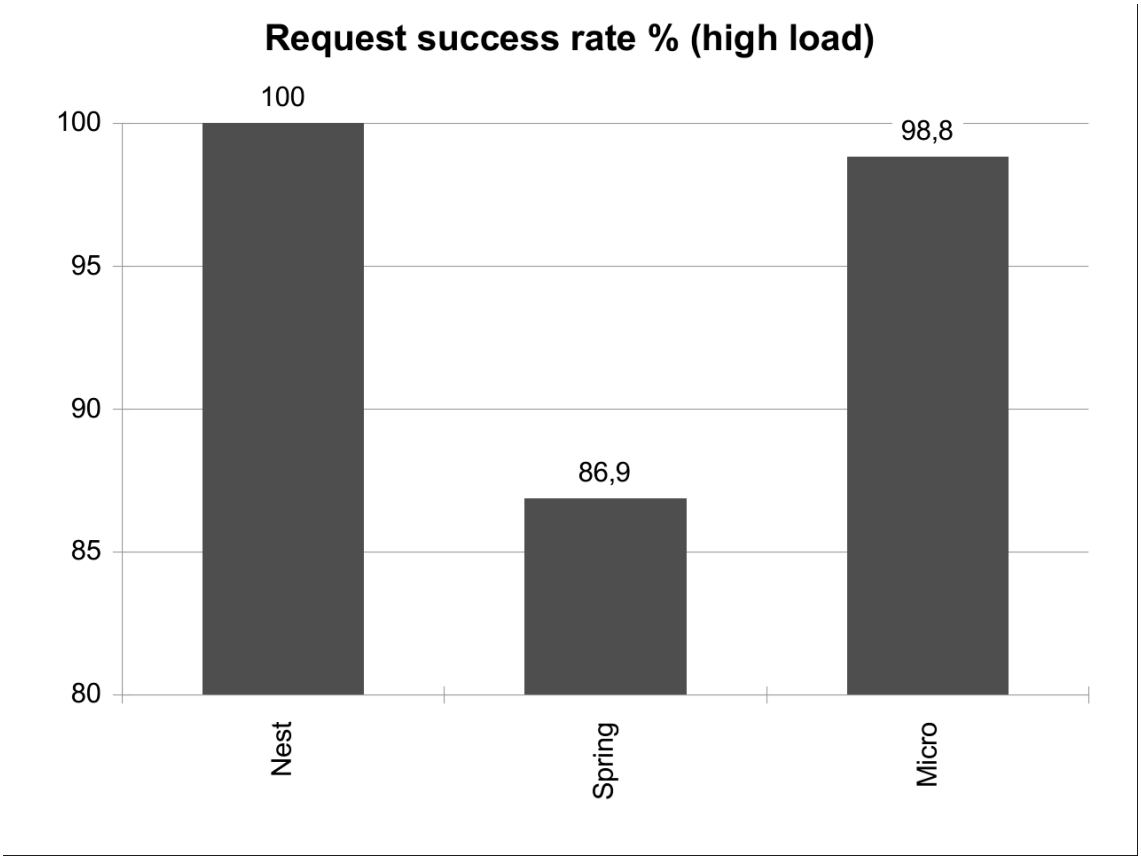


**Figure 22** Request success rate of Universities application in high load stress test

The low load request success rate was 100% for all the frameworks, but high load results in Figure 22 showed Spring failing to respond successfully in 13.1% of the requests. Nest was again 100% successful and Micros success rate was 98.8% although the requests were slow.

Upon investigating the issue with low success rate of Spring responses, it was determined that the most probable issue was the slowness of the first requests. In a study from Tuan, Beierle, Garzon and Mora the same issue was noticed when measuring response times of different Java Virtual Machine microservice frameworks. It was attributed to slow warm up time that JVM -applications require. [1] The results of this thesis also support that, as the other non JVM -frameworks didn't show as big difference between initial and subsequent requests. The high load stress test started firing 100 simultaneous requests with half a second delay in between them, if the runner didn't receive a successful response in that half second period it proceeded to fire next request

and marked the slow one as a failure although the request eventually received a successful response. The same reason applies for Micro. This needs to be kept in mind when reading the results, as the sub 100% success rate doesn't necessarily mean failing to produce successful result, but that the successful result wasn't produced in short enough time.

# 5 DISCUSSION

Performance analysis showed that there were very noticeable differences between the frameworks used in this study. Especially JavaScript framework Nest showed very good results in both the response times and resource utilization, by being clear number one in all the measured values, even beating the other frameworks by over 10 times performance difference in high load stress tests. Second best framework by performance was the Java based Spring, which measured second best in all categories except total memory consumption in which it used more memory than the other frameworks. Slowest and heaviest was the Go -language based Micro, especially its response times were much slower than the other frameworks.

A correlation between the framework performance and resource utilization was consistent, although reversed, the fastest response times required the least resources. The Nest applications used the least memory and CPU while producing the fastest results. Spring that produced the second fastest response times were often second in resource utilization, although it required the most memory, especially in high load tests. Micro was the slowest of all three and used the most CPU across all tests, with significantly higher CPU usage in high load stress test of Universities application where its usage was over 10 times higher than the other two.

The simple Hello-World application was designed to measure the overhead created by the frameworks. This method nicely showed how much different design philosophy of the framework affected performance, especially in Micros case. Micro differed from the other two frameworks by being more of a full fledged application building ecosystem than a modular skeleton like Nest and Spring. Most of the processing time and power went to the core services of Micro instead of producing the “Hello World!” string, there were 10 services running in Micro instance where the other frameworks only had two or three. Some of the Micros slowness could also be explained by usage of RPC data transfer protocol between services instead of simpler TCP calls the other two used. More in-depth study is required to better analyze what affects the performance of Micro. One major notion to understand is the maturity of the frameworks. Nest being based on Node.js and especially Spring have years of development behind them, but Micro was released only approximately three months before implementation of the applications in this study. Micro has continually received major updates after its release and will most probably gain

significant performance improvements in the future, readers of this study should keep this in mind when comparing the results, especially if reading this after 2021.

Interesting observation from the results was the fact that the frameworks programming language performance didn't have a correlation with the frameworks performance. From "The Computer Language Benchmarks Game" [28] website that includes different mini programs to measure programming language performance, it can be seen that there aren't significant performance differences between the languages in this study. Some languages are better in certain types of programs than others, but none demonstrate the tenfold difference as this study did. Micros Go is often the fastest language of the three when comparing the raw speed and efficiency.

The sample applications developed for this study were each done using the official or most widely used documentation of the frameworks. This meant that the applications were not carbon copies of each other, but instead aimed to reflect applications the developers would create to achieve the same end result. This also probably had some affect on performance, for example Spring implementation having the Eureka service discovery service to provide information for the gateway about the network locations of the available services. Also Micro had service discovery as a built-in core service but Nest didn't, instead using hard-coded values for service locations.

In addition to performance, also developer experience plays strong part in choosing the correct framework. This is a subjective analysis of the thesis author and only covers the experience related to creating applications for this study. The most easy to find and understand documentation was for Nest, especially the official documentation covered all the needed topics in enough detail. Nest being a JavaScript frameworks, the container building and deployment process was the most simple out of the three. Based on that, the Nest framework would be most recommended for more junior developers and smaller projects but that doesn't mean it could not be used for larger enterprise applications, the performance certainly backs that up. Spring being the most known enterprise-scale framework it had a lot of documentation available, but it was more difficult to find appropriate documentation for the used version. The official documentation was more of an API documentation, instead of thorough overview with examples. Springs nature of being very modular ecosystem with separate subprojects for handling specific use-cases made it harder to get started and required some understanding of the whole ecosystem to create the desired application in a proper way. In the other hand, being very widely used and developed solution it would be safest framework for large enterprise scale

applications that require support for several external technologies and integrations. Because of the young age of Micro, it was difficult to find good documentation regarding aspects that wasn't covered in official documentation, but the official Slack channel offered help to figure out those. The official documentation at the time was quite shallow, it covered creating a simple application and running it, but not much more. This will also most probably improve by time. One insignificant sounding but notable problem with working on Micro was its name. The word "Micro" resulted in search results including general information about microservices or being synonym for very small, instead of the framework. The most troublesome characteristic of Micro was its focus to be used in cloud. In real-life scenario there's no problem with that as applications are generally run in cloud, but for this study each application needed to be runnable in a Docker environment on a local machine. This forced the whole Micro application to be run in single container instead of running each service in separate container as the other frameworks.

In order to improve the results of this study, following future research improvements should be considered. Although the general guideline in this study for creating the sample applications was to follow official documentation and best practices of each framework, some changes to the applications would improve the comparability of the results. Most important changes would be adding a service discovery service for Nest implementation, as both Spring and Micro had one and second would be using the same data transfer method/protocol across all the frameworks, preferably RPC -protocol, as that is the one Micro uses. The Universities application using a JSON -format list might have given an edge to Nest framework, using CSV -file could be more comparable across the framworks. It would be interesting to see how much implementing the Spring applications in reactive way using Project Reactor would affect the performance, as in this study the Spring application was created in more common way using Spring Boot and Cloud projects. The underwhelming results of Micro were probably significantly affected by Micro being in very early stage of development compared to the other two frameworks, updating Micro application and re-running the tests in the future might give more comparable results. The last but not least improvement would be to add more frameworks, especially frameworks that have been developed in performance as a main feature to see if Nest would still hold up against those.

# 6 CONCLUSION

This study presented the performance differences of three microservice frameworks (Nest, Spring and Micro) measured in request execution times, memory consumption and CPU utilization. For each framework, two separate applications were created, one simple application that only returned string “Hello World!” when requested and other more real-life simulating application that also authorized the requests and then returned list of basic university info by country provided in the request. The application performance was tested by firing requests to the application endpoint using stress test tool and measuring the response times, after the stress test the memory and CPU usage values were recorded.

Results showed that applications created with different frameworks produced surprisingly large performance differences. Nest were the most efficient in all categories. Nests response times were often many times quicker than the second fastest framework, being over 20 times faster in the Universities application when measured with high load. The Hello-World application result differences weren’t as massive as the Universities one, but still Nest was clearly faster. Applications created with Spring framework were reliably second fastest and Micro was the slowest in all tests. Springs response time medians were damaged by the slow initial responses, as the subsequent responses were close to those of Nest. Memory and CPU consumption values followed the same trend as response times, Nest was the most efficient in all the test, Spring being often second and Micro last, although Spring showed the highest memory usage of all three in the Universities application. One notable observation from the results was that the Nest implementation utilized around ten times less resources than the other two, but still produced the fastest responses.

Some causes for the significant performance differences across the frameworks would be the fact that the applications weren’t created as carbon copies of each other, but instead created following the official documentation and best-practices of each framework. This resulted in slight differences between the architecture of applications. Main attributes for the Micro being last in most of the categories would be its young age compared to others and it being more of a full application creation ecosystem than a tool for creating microservices.

# REFERENCES

[1] Dinh Tuan, Hai & Mora, Maria & Beierle, Felix & Garzon, Sandro. Development Frameworks for Microservice-based Applications: Evaluation and Comparison. In 2020.

[2] Delmas C. Performance of Microservices frameworks [Internet]. 2016 [cited 2021 May 31]. (A modest developer's blog). Available from: https://cdelmas.github.io/2016/06/20/Performance-of-Microservices-frameworks.html

[3] Engel A. Comparison of Microservice Frameworks with a Streaming Example [Internet]. 2019 [cited 2021 May 31]. Available from: https://medium.com/swlh/comparison-of-microservice-frameworks-with-a-streaming-example-6bfe284a66a

[4] GitHub - networknt/microservices-framework-benchmark: Raw benchmarks on throughput, latency and transfer of Hello World on popular microservices frameworks [Internet]. [cited 2021 May 31]. Available from: https://github.com/networknt/microservices-framework-benchmark

[5] Brown K. Beyond buzzwords: A brief history of microservices patterns [Internet]. 2018 [cited 2021 Mar 21]. (IBM Developer). Available from: https://developer.ibm.com/depmodels/microservices/articles/cl-evolution-microservices-patterns/

[6] Fowler M, Lewis J. Microservices [Internet]. [cited 2021 Mar 21]. (martinfowler.com). Available from: https://martinfowler.com/articles/microservices.html

[7] Villamizar M, Garcés O, Castro H, Verano M, Salamanca L, Casallas R, et al. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In 2015. p. 583–90.

[8] Jayawardana Y, Fernando R, Jayawardena G, Weerasooriya D, Perera I. A Full Stack Microservices Framework with Business Modelling. In 2018. p. 78–85.

[9] Risberg T. Spring Framework 1.0 Final Released [Internet]. 2004 [cited 2021 Apr 5]. Available from: https://spring.io/blog/2004/03/24/spring-framework-1-0-final-released

[10] Maple S, Binstock A. JVM Ecosystem report 2018 - About your Platform and Application | Snyk [Internet]. 2018 [cited 2021 Apr 5]. Available from: https://snyk.io/blog/jvm-ecosystem-report-2018-platform-application

[11] Spring Boot Documentation [Internet]. [cited 2021 Apr 5]. Available from: https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html

[12] Spring Cloud [Internet]. [cited 2021 Apr 5]. Available from: https://spring.io/cloud

[13] V8 Documentation [Internet]. [cited 2021 Apr 5]. Available from: https://v8.dev/docs

[14] Documentation | NestJS - A progressive Node.js framework [Internet]. [cited 2021 Apr 5]. Available from: https://docs.nestjs.com

[15] Micro [Internet]. [cited 2021 Apr 5]. (Micro). Available from: https://micro.mu

[16] GitHub - micro/micro [Internet]. [cited 2021 Apr 10]. Available from: https://github.com/micro/micro

[17] GitHub - Spring [Internet]. [cited 2021 Apr 10]. Available from: https://github.com/spring-projects

[18] GitHub - nestjs/nest [Internet]. [cited 2021 Apr 10]. Available from: https://github.com/nestjs/nest

[19] Kurmi A. Top 10 Microservices framework for 2020 [Internet]. 2021 [cited 2021 Apr 10]. (Medium). Available from: https://medium.com/microservices-architecture/top-10-microservices-framework-for-2020-eefb5e66d1a2

[20] Flask Documentation [Internet]. [cited 2021 Apr 10] Available from: https://flask.palletsprojects.com/en/1.1.x/

[21] Introduction to gRPC [Internet]. [cited 2021 Apr 16]. (gRPC). Available from: https://grpc.io/docs/what-is-grpc/introduction/

[22] JWT.IO - JSON Web Tokens Introduction [Internet]. [cited 2021 Apr 17]. Available from: http://jwt.io/introduction

[23] GitHub - Hipo/university-domains-list: University Domains and Names Data List & API [Internet]. [cited 2021 Apr 17]. Available from: https://github.com/Hipo/university-domains-list

[24] Make - GNU Project - Free Software Foundation [Internet]. [cited 2021 Apr 18]. Available from: https://www.gnu.org/software/make/

[25] Docker - Runtime metrics [Internet]. 2021 [cited 2021 Apr 18]. (Docker Documentation). Available from: https://docs.docker.com/config/containers/runmetrics/

[26] Lönn R. Open source load testing tool review 2020 [Internet]. 2020 [cited 2021 Apr 19]. (Open source load testing tool review 2020). Available from: https://k6.io/blog/comparing-best-open-source-load-testing-tools

[27] Laaber C, Schneuner J, Leitner P. Performance testing in the cloud. How bad is it really? [Internet]. 2018. Available from: https://peerj.com/preprints/3507.pdf

[28] The Computer Language Performance Game [Internet]. [cited 2021 May 27]. Available from: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

# Appendix 1. Source codes.

Source codes can be found from: https://github.com/heikkikesa/microservice-performance-comparison (9.6.2021)