Master's thesis

Master's Programme in Computer Science

A Comparative Study of Application Performance and Resource Consumption between Monolithic and Microservice Architectures

Youqin Sun

May 13, 2024

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Youqin Sun

Työn nimi — Arbetets titel — Title

A Comparative Study of Application Performance and Resource Consumption between Monolithic and Microservice Architectures

Ohjaajat — Handledare — Supervisors

Prof. Mika Mäntylä

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	May 13, 2024	53 pages, 25 appendix pages

Tiivistelmä — Referat — Abstract

Monolithic and microservice architectures represent two different approaches to building and organizing software systems. Monolithic architecture offers various advantages, such as simplicity in application deployment, smaller resource requirements, and lower latency. On the other hand, microservice architecture provides benefits in aspects including scalability, reliability, and availability. However, the advantages of each architecture may depend on various sectors especially when it comes to application performance and resource consumption.

This thesis aims to provide insights into the differences in application performance and resource consumption between the two architectures by conducting a systematic literature review on the existing literature and research results in this regard and performing a benchmarking with various load tests on two applications of identical functionalities but using the two abovementioned different architectures.

Results from the load tests revealed the applications in both software architectures delivered satisfactory outcomes. However, the test outputs indicated the microservice system outperformed by a high margin in nearly all test cases in aspects including throughput, efficiency, stability, scalability, and resource effectiveness. Based on the research outcomes from the reviewed literature, in general, monolithic design is more efficient and cost-effective for simple applications with small user loads. While microservice architecture is more advantageous for large and complex applications targeting high traffic and deployment in cloud environments. Nevertheless, the overall research results indicated both architectures have strengths and drawbacks in different aspects. Both architectures are used in many successful instances of applications. The differences between the two architectures in application performance and resource effectiveness depend on various factors, including application scale and complexity, traffic load, resource availability, and deployment environments.

ACM Computing Classification System (CCS)
General and reference → Document types → Surveys and overviews
Applied computing → Computers in other domains

Avainsanat — Nyckelord — Keywords

software architecture, monolith, microservice, performance, resource consumption

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction

Two of the main software architectures adopted by enterprise application development nowadays are monolith and microservice (Blinowski et al., 2022). In a software application developed in a monolithic architecture, all components, modules, and functionalities are packaged and deployed as a single, indivisible unit, including user interface, data access, and business logic. A monolithic application normally functions as an independent system and does not rely on other applications (Gos and Zabierowski, 2020). Microservice architecture is a service-oriented architecture in which an application is broken down into multiple smaller, independent services. Typically each service is responsible for a specific business function that can be developed and deployed independently(Namiot and sneps-sneppe, 2014).

Traditionally software systems are developed in a monolithic architecture which offers various advantages in many aspects, such as resource efficiency, cost-effectiveness, and faster response time (De Lauretis, 2019). Nonetheless, components in an application developed with a monolithic architecture cannot be scaled independently; therefore, resource usage may not be optimized. Monolithic architecture is also known to have a single point of failure, which affects application reliability and fault tolerance (Poonam et al., 2023). Microservice architecture is known to have advantages in scalability, reliability, and fault tolerance (Blinowski et al., 2022). In recent years, the software industry has been experiencing a trend of moving towards the microservice architecture (De Lauretis, 2019), as shown in Figure 1.1 (Stack Overflow Trends) and Figure 1.2 (Google Trends). More applications have been built in microservice architecture and a growing number of companies seek to migrate monolithic applications to microservices (Velepucha and Flores, 2021). That being said, microservice architecture is not flawless. All the subsystems in an application of microservice architecture need to be separately deployed, requiring additional infrastructure resources such as memory, CPU, and network bandwidth. In addition, microservices usually communicate with each other over a network, which introduces delays and possible points of failure.

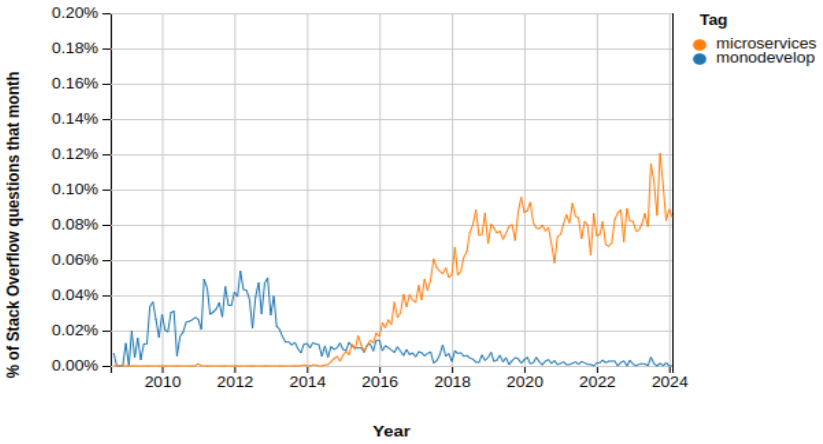


Figure 1.1: Stack Overflow Trends

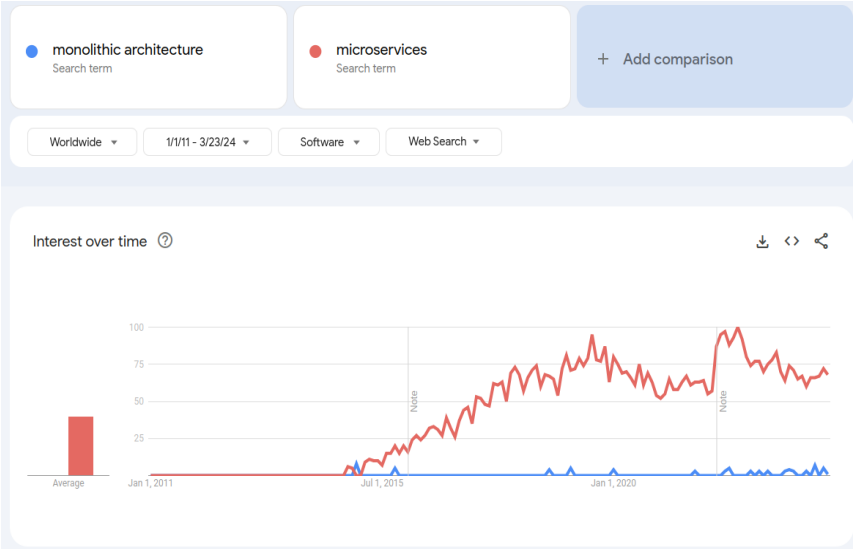


Figure 1.2: Google Trends

The two architectures may show drawbacks and advantages in many different aspects. This thesis aims to study the difference between the two software architectures from the perspectives of application performance and resource consumption. As the research methods, a systematic literature review was performed to review existing literature and research results in this regard. In addition, a benchmarking process was carried out by performing load testing on two applications in cloud environments on the Google Kubernetes Engine. The research efforts focused on the following research questions(RQs):

1. What are the differences in resource consumption between applications in monolithic and microservice architectures?

2. What are the differences in application performance between applications in monolithic and microservice architectures?

The rest of this paper is organized as follows: Section 2 provides background knowledge about the two software architectures studied in this paper, namely monolithic architecture and microservice architecture, and compares the differences between the two architectures. An introduction to cloud computing is also included in this part, as it is an important sector in the shift from monoliths to microservices. An overview of load testing, the applications under test, the test platform, and the test framework are also presented in this section. Section 3 describes the methods that were used in this thesis: 3.1 a systematic literature review and 3.2 a benchmarking process with load testing. The subsections under 3.1 describe the literature review data sources, and research procedures. The subsections under 3.2 introduce the test environments, and test cases for the benchmarking. Section 4 presents the results obtained from the literature review and benchmarking experiments with detailed statistics, tables, and graphs. Section 5 discusses the results of Section 4. Finally, Section 6 summarizes the findings obtained in this research efforts of the thesis.

2 Background

2.1 Software Architecture

Software architecture is defined as the "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" in the ISO/IEC/IEEE 42010 Systems and Software Engineering Architecture Description. It can be interpreted as the guidelines and principles that define how the components of a system are organized and structured as well as the relationships among the components (Kazman et al., 1994). The software architecture of a system includes all its components, attributes, properties, and structures, as well as the relationships among them (Bengtsson and Bosch, 1999). It should take into account both the external context and the internal logic of the system (Dobrica and Niemela, 2002). The software architecture is responsible for defining the technical and operational requirements of the system. It plays an important part in the application's performance metrics, including efficiency, scalability, reliability, and cost-effectiveness (Gos and Zabierowski, 2020). Nowadays, monolith and microservice are two of the primary software architectures used in enterprise application development (Blinowski et al., 2022).

2.1.1 Monolithic Architecture

Monolithic software architecture is a traditional approach to building software applications. In this architecture, an entire application, including all its functionalities and features, is developed in one code base and typically uses the same technology stack. For a software system developed in a monolithic architecture, all its components and modules are tightly coupled and integrated into a whole package. Everything needed to run the application is packaged as one unit and deployed as a single executable (Ponce et al., 2019). A typical monolithic application is designed with the well-established three-tier model, which consists of distinct layers for user interface (UI), business logic, and data access, as illustrated in Figure 2.1. All the data related to the application is kept in a single database (Microsoft, 2024).

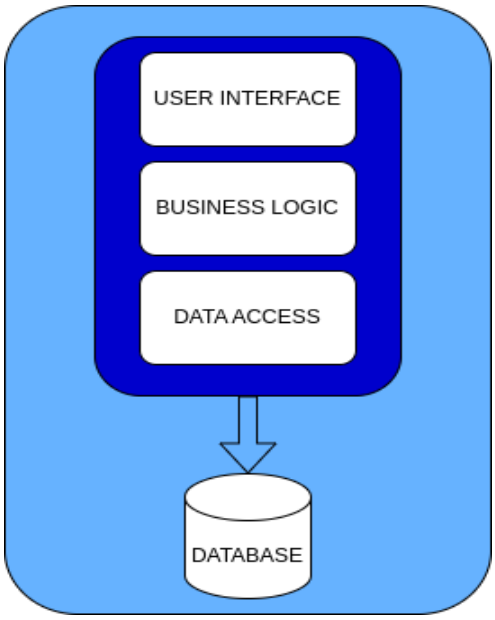


Figure 2.1: monolith architecture

Monolithic architecture provides many benefits over microservice architecture especially for smaller projects and prototyping efforts, despite being more traditional. To some extent, one code base in this architecture simplifies development, maintenance, testing, and debugging processes, especially during the early stages of application development (De Lauretis, 2019). Since there is only one database, data consistency is better guaranteed. All communications between subsystems are internal, which makes them more efficient and free from issues with inter-process communication (IPC). Less overhead also means higher efficiency in user request handling. The feature of being a single service makes it easier to build and deploy (Blinowski et al., 2022), as it eliminates the complexity of managing various services and coordinating their deployment schedules, and reduces errors during deployment. In addition, projects of smaller size, lower traffic, and modest scalability requirements can be more cost-effective as they usually consume fewer resources.

Monolithic architecture has many disadvantages as well. Since there is only one code base, it means that the entire application has to be rebuilt and redeployed for every code change. Furthermore, as the system grows in size and complexity, application development and maintenance may become more challenging and less efficient (Ponce et al., 2019). All components are tightly coupled and have hidden dependencies on each other, increasing the chance of vulnerabilities. If even one component's source code contains errors, the entire application may become unavailable. A larger application needs a longer time to start up, build, and deploy, which inevitably slows down deployment and delivery efforts and leads

to lower productivity. Another issue with many monolithic applications is the difficulty of scaling individual components based on market or business demands (Velepucha and Flores, 2021). The need to scale the entire application during high user traffic can cause higher resource consumption compared to distributed systems.

2.1.2 Microservice Architecture

In recent years, cloud technologies have become more advanced and cloud platforms are widely available. In addition, modern applications are becoming more sophisticated and the number of features continues to increase, making it difficult to achieve with a monolithic architecture (Microsoft, 2024). As a result, microservice architecture has gained prominence in the software industry (Zhang et al., 2019).

Microservices Architecture (MSA) divides an application into an ecosystem of loosely coupled and independently functional subsystems that communicate with each other over the network using lightweight protocols. Each business functionality forms a microservice with a single responsibility. In most cases, each microservice contains its own structure layers for user interaction, application logic, the back-end, and even the database (Zhang et al., 2019).

As shown in Figure 2.2, each microservice functions as an independent unit, therefore it can be developed, tested, and deployed separately. Loose coupling brings higher fault tolerance. Failure of one component does not necessarily affect the functionalities of other parts, which in turn increases application availability (Almeida and Silva, 2020). Microservices are small and lightweight, making them easier for development, building, and deployment compared to large and complex monolithic applications. Some other advantages include higher efficiency in development and delivery, shorter time to market, improved reliability and scalability, less complex maintenance, and easier updating procedures (Blinowski et al., 2022). Due to its benefits in various areas, a significant number of top-tier Internet services, including Netflix, Amazon, and eBay have adopted this technology (Gos and Zabierowski, 2020).

That being said, the microservice architecture is not flawless, and several drawbacks have been identified in recent research and industrial experiences. Given the distributed nature of microservices, a functionality may need to interact with multiple subsystems while each has its own separate data handling logic, which may result in difficulties managing transactions, maintaining data consistency, and communicating amongst services inside

networks (Zhang et al., 2019). Applications developed with microservice architecture consist of numerous independent units that need to be deployed and managed separately. As more features are added to the application, the quantity of microservices increases, which may cause the network's response times and latency to increase (Velepucha and Flores, 2021). A larger number of interfaces also makes it more complex to maintain the decoupling. New components must be carefully designed to avoid disturbing existing functionalities.

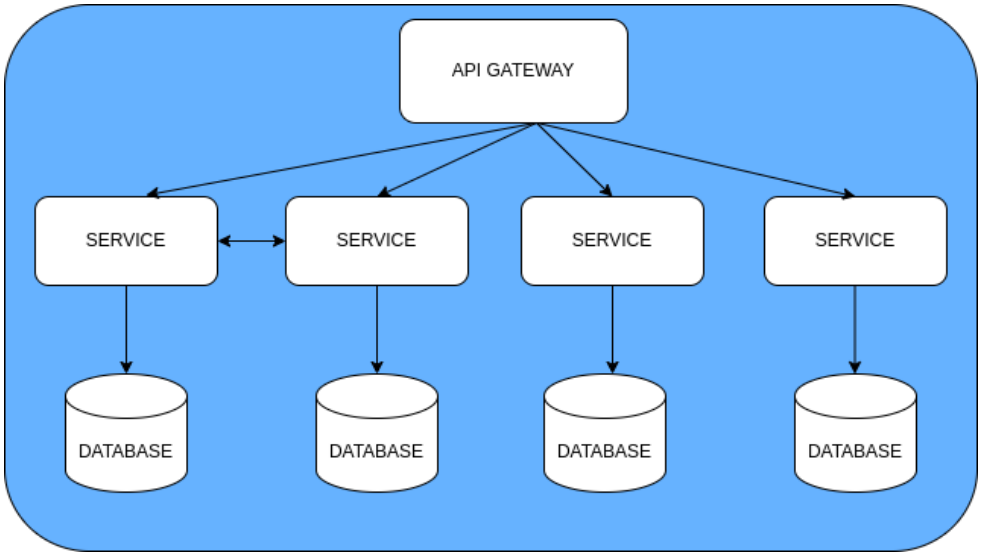


Figure 2.2: microservice architecture

2.2 Cloud Computing

Microservice software architecture has become increasingly popular in recent years and cloud technologies have significantly contributed to the popularity of microservice architecture. The fundamental infrastructure and tools required for creating, implementing, and orchestrating microservices at scale are provided by cloud technologies. Businesses can adopt microservices architecture in the cloud-native era to increase agility, scalability, resilience, and efficiency (OpenAI, 2024). The benefits of using the cloud to improve application performance, such as scalability, availability, and reliability, and reduce costs on maintenance and infrastructure have contributed to the popularity of microservice architecture and the trend of migrating legacy monolithic applications to microservices (Esposito et al., 2016).

How is "cloud" defined in this context? The word "cloud" refers to a worldwide network of servers with various distinct purposes. Instead of being a physical object, the cloud

is a massive global network of isolated servers that are interconnected over the internet and designed to function as a unified ecosystem (A. Microsoft, 2024). A wide variety of resources, physical and virtual, are gathered in large pools and offered as services with easy accessibility (Gong et al., 2010) through mainly three kinds of cloud platforms, i.e., public cloud, private cloud, and hybrid cloud (Jadeja and Modi, 2012). The term "cloud computing" refers to the on-demand provision of computer system resources, particularly processing power and data storage, without the need for direct user supervision (Montazerolghaem et al., 2020). Based on Microsoft cloud service Azure, in cloud computing, the internet, or "the cloud", is used to provide resources and services such as servers, storage, networking, databases, software, and analytics.

Functions in large clouds are frequently dispersed among several sites, each serving as a data center. Optimizing the usage of distributed resources, combining them to produce faster throughput, and addressing large-scale computing challenges are the primary objectives of cloud computing (Jadeja and Modi, 2012). Without having to invest in expensive hardware upgrades, cloud computing allows businesses to swiftly and effectively scale their IT resources up or down. This can facilitate an organization's ability to react swiftly to shifting market conditions and business requirements. Organizations can lower their IT infrastructure expenses and boost operational effectiveness by shifting IT resources to the cloud. Additionally, the pay-per-user policy of cloud computing enables businesses to just pay for the resources they utilize as opposed to purchasing costly hardware and software licenses (Islam et al., 2023).

The industry of cloud computing has experienced enormous expansion throughout the world in the past few decades. The way to store, process, and access data has already evolved as a result of cloud computing, and this trend is predicted to have a big impact on information technology going forward (Islam et al., 2023). Amazon cloud platform Amazon Web Services (AWS) claims that cloud computing is being used by enterprises of all shapes and sizes for a wide range of use cases, including email, virtual desktops, disaster recovery, software development and testing, big data analytics, and online apps that interact with customers. A wide variety of applications have been made available through the cloud, including data storage, document processing, service hosting, and video streaming (Ray, 2018). The majority of businesses anticipate that the cloud will replace on-premises infrastructure as the venue where applications are hosted. When it comes to application deployment in the cloud, microservices are a better choice compared to monolithic applications Linthicum, 2016). However, based on the literature research

performed in this thesis, very limited research results have been established on which software architecture is a better choice for running applications in the cloud and the exact advantages of each architecture depend on various factors.

2.3 Load Testing

Load testing was used as the main method for the benchmarking process of this thesis. The technique of simulating numerous users accessing a software program concurrently to predict the expected usage of the application is known as load testing. It is a testing strategy used by the professional software testing community in a variety of ways (Wescott, 2013). The purpose of load testing is to ascertain how a system behaves both at peak and average loads. It is essential for finding performance limitations, comprehending system capacity, and ensuring the system can manage anticipated loads without experiencing unnecessarily high latency. In the software industry, load testing is a standard method of assessing how a software project behaves under load. It is an essential component of assessing large-scale software systems because it can reveal various potential load-related problems when the system receives large-scale requests synchronously (Chen et al., 2017). The process of load testing provides thresholds to evaluate the system's performance under both high and low loads (Abbas et al., 2017). Load testing verifies the behaviors of the system under test (SUT), reveals bottlenecks, and detects functional and performance issues under various user workload scenarios (Mohamed et al., 2021).

In order to simulate real-world usage scenarios, load testing aims to apply different levels of traffic loads to the system. Usually, to accomplish this, specific software tools that produce virtual users, transactions, or requests are used. The system is then subjected to the behaviors of these virtual users in the same way as actual users. Load testing can be carried out manually by testers. However, with the help of test automation tools, testing processes can be more efficient and reliable. In addition, testing tools typically generate test reports with detailed statistics, graphs, and tables for further analysis (Ali et al., 2021). Popular load-testing frameworks include Azure Load Testing, JMeter, Artillery, Gatling, Locust, K6, NBomber, and WebValidate.

A collection of suitable test cases is typically used in load testing, which is executed to examine the responsiveness, throughput, reliability, and resource consumption of the target application under different user loads. Load testing can be carried out at various levels, including smoke test, average-load test, stress test, spike test, breakpoint test, and

soak test. Smoke tests are used to evaluate the system's performance under minimal load. Average-load tests aim to simulate the typical traffic of an average day. Stress tests are designed to assess the system's behavior under peak traffic. Spike tests simulate sudden spikes in traffic loads. The purpose of breakpoint tests is to test the upper limits and maximum capacity of an application. Soak tests are average-load tests that have longer test duration, which can last for hours or even days (Grafana, 2024).

3 Methods

3.1 Literature Review

3.1.1 Data Source and Search Terms

An extensive search was carried out in the abstract and citation database Scopus which provides research results from some of the most well-known scientific databases, including IEEE Xplore, ACM Digital Library, Springer Link, and Wiley Online Library. The strategy to form the search term was that the two words "monolith" and "microservice" both needed to be present in the Keywords. Phrases and expressions indicating research and comparison of resources and performance were added as a second item to search in the Title, Abstract, or Keywords of articles. The Subject area was limited to Computer Science and the Language was limited to English. The search terms were improved through a few iterations to find research literature concentrating on the comparison of resource consumption and application performance between monolithic and microservice architectures. The final search term:

KEY ((monolith OR monoliths OR monolithic) AND (microservice OR microservices OR micro-service OR micro-services)) AND

TITLE-ABS-KEY ((cost OR resource OR infrastructure OR compute OR performance OR efficiency OR responsiveness OR reliability) AND (analyze OR analysis OR study OR assessment OR review OR evaluation OR evaluate OR evaluating OR survey OR comparison OR compare OR comparative OR difference)))

AND (LIMIT-TO (SUBJAREA, "COMP")) AND (LIMIT-TO (LANGUAGE, "English"))

3.1.2 Research Procedures

The search with the final search terms in Scopus produced 106 papers across all the databases. Five papers were found through ad-hoc efforts. A list of 17 papers was selected after applying inclusion criteria and exclusion criteria: Inclusion criteria 1 (I1): must

contain comparisons of performance or related results between monolith and microservice architectures; Inclusion criteria 2 (I2): must contain comparisons of cost or related results between monolith and microservice architectures; Exclusion criteria 1 (E1): does not meet any inclusion criteria. The list of papers was used as seed papers for one-step forward snowballing, which produced 9 papers, out of which 2 papers were included after the same inclusion and exclusion criteria were applied. So in total 19 papers were selected for full-text reading. The research procedures and results are presented in Figure 3.1. The papers selected for full-text reading and final review are listed in Table 3.1. In this table, the articles are given unique identifiers from A1 to A19, which are used to identify the source articles in Table 4.2 of research results.

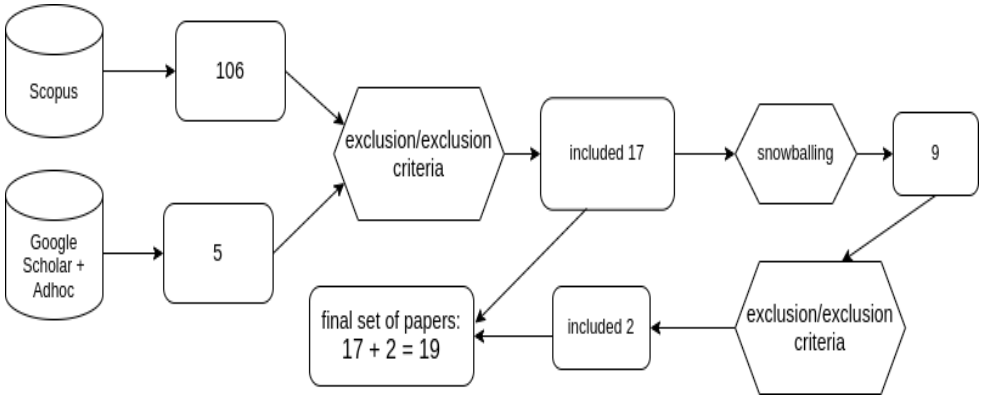


Figure 3.1: Literature Research Process

Table 3.1: Selected Articles

ID	Title	Author(s)	Year	Source
A1	Differences in performance, scalability, and cost of using microservice and monolithic architecture	Okrój Szymon; Jatkiewicz Przemysław	2023	ACM Digital Library
A2	Comparative Analysis of Monolith, Microservice API Gateway and Microservice Federated Gateway on Web-based application using GraphQL API	Kendricko Adrio; Clementius Nichklaus Tanzil; Michael Christian Lianto; Zulfany Erlisa Rasjid	2023	IEEE
A3	Evaluation of a Multitenant SaaS Using Monolithic and Microservice Architectures	Mangwani Poonam; Mangwani Niti; Motwani Sachin	2023	SpringerLink
A4	Reliability Evaluation of Microservices and Monolithic Architectures	Raharjo Agus Budi; Andyartha Putu Krisna; Wijaya William Handi; Purwananto Yudhi; Purwitasari Diana; Juniarta Nyoman	2022	IEEE
A5	Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation	Blinowski Grzegorz; Ojdowska Anna; Przybyłek Adam	2022	IEEE
A6	Comparative Analysis of Microservices and Monolithic Architecture	Benavente, Victor and Yantas, Luiggi and Moscol, Isabel and Rodriguez, Ciro and Inquilla, Ricardo and Pomachagua, Yuri	2022	IEEE
A7	Container Based Scalability and Performance Analysis of Multitenant SaaS Applications	Mangwani Poonam; Tokekar Vrinda	2022	IEEE
A8	A Comparative Study of Software Architectures in Constrained Device IoT Deployments	du Plessis Shani; Correia, Noélia	2021	IEEE
A9	Performance Comparison between Monolith and Microservice Using Docker and Kubernetes	Toomwong Napawit; Viyanon Waraporn	2021	IJCTE
A10	From monolithic systems to microservices: A comparative study of performance	Freddy Tapia; Miguel Ángel Mora; Walter Fuertes; Hernán Aules; Edwin Flores; Theofilos Toulkeridis	2020	MDPI
A11	Implementation of a web-based audience response system as microservice application vs. monolithic application	Iris Braun; Manuel Hoffmann; Robert Mörseburg	2019	IADIS digital library
A12	Microservice and Monolith Performance Comparison in Transaction Application	Alexander Jason Lauwren	2022	Unika SOEGI-JAPRANATA
A13	A Comparative Review of Microservices and Monolithic Architectures	Al-Debagy, Omar and Martinek, Peter	2018	IEEE
A14	Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures	Mario Villamizar; Oscar Garcés; Lina Ochoa; Harold Castro; Lorena Salamanca; Mauricio Verano; Rubby Casallas; Santiago Gil; Carlos Valencia; Angee Zambrano; Mery Lang	2017	SpringerLink
A15	The Comparison of Microservice and Monolithic Architecture	Gos Konrad; Zabierowski Wojciech	2020	IEEE
A16	Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud	Villamizar Mario; Garcés Oscar; Castro Harold; Verano Mauricio; Salamanca Lorena; Casallas Rubby; Gil Santiago	2015	IEEE
A17	Performance Evaluation of Monolithic and Microservice Architecture for an E-commerce Startup	Nayim Nahid Nawal; Karmakar Ayan; Ahmed Md Razu; Saifuddin Mohammed; Kabir Md. Humayun	2023	IEEE
A18	Performance characteristics between monolithic and microservice-based systems	Robin Flygare; Anthon Holmqvist	2017	DiVA
A19	Performance Analysis of Monolithic and Micro Service Architectures – Containers Technology: Proceedings of the 7th International Conference on Software Process Improvement (CIMPS 2018)	Alexis Saransig; Freddy Tapia Leon	2019	ResearchGate

3.2 Benchmarking

3.2.1 Applications Under Test

Two software systems with identical functionalities were selected from the GitHub public repositories for load testing to compare the performance and resource consumption differences between monolithic and microservice software architectures. The two systems were among a series of research projects developed by a Chinese software researcher. The two selected projects were developed with the same programming language and framework: Java and SpringBoot. The application is called Fenix's BookStore with the most basic functionalities of an online bookstore where customers can register an account, browse through the products, select products to add to the shopping cart, and finally checkout with a simulated payment process.

The project in a monolithic architecture is named monolithic_arch_springboot *. As shown in Figure 3.2, the monolithic application follows the principles of the domain-driven design (DDD) approach and is divided into four layers: infrastructure, domain, application, and user interface). In this design, the application layer contains application-specific business logic and orchestrates interactions between the presentation layer, domain layer, and infrastructure layer. The domain layer represents the core business logic and domain model of the application. The Infrastructure layer encompasses components responsible for interacting with external systems, databases, and infrastructure services. Despite not being a typical 3-layer monolithic design, all components of the application are included in the same code repository and the application is designed to be deployed as a single service. Every component in this application needs to be properly configured for the entire application to be functional.

The application in microservice architecture is named microservice_arch_kubernetes†. As shown in Figure 3.3, the whole application is divided into 5 subsystems: account, payment, security, warehouse, and gateway. The application is exposed via the gateway service which can be accessed through its public IP address. The subsystems are loosely coupled. Each one is separately configured and deployed as an independent microservice. The subsystems communicate with each other through HTTP protocol.

*https://github.com/fenixsoft/monolithic_arch_springboot
†https://github.com/fenixsoft/microservice_arch_kubernetes

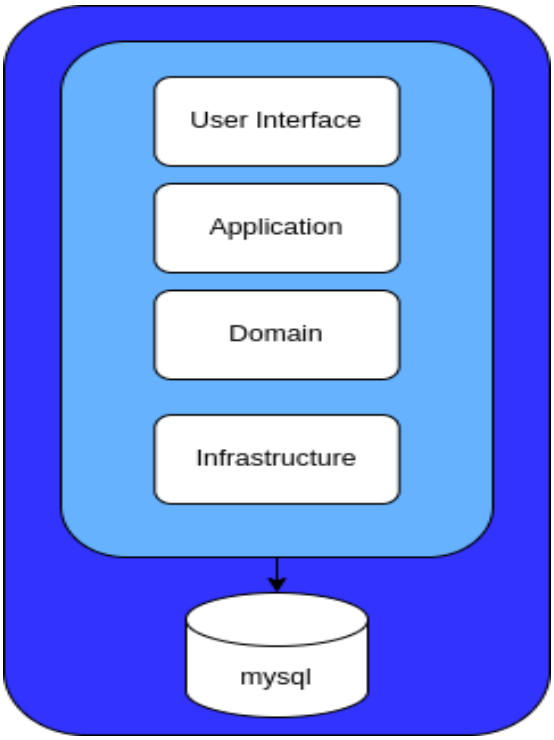


Figure 3.2: Fenix's BookStore in monolithic architecture

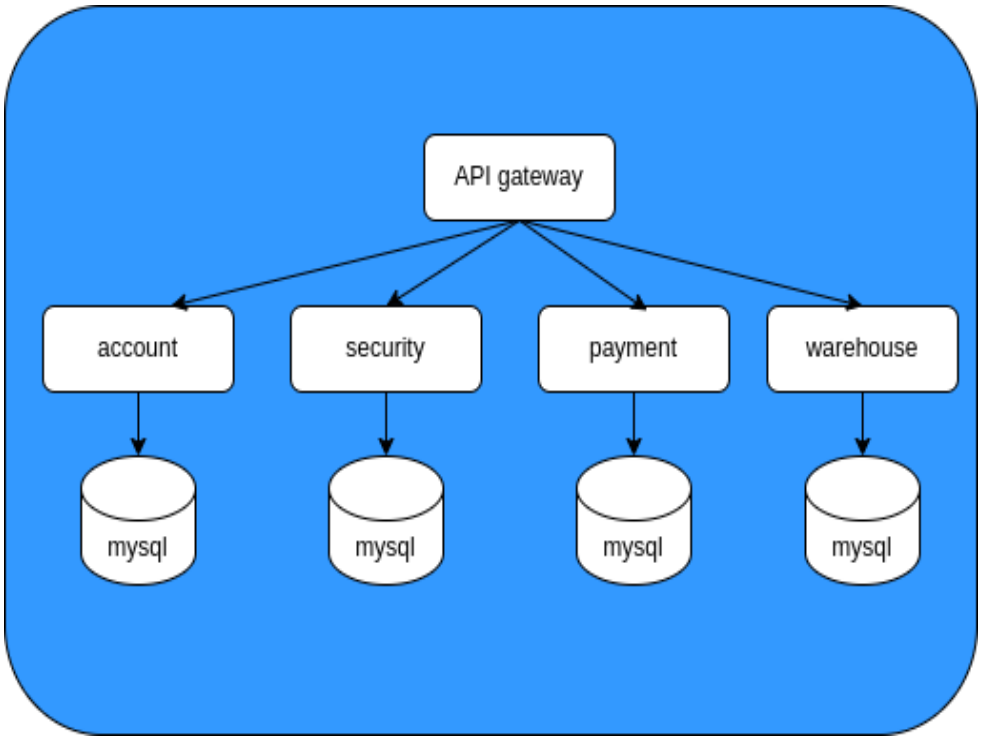


Figure 3.3: Fenix's BookStore in microservice architecture

3.2.2 Test Platform and Test Framework

For the benchmarking process, a series of load tests were performed against two applications using the load testing tool Locust*. Locust is an open-source tool for load testing. It can be deployed as an independent service that provides a web interface where testers can specify the total number of users for peak concurrency and the Ramp Up/Spawn Rate (users to be added per second). The test tasks can be created in Python programming language where different test scenarios and endpoints can be specified. Locust produces test reports with detailed statistics of tested endpoints, total requests, failures, and response time. The reports are presented in different formats, including text, graphs, and Comma-Separated Values(CSV). There are also logs with timestamps of the executed tests.

The applications under test and the test tool Locust were deployed in cloud environments provided by the Google Kubernetes Engine (GKE)†. Kubernetes is an open-source system to manage containerized applications. It helps automate the procedures of software deployment, scaling, and management. GKE is an implementation of the Kubernetes platform by Google. GKE provides easy steps to create and manage clusters, nodes, pods, and workloads through a web interface. Users can also deploy and manage applications in a cloud shell provided by the platform. In addition, GKE produces real-time statistics of resource consumption, including memory, CPU, and disk. It is also worth mentioning that for new users, the platform provides $300 worth of credits for a free trial period of 90 days.

3.2.3 Cloud Deployment

In Kubernetes, a node is a single machine. A "cluster" refers to a set of machines, which are used to run containerized applications managed by Kubernetes. A Kubernetes cluster normally consists of master nodes (or the control plane) and worker nodes (also known as "minions" or "slaves"). The master nodes are typically responsible for managing the Kubernetes cluster and coordinating the activities of the worker nodes. While the worker nodes are the machines to run containerized applications. In GKE, nodes are usually virtual machines, and customers can access only the worker nodes as the master nodes are managed by the cloud provider Google.

*https://docs.locust.io/en/stable/

†https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview

In the context of Kubernetes, a container is a standalone executable package that can run consistently across different computing environments. Container instances are created based on images using technologies such as Docker and contained. A pod is the smallest deployable unit in Kubernetes. In one pod, there can be one or more containers. A service is a resource in Kubernetes providing an access endpoint to one or more pods. Components and subsystems of an application communicate with each other via the IP addresses provided by their services. Typical types of services include ClusterIP, NodePort, and LoadBalancer. ClusterIP provides an internal IP address within the cluster for internal communication inside the cluster. Both NodePort and LoadBalancer can expose services for external access. NodePort exposes the service on a static port on each node's IP address. LoadBalancer uses a cloud provider's load balancer to distribute incoming traffic across the nodes.

For the cloud deployment of the load tests, a Kubernetes cluster was created in each test case with three worker nodes, which were of different machine types for different test scenarios. Each node was labeled for pod scheduling, and resource isolation was configured to ensure the application under test (AUT) was allocated with the amount of resources as planned. The test tool Locust and the database server MySQL were scheduled on different nodes than the application under test. In the test scenarios with no scaling and vertical scaling, Locust, MySQL, and the AUT were all deployed on different nodes, as shown in Figure 3.4, and Figure 3.5. For the test cases with horizontal scaling, Locust and MySQL were deployed on the same node so that the AUT could have two nodes for scaling capacity. The monolithic application was deployed as a single service on one worker node, so there was one pod with one container running inside. The five subsystems of the microservice-based application were deployed independently as five pods, with one container inside each pod. Since all pods, including Locust, MySQL, and the application AUT were deployed inside the same cluster, the communication between them was cluster internal, and the service type was set to be ClusterIP for both the AUT and MySQL. The Locust pod was exposed via a LoadBalancer so that it could be accessed on a local computer for test configuration, monitoring, and result data collection. The two applications were deployed and tested independently from each other and in each test case, the two applications were allocated with the same amount of resources.

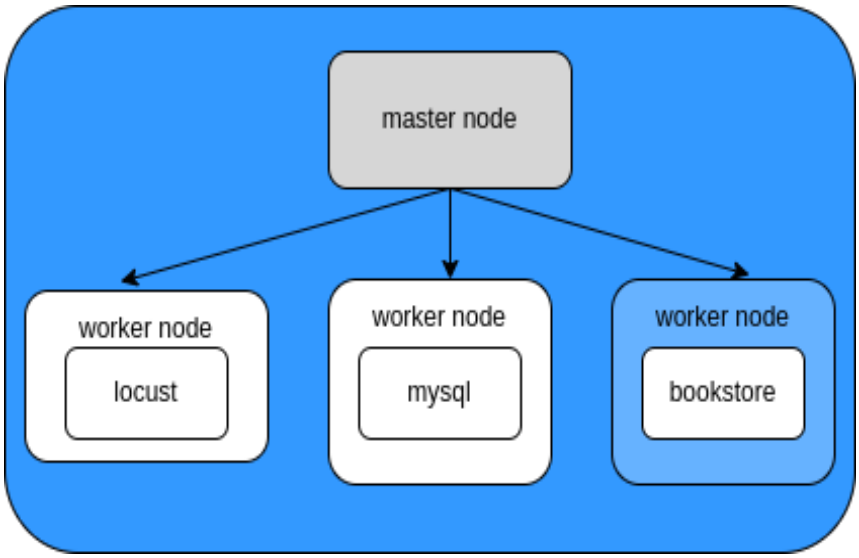


Figure 3.4: Cloud Deployment with Monolithic Bookstore

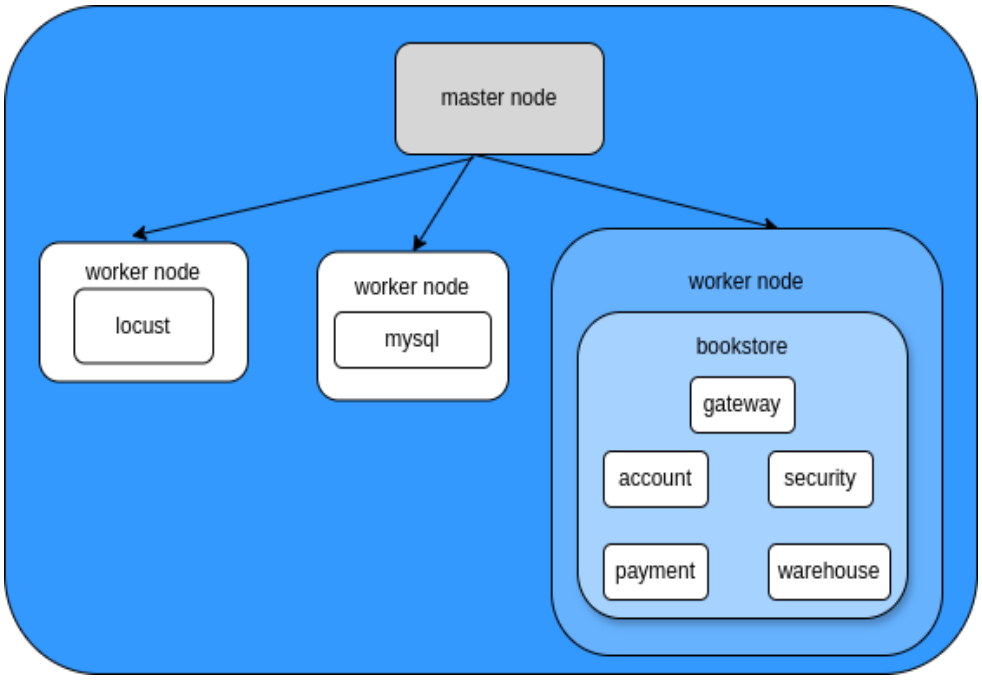


Figure 3.5: Cloud Deployment with Microservice Bookstore

3.2.4 Testing

Locust Tasks

In Locust, "tasks" refer to the actions or behaviors that virtual users (also known as "swarms") perform during a load test to simulate the actions of real users when interacting with an application under load. Tasks are defined using Python functions and are executed by each virtual user during the load test.

As listed in Table 3.2, two locust tasks (LT) were created with different API endpoints and tested using Locust against each application separately with different test environment setups. The first task LT1 was a simple HTTP GET request for a book specified by a randomly generated ID to simulate the action of a user clicking on a book image to view its detailed information. The request sequences for the monolithic and microservice applications are shown in Figure 3.6 and Figure 3.7. The second task LT2 was a more complex one that required communication among different components in the application to simulate the flow of requests a customer would send when buying a book from the online bookstore, including registering an account, selecting a book, adding it to the shopping cart, and then checking out. This task included multiple HTTP POST and GET requests. All the requests were chained and tested as one task with the Locust testing tool. The request sequences for the two applications are shown in Figure 3.8 and Figure 3.9.

Table 3.2: Locust Tasks

ID	HTTP Requests
LT1	GET /restful/products/id
LT2	POST /restful/accounts GET /oauth/token?username=username&password=password&grant_type=password&client_id=bookstore_frontend&client_secret=bookstore_secret GET /restful/products/id GET /restful/accounts/username POST /restful/settlements

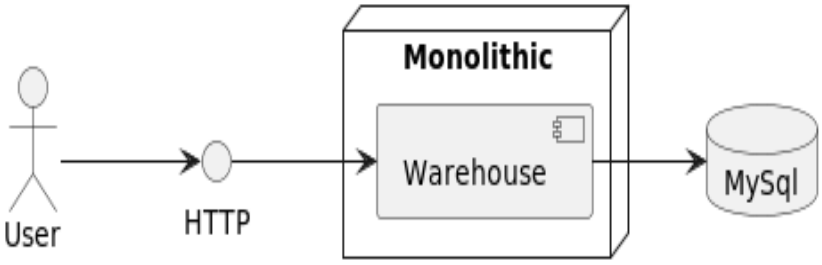


Figure 3.6: LT1 Request Sequence: Monolithic Bookstore

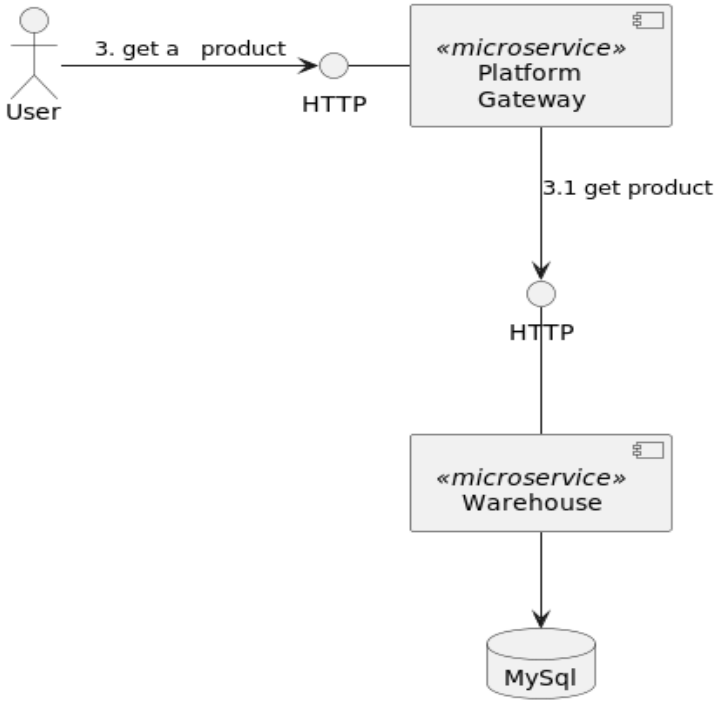


Figure 3.7: LT1 Request Sequence: Microservice Bookstore

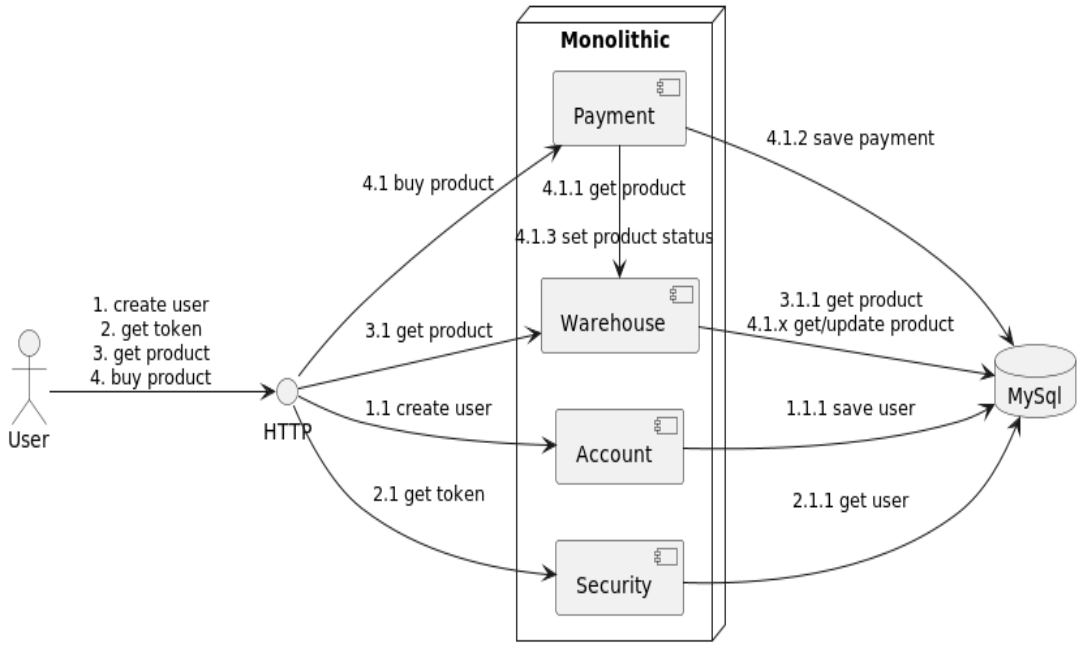


Figure 3.8: LT2 Request Sequence: Monolithic Bookstore

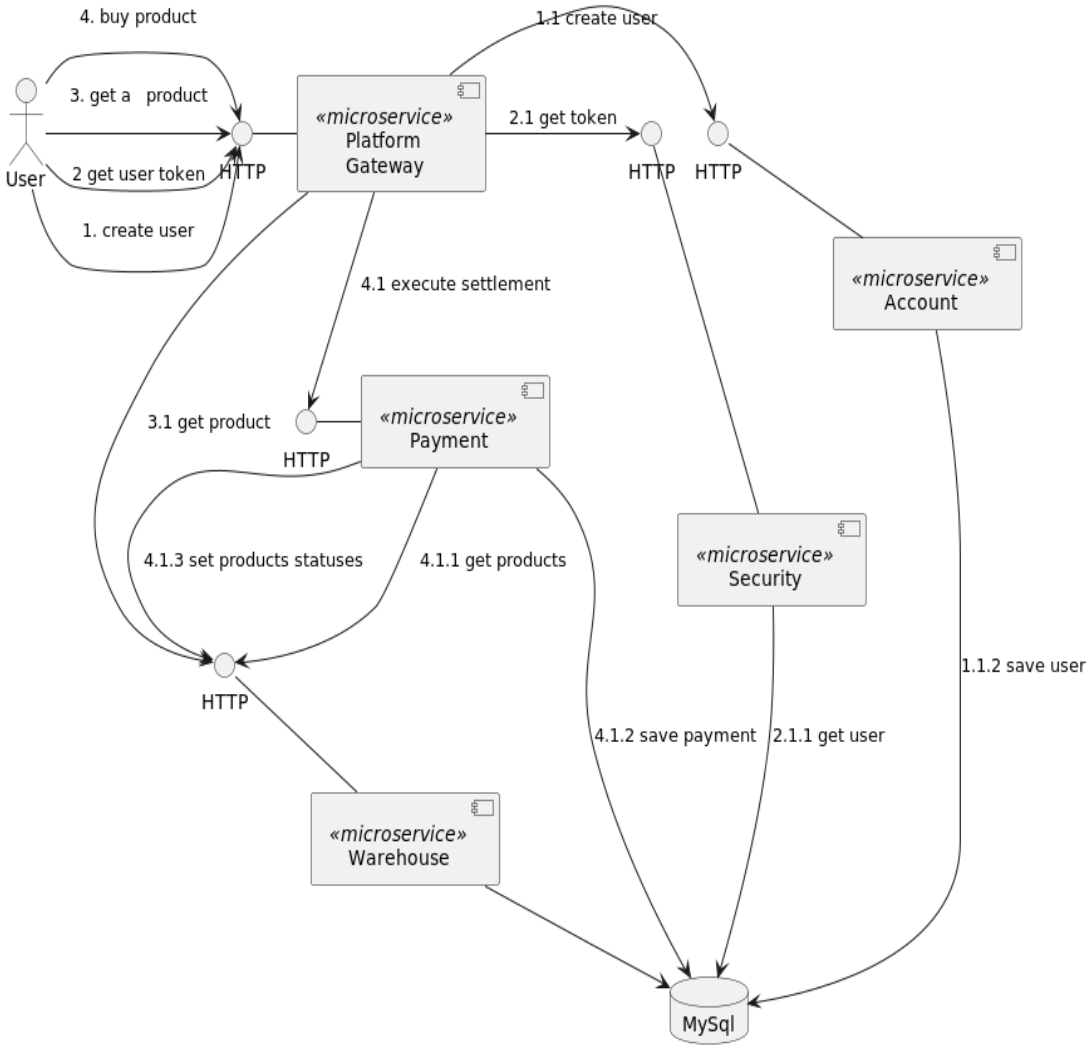


Figure 3.9: LT2 Request Sequence: Microservice Bookstore

Resources

As listed in Table 3.3, two machine types with different resource offerings were used in the tests. The first type was GKE e2-standard-4 with 4 vCPU and 16 GB memory. This type was used in test cases with a smaller number of users and no scaling policy. The second type was GKE e2-standard-8 with 8 vCPU and 32 GB memory. It was selected for test cases with a bigger number of users and auto-scaling policies.

Table 3.3: Resources

Machine Type	Total Resources	Estimated Monthly Cost
e2-standard-4	4 vCPU(2 core), 16 GB memory, 100 GB Disk	$396.51
e2-standard-8	8 vCPU(4 core), 32 GB memory, 100 GB Disk	$690.02

User Groups

Table 3.4 lists the different user groups planned for the load testing. "Peak Concurrency" represents the total number of users created for the test case for peak concurrency. "Spawn Rate" specifies the number of users to be started per second when the test case starts until it reaches the number of users defined in "Peak Concurrency".

Table 3.4: User Groups

ID	Peak Concurrency	Spawn Rate (users/s)
UG1	100	10
UG2	500	10
UG3	1,000	5

Scaling

Two auto-scaling policies were used in the load testing as defined in 3.5.

Horizontal Pod Autoscaler (HPA) automatically increases or decreases the number of pod replicas in a deployment based on defined metrics of resource consumption, such as CPU and memory. Horizontal Pod Autoscaler helps make sure that the application has the resources it needs to manage workloads and traffic fluctuations without going overboard as it monitors how much each pod in a deployment is using its resources to adjust the number of replicas to match the intended level of resource use.

Vertical Pod Autoscaler (VPA) adjusts the requests for CPU and memory resources of individual pods vertically. The Vertical Pod Autoscaler keeps track of how much memory and CPU each pod is using, and it dynamically modifies these requests in response to consumption trends. This optimizes resource allocation and utilization inside the cluster by enabling pods to request more resources when needed and release resources when not needed.

Table 3.5: Scaling Policy

Name	Description
Horizontal Pod Autoscaler(HPA)	HPA automatically adjusts the number of pod replicas in a deployment or replica set based on observed resource utilization and values defined in scaling metrics.
Vertical Pod Autoscaler(VPA)	VPA automatically adjusts the CPU and memory resource requests of individual pods vertically based on observed usage patterns.

Test Cases

All test cases are listed in Table 3.6. In the table, "User Group" refers to the user groups defined in Table 3.4; "Locust Task" refers to the locust tasks in Table 3.2; "Duration" is the total duration of the test in minutes. The column "Resources" refers to the machine type listed in Table 3.3 and "Nodes" is the number of nodes used for the application under test. The column "Scaling" shows the scaling policy configured for the test case. The scaling policies are described in Table 3.5.

Three scaling scenarios were planned, and two locust tasks with two groups of concurrency users were tested under each scenario. This formed a set of 12 test cases in total.

In the first scenario, no scaling policy was configured, and one node of machine type e2-standard-4 was allocated. One instance of the application under test was created and the two groups of concurrency users were set to 100 and 500, smaller than the numbers for the other two scenarios as the machine type in this scenario had a smaller amount of resources. The ramp-up was set to 10 and the duration to 5 minutes.

The second scenario had two nodes of machine type e2-standard-8 and the Horizontal Pod autoscaling (HPA) was configured to automatically increase or decrease the number of pod replicas based on predefined metrics, which was set to 80% usage of the requested CPU or memory. The memory request for each subsystem in the microservice-based application was set to 2.75 GB and the CPU request to 750 mCPU. The maximum number of replicas for each subsystem was set to 4. The memory request for the monolithic application was set to 5.5 GB and the CPU to 1500 mCPU. The maximum number of replicas was set to 10. With these settings, the two applications were ensured to have the same resources of 15 CPU and 55 GB memory. The two groups of concurrence users were set to 500 and 1000 and the test duration was set to 10 minutes. The ramp-up for the 1000-user group was set to 5 users per second to allow the application time for scaling.

In the third scaling scenario, the Vertical Pod Autoscaling (VPA) was enabled to allow automatic adjustments in resource allocation to the pod based on actual usage. The

machine type and the number of users were the same as in the second scenario except that only one node was allocated for the target application since the monolithic application could not expand to a second node under the VPA policy.

Table 3.6: Test Cases

ID	User Group	Locust Task	Duration(m)	Resources	Nodes	Scaling
TC1	UG1	LT1	5	e2-standard-4	1	None
TC2	UG2	LT1	5	e2-standard-4	1	None
TC3	UG1	LT2	5	e2-standard-4	1	None
TC4	UG2	LT2	5	e2-standard-4	1	None
TC5	UG2	LT1	10	e2-standard-8	2	HPA
TC6	UG3	LT1	10	e2-standard-8	2	HPA
TC7	UG2	LT2	10	e2-standard-8	2	HPA
TC8	UG3	LT2	10	e2-standard-8	2	HPA
TC9	UG2	LT1	10	e2-standard-8	1	VPA
TC10	UG3	LT1	10	e2-standard-8	1	VPA
TC11	UG2	LT2	10	e2-standard-8	1	VPA
TC12	UG3	LT2	10	e2-standard-8	1	VPA

4 Results

4.1 Literature Review Results

As shown in Table 4.1, among the selected 19 articles, 18 contributed empirically with case studies and tests to demonstrate the process of their research results. In terms of research focus, 10 articles focused on application performance, 2 focused on resource consumption and cost-effectiveness, and the rest articles included research results from both perspectives. However, not all papers were from professional sources. Among the 18 papers in the systematic literature review, only 10 were published in IEEE, and 5 were from some not well-known publishers. All the papers that included empirical experiences were included in this literature review because the total number of research papers on this topic was minimal. The quality of the research papers varies. Many papers were just a few pages long, lacking detailed documentation on the test procedures. In four of the articles, the test environment and resource allocation were not mentioned. The test cases or scenarios were not clear in two of the papers.

The test results and research findings of the reviewed papers are categorized by application performance and resource consumption and summarized in the following two subsections. The detailed research results of all the reviewed articles are listed in Table 4.2, which includes the test scenarios, test environments, and resource allocation, as well as overall test results from all the reviewed articles. The article IDs in this table are from Table 3.1 Selected Articles.

Table 4.1: High-level Findings

Contribution type	No. of papers
empirical	18
theoretical	1
Research focus	**No. of papers**
application performance	10
resource consumption	2
performance & resource	7

4.1.1 Application Performance

According to the overall test results in the reviewed articles, the monolithic architecture was found to have better performance in response time and throughput when handling smaller numbers of user requests, and the microservice demonstrated advantages in processing heavier traffic, reliability, fault tolerance, and scalability. In more detail, in article A1 (Jatkiewicz and Okrój, 2023), the monolithic architecture performed better in both local and cloud environments with the same resource allocation. However, the horizontal scaling of the microservice architecture was found more efficient than the monolithic architecture. The monolithic architecture in article A2 (Adrio et al., 2023) was reported to have higher throughput and smaller response time, while the microservices were proven to be more reliable and scalable. The microservice-based system in article A4 (Raharjo et al., 2022) was found to be slower than the monolithic system, but it returned fewer errors and had better fault tolerance. The test results in articles A3 (Poonam et al., 2023), A11 (Iris et al., 2019), A12 (Lauwren, 2022), A15 (Gos and Zabierowski, 2020), and A18 (Flygare, 2017) showed that the monolithic application had better performance results when handling lower traffic loads, while the microservice-based application outperformed when processing heavy traffic loads. The article A5 (Blinowski et al., 2022) reported that the monolithic application performed better in both local and cloud environments without scaling, while the microservice system achieved better performance when scaling was enabled. A6 (Benavente et al., 2022) found that the microservice architecture was a better option for handling a large number of requests and it showed better fault tolerance. The tests also found that vertical scaling of the monolithic application could no longer improve performance after it reached a certain upper limit. In article A7 (Mangwani and Tokekar, 2022) the test results showed that the application in a microservice architecture had better performance under heavy load in terms of response time, throughput, and scalability. The monolith architecture in A9 (Napawit and Waraporn, 2021) performed better than the microservice architecture in terms of both average response time and throughput under a test of 300 transactions per second load for 10 minutes. A13 (Al-Debagy and Martinek, 2018) concluded that the monolithic application performed better in terms of throughput and response time under low loads, and microservices and monolithic applications can have similar performance under normal loads. In A16 (Villamizar et al., 2015), the monolith application has slightly better performance in terms of average response time, but the cost was also higher compared to the microservice approach.

4.1.2 Resource Consumption

In terms of resource consumption and cost-effectiveness, based on the overall research findings, applications in microservice architectures were more cost-efficient despite some research papers reporting that microservice systems needed more infrastructure resources for cloud deployment. To be more specific, in article A2 (Adrio et al., 2023), it was found that scaling of independent microservices resulted in more effective resource consumption. It was reported that resource utilization was more effective for microservices and the need to scale the entire monolithic application resulted in underutilization of resources in article A3 (Poonam et al., 2023). In article A4 (Raharjo et al., 2022) it was found that during peak traffic, the monolithic system used more memory than the microservice application. The article A5 (Blinowski et al., 2022) found that the most cost-efficient was the microservice system with both horizontal and vertical scaling enabled. The microservice-based system in A7 (Mangwani and Tokekar, 2022) achieved better utilization of resources, such as CPU, under heavy traffic loads, while no big difference was found between the two architectures during low traffic. In A8 (Plessis and Correia, 2021), it was concluded that for small-scale applications, the monolithic architecture returned better results across most metrics, including maximum instantaneous power consumption, total power consumption, overall runtime, maximum RSS, and CPU. In A10 (Tapia et al., 2020), the microservice architecture was found more efficient in hardware resource consumption, cost reduction, and productivity. The microservice architecture in A14 (Mario et al., 2017) provided better performance and the infrastructure cost was lower than the monolithic architecture. in A17 (Nayim et al., 2023), the microservice system performed better during high-load testing but it needed more infrastructure resources and the cost was much higher than the monolithic application. A18 (Flygare, 2017) reported that the monolithic architecture was more resource-friendly with higher hardware efficiency compared to the microservice architecture. In A19 (Saransig and Tapia Leon, 2019), the microservice application used more infrastructure resources, including CPU, RAM, and networking than the monolithic application and delivered better performance results.

Table 4.2: Literature Review Results

Article	Test Environments	Test Scenarios	Overall Results
A1	local computer: i5-3350P 3.1GHz processor, 8GB RAM Azure cloud: B1 - single-core processor, 1.75 GB RAM; B2 - dual-core processor, 3.5 GB RAM	-1000 queries for simple request -100 queries for complex request -100 queries for simple request 10 queries for complex request -100 queries for simple request and 50 queries for complex request	- local computer: for time-consuming requests, the performance differences are negligible, otherwise the monolithic architecture performs much better - cloud: based on vertical scaling to a maximum of a B2 machine and horizontal scaling to a maximum of three B1 machines, the monolithic architecture performs better with the same cost on vertical and horizontal scaling. - scaling: vertical scaling incrases the monolithic architecture performance approximately 2 times, for the microservice architecture, it can be 2 - 5.5 times.
A2	4 CPU (core), 512 RAM (MB)	Load Testing specifications: - Number of threads (users): 500-2000 threads - Ramp-up period: 60 seconds - Number of iterations: 5 iterations for each thread. Stress testing specifications: - Number of threads (users): 100-2000 threads - Ramp-up period: 1 second - Number of Iteration: 1 iteration for each thread	the monolith architecture-based application achieves the best result both in terms of speed and throughput to handle a high number of requests from various users at the same time. The microservice architecture achieves better results in terms of scalability.
A3	GKE (resource allocation/limit not defined)	100 user requests for 2 min, then increased to 500 users	independent scaling of micro-services leads to effective utilization of resources compared to the monolithic approach. the monolithic approach performs better at low loads and microservices work better at higher loads.
A4	Heroku cloud: 512 MB of RAM with 1x CP	endpoints used are login, retrieve posts and papers, and create posts	both systems show the same degree availability and maturity. the monolithic system was on average faster than the microservices system when handling the same requests. microservices showed better reliability, recoverability and fault tolerance.

Table 4.2 – continued from previous page

A5	local: Intel(R) Core(TM) i7-9850H CPU 2.60GHz, six physical, 12 logical cores, and 32 GB RAM (Dell Precision 7540) Azure Spring Cloud: 16 cores and 32 GB RAM Azure App Service: unclear	endpoints: - a simple single-object query invoked 1000 times - a computationally intensive query invoked 100 times - the number of threads was set to 10 for both scenarios - each of the test runs was repeated 20 times	- on a single machine, the monolith performs better than the microservice approach; - in cloud environments, the monolith outperforms the microservices - a monolithic architecture seems to be a better choice for simple, small-sized systems that do not have to support a large number of concurrent users.
A7	GKE: 4 nodes, 7 vCPU, 14GB memory	started with 50 users, then increased to 500; test duration 2 min	microservices outperforms monolithic applications by minimizing the response time under high load whereas monolithic is better choice at low load. Moreover, higher throughput and better utilization of CPU is achieved with a greater number of user requests handled as compared to monolithic.
A8	Raspberry Pi 4 with 4GB RAM, 16GB MicroSD MacBook Pro with a 2,3 GHz Dual-Core Intel Core i5 processor, 8GB RAM	unclear	the monolithic architecture had better performance in most metrics, including power consumption, overall runtime, maximum RSS and CPU. when deploying small-scale applications on IoT devices, the monolithic architecture may offer more benefits
A9	Amazon EC2 T3.small: 2vCPUs, 2.5 GHz, Intel Skylake P-8175, 2 GiB memory horizontal autoscaling if CPU utilization more than 50 percent	300 transactions per second for 10 minutes	the monolith architecture performed better than the microservice architecture in terms of both average response time and throughput
A10	unclear	- GET requests to generate the databases and data: 273 requests, 10,000 generated data, 30 repetitions, three threads, - GET requests to select the information from DB: 1053 requests, 20,000 data generated, 70 repetitions, five threads	the architecture of microservices with containers proved to be more efficient; the study demonstrates more efficiency concerning hardware resources, cost reduction, and high productivity when using microservices.

Table 4.2 – continued from previous page

A11	unclear	service scenarios: Login with registration; Login without registration; Enroll into one course; Get lectures user load: 1) 100 users try to execute the given test in parallel; 2) 100 users get ramped up from 0 within 10 s and execute the given scenario; 3) 10 users per second over 60 s start executing a test	for small user loads of up to 10 users per second, the monolith was the better choice because it had a higher success rate and shorter response times; the microservice architecture performs better for higher user loads
A12	unclear	The testing scenarios conducted are basic create, update, delete on merchant service, login service from the user service, and bulk transfer from transaction service with user loads of 100, 1000, and 5000 separately	the overall testing data shows the average latency from the monolith is better than the microservice, while microservices average success rate is higher. Overall, if the service is more loaded, and needs high success rates, microservice is a better choice. If the service is lightweight and needs faster response time, the monolith is a better option.
A13	unclear	- load Testing: starts with 100 threads with a ramp-up of 2 minutes and holds time of another 2 minutes, then increases the number of threads until 7000 threads each time with 2 minutes for rampup and holds time. - concurrency Testing: started with 100 requests for each service with no specific ramp-up time and increasing the number of requests gradually until 1000 requests	load testing results: microservices and monolithic application can have similar performance under normal load on the application. For a small load with less than 100 users, the monolithic application can perform a little bit better concurrency testing results: The monolithic application showed higher throughput on average.
A14	AWS: monolith: 4 instances of c4.large type(2 vCPUs, 8 ECUs, and 3,75GB RAM) microservice S1: 3 instances of c4.large type (2 vCPUs, 8 ECUs, and 3,75GB RAM) microservice S2: one EC2 instance of t2.small type (1 vCPUs, Variable ECUs, and 2,0GB RAM)	unclear	based on the calculation of cost per million requests, the results of the performance tests show that microservices can provide a better performance than the monolithic architecture at a lower cost and supported more requests per minute; however the average response time of the microservice approach is higher than the monolith counterpart because each request must pass between the gateway and each microservice.

Table 4.2 – continued from previous page

A15	local PC: Processor - Intel i9-9900K 3.6 GHz – 8 cores, 16 threads RAM - G.SKILL 32 GB DDR4 3200 MHz CL14 HDD - Samsung SSD 840 EVO 120 GB	- 1 000 requests made at once by 30 users (1 000 * 30 = 30 000 requests) - 10 000 requests made at once by 30 users (10 000 * 30 = 300 000 requests)	- for the tests with 30,000 requests, in terms of requests per second and response time, the most powerful is the monolithic application. - for the test with 300,000 requests, the microservice performed better in terms of both requests per second and response time. - microservice architecture is more efficient if an application has to handle a bigger number of requests. the monolithic architecture is more efficient during lower loads
A16	AWS: monolith: c4.2xlarge (8 vCPUs, 31 ECUs and 15GB RAM) microservice S1: 4 vCPUs,16 ECUs and 7,5GB RAM microservice S2: 1,vCPUs,3 ECUs and 3,75GB RAM	30 requests per minute to the service S1 and 1,100 requests per minute to the service S2 during 10 minutes	the monolith application has slightly better performance in terms of average response time, but the cost is also higher for the monolith compared to the microservice approach; the microservice architecture may reduce infrastructure costs by 17% compared to the monolith
A17	AWS: monolith: (EC2) instance c5d.2xlarge (8vcpus, 16gb ram, 1*200 nvme ssd), db.t3. medium microservices: (EC2) instance c5d.xlarge (4vcpus, 8gb ram, 1*200 nvme ssd), db.t3. medium, cache.r7g.large(vcpu 2, memory 13.07gb)	load testing(1000 users), stress testing(50000 users)	the cost of the microservice deployment is much higher than the monolithic architecture deployment. monolithic architecture can reduce cost by up to 60.64% compared to microservice architecture. Load testing: microservice has a higher http-req than monolithic architecture. Stress testing: monolithic architecture is better than microservice architecture.

Table 4.2 – continued from previous page

A18	2x Intel NUC5i7RYH with the following specifications: 16GB RAM, Intel Core i7-5557U Processor	Test cases: TC1 Create a Person and fetch a Person (2 properties); TC2 Create an Address and fetch an Address (5 properties); TC3 Create a Profile and fetch a Profile (10 properties); TC4 All of the above test cases concurrently User loads: (a) 1 simultaneous user; (b) 10 simultaneous users; (c) 100 simultaneous users; (d) 1000 simultaneous users	the monolithic architecture is more resource-friendly with higher hardware efficiency; the monolithic architecture has lower latency in all test cases except when the number of users is increased to 1000; the monolithic architecture has a higher throughput rate in all test cases compared to the microservices architecture.
A19	Intel Core i7 2.7 GHz CPU, 16 GB memory, HD disk and Virtual Network	Two cases of study are generated. The first with a reduced number of requests (273) and the second case with a larger number of requests (1053).	the performance of the microservice application shows an advantage over the monolithic application in terms of request handling efficiency and the total number of requests, but its resource consumption is also higher, on average it consumes 12.38% more CPU, and 8.87% more memory. The monolithic architecture consumes fewer resources, including CPU, RAM, and network but the response time is longer. The two architectures are reasonably equal when time is a relevant factor.

4.2 Benchmarking Results

4.2.1 Load Test Results

A set of twelve load test cases was performed on both the monolithic and the microservice applications. Each test case was performed on one application at a time. After each test case, the resource consumption data, namely CPU and memory usage, was collected from GKE, and the application performance data, such as total requests, failures, and response time, was collected from the test tool Locust. The test results are presented in two tables, Table 4.3 and Table 4.4, and a PDF document titled "Resource Consumption and Application Performance" in Appendix A "Load Test Report".

The values in Table 4.3 are organized by test case numbers, which are defined in Table 3.6. Each test case has two rows, the upper row for the microservice application and the lower row for the monolithic application so that the values can be easily compared between the two applications. The column "Avg. RPS" represents the average requests per second. "Avg. RT(ms)" is the average response time in milliseconds. The column "Failure Rate" is the value of (total failures)/(total requests). If the failure rate is < 0.001, it is rounded to 0 and marked as ~0. The columns "Avg. RPS", "Avg. RT" and "Failure Rate" are based on the statistics from Locust test reports. The column "Avg. CPU" is the average CPU usage per 10,000 requests in mCPU, and the column "Avg. Memory" is the average memory usage for every 10,000 requests in MiB. The average CPU and memory usage values were calculated using the Comma-Separated Values(CSV) downloaded from GKE metrics.

Table 4.3: Load Test Results: Microservice vs Monolith

Test Case	Application	Avg. RPS	Avg. RT (ms)	Requests In Total	Failure Rate	Avg. CPU (mCPU)	Avg. Memory (MiB)
TC1	Microservice	760.5	99	229514	0	31.806	150.987
TC1	Monolith	185	531	56842	0	265.649	90.060
TC2	Microservice	701.1	453	220575	0	32.642	165.237
TC2	Monolith	179.9	2545	55733	~0	249.403	105.934
TC3	Microservice	300.3	327	96383	0.056	284.282	266.909
TC3	Monolith	39.4	2483	11972	0.078	1445.038	394.587
TC4	Microservice	221.2	1339	68165	0.006	394.631	337.059
TC4	Monolith	32.8	7255	9976	0.123	2425.822	540.186
TC5	Microservice	713.9	463	433735	0	29.281	118.851
TC5	Monolith	415.8	1130	249527	~0	254.081	88.261
TC6	Microservice	676.5	839	411192	0	27.238	119.131
TC6	Monolith	468.1	1588	281191	0.003	251.075	116.815
TC7	Microservice	389.1	1215	235144	0.059	278.127	293.609
TC7	Monolith	110.5	4296	66304	0.123	1491.614	668.881
TC8	Microservice	434.4	1845	263390	0.159	256.273	247.801
TC8	Monolith	108.0	7566	64827	0.229	1488.577	686.366
TC9	Microservice	942.9	350	573817	0	15.858	38.421
TC9	Monolith	289.5	1653	175565	0	193.660	39.011
TC10	Microservice	897.1	634	539409	0	14.646	41.191
TC10	Monolith	292.9	2849	180799	0	174.226	48.936
TC11	Microservice	527.9	878	239286	0.199	152.119	142.176
TC11	Monolith	66.6	7155	41276	0.309	1238.007	242.431
TC12	Microservice	611.0	1295	366689	0.287	119,992	178.927
TC12	Monolith	59.6	12736	36173	0.109	1119.619	237.373

The data in Table 4.4 are the results of divisions using the data from Table 4.3. The values were calculated by dividing the data from the microservice application by the corresponding data from the monolithic application to show the differences in the ratio in the compared columns between the two applications.

Table 4.4: Load Test Results: Microservice ÷ Monolith

Test Case	Avg. RPS	Avg. RT	Total Requests	Avg. CPU	Avg. Memory
TC1	4.111	0.186	4.038	0.120	1.677
TC2	3.897	0.178	3.958	0.131	1.560
TC3	7.622	0.132	8.051	0.197	0.676
TC4	9.793	0.185	6.832	0.163	0.625
TC5	1.717	0.410	1.738	0.115	1.347
TC6	1.392	0.528	1.462	0.109	1.020
TC7	3.521	0.283	3.546	0.186	0.439
TC8	4.022	0.244	4.063	0.172	0.361
TC9	3.257	0.212	3.268	0.082	0.985
TC10	3.063	0.223	2.983	0.084	0.842
TC11	7.926	0.123	5.797	0.123	0.586
TC12	10.252	0.102	10.137	0.107	0.754

The PDF document titled "Resource Consumption and Application Performance" in Appendix A is a collection of charts of CPU and memory usage, and load test request and response statistics. The CPU and memory usage charts were created using the CSV downloaded from GKE metrics, for which filters were applied to ensure resource consumption by the AUT only. The CSV data in GKE were the average resource utilization per minute throughout each test duration. The request and response statistics charts were downloaded from Locust. The charts are organized by test case numbers.

Based on the data in the above-mentioned PDF document and Table 4.3 and Table 4.4, the test results are summarized as follows:

TC1

In TC1, both applications successfully handled all the requests with no sudden spikes or drops during the whole test run. The microservice application handled 229,514 requests with a remarkably low average response time of 99ms per request. It consumed 31.806 mCPU and 150.987 MiB memory on average for every 10,000 requests. The monolithic application processed 55,733 requests in total and the average response time was 185 ms. The average CPU and memory consumption was 265.649 mCPU and 90.06 MiB respectively.

TC2

In TC2, no failure was reported for the microservice system. An unnoticeable tiny percentage of requests failed for the monolithic application. The charts of the Total Requests and Response Time indicated little deviation for both applications. The total number of requests reached 220,575 and the average response time stood at 701 ms for the mi-

croservice application. The monolithic application responded to 55,733 user requests at an average response time of 2,545 ms. The microservice system used 32.642 mCPU and 165.237 MiB memory while the monolithic application used 249.403 mCPU and 105.934 MiB memory for every 10,000 requests.

TC3

In TC3, the Total Requests charts were relatively stable for both applications. The microservice application handled 96,383 requests in total. The number of requests by the monolithic application was 11,972. The Response Times chart of the microservice application was stable and the values were quite low with an average value of 327 ms. The response time chart of the monolithic application showed many ups and downs in the 50th percentile line while the 90th percentile line remained stable. For both applications, nearly half of the requests failed during the last minute of the test run. Based on error messages in the application logs, the failures might have been caused by concurrency issues due to high traffic. For every 10,000 requests, the average CPU and memory usage was 284.282 mCPU and 266.909 MiB for the microservice system, and 1445.038 mCPU and 394.587 for the monolithic application.

TC4

In TC4, for the monolithic application, some requests started failing at the beginning of the test case when more users were created to reach the peak concurrency, and the application crashed two times at around the time when the number of users was almost at the peak level. After that, the total number of users dropped to about 200. Due to the crashes, request failures were recorded and the Response Times chart showed irregular and unpredictable patterns during this period. The microservice application faced similar issues but showed better results than the monolithic application as it was able to maintain the number of users at around 300 and the number of failed requests was only around 4% of the monolith's failures. In total 68,165 requests were processed by the microservice application and 9,976 by the monolithic application. The average CPU and memory used for every 10,000 requests by the microservice system was 394.631 mCPU and 337.059 MiB, and 2425.822 mCPU and 540.186 MiB by the monolith.

TC5

In TC5, the microservice application did not fail any request from all 500 users during the whole test run. The charts of Total Requests and Response Times showed persistent stability. In total, it handled 433,735 requests by using 29.281 mCPU and 118.851 MiB memory for every 10,000 requests. The charts of the monolithic software indicated that

at the beginning of the test, the total number of requests was as low as 80, and the 95th percentile response time reached 9600 ms. After 5 minutes into the test, the application was able to scale out and the total requests per second increased to around 700. Overall, it managed to process 249,527 requests with 103 failures with average CPU and memory consumption at 254.081 mCPU and 88.261 MiB for every 10,000 requests.

TC6

In TC6, both applications showed similar patterns to those in TC5 in the Total Requests and Response Times chats. The total number of requests responded to by the two applications was 411,192 and 281,191 respectively. The microservice system consumed 27.238 mCPU and 119.131 MiB memory on average for every 10,000 requests. The monolithic application used 251.075 mCPU, while its memory consumption was 116.815 MiB, slightly lower than the microservice application's consumption.

TC7

In TC7, the monolithic application was slow at scaling in the first half of the test run, causing the response time to reach over 11,000 ms and the total requests per second to be at a low level of 70. After it managed to scale out, the total requests increased to 150 and the response time dropped to 8,000 ms. The piled-up requests caused concurrency issues, which led to half of the requests failing at the end of the test case. For the microservice system, the total number of requests per second was around 400 and the response time was over 5,000 at the beginning of the test. Half of the requests failed for one minute in the middle of the test due to concurrency issues. The application was able to recover itself through restarts and scaling efforts and increased the total requests to around 500 halfway through the test. Overall, the total number of requests and the average response time were 235,114 and 1,215 ms for the microservice system and 66,304 and 4,296 ms for the monolithic software. The average CPU and memory usage for every 10,000 requests were 278.127 mCPU and 293.609 MiB for the microservices, and 1491.614 mCPU and 668.881 MiB for the monolith.

TC8

In TC8, the charts of the microservice application showed that half of the requests failed in the middle of the test for about 3 minutes due to concurrency issues. The issues were resolved one minute before the end of the test. On average, the response time was 1,845 ms and the number of requests per second was 434.4. In total, 263,390 requests were processed at a failure rate of ~16%. The average CPU and memory usage for every 10,000 requests by the microservice was 256.273 mCPU and 247.801 MiB. For the monolithic application,

the Total Requests and Response Times charts showed higher fluctuations during the whole test run. Half of the requests started failing 6 minutes into the test and the failures continued until the end of the test, causing the failure rate to reach ~23%. Overall, the monolithic application handled 64,827 requests with an average response time of 7,566 ms. On average, it used 1488.577 mCPU and 686.366 MiB memory for every 10,000 requests.

TC9

In TC9, for both applications, all requests were successfully handled. The Total Requests and Response Times charts of the microservice system showed persistent stability with the total requests per second maintained at around 1,000 and the response time under 500 ms. The monolithic application had relatively stable charts as well except for two small downward curves in the Total Requests chat for unknown reasons. The total number of requests was 573,817 by the microservices and 175,565 by the monolith. The average CPU and memory consumption for every 10,000 requests were 15.858 mCPU and 38.421 MiB for the microservice system, and 193.66 mCPU and 39.011 MiB for the monolithic application.

TC10

In TC10, the application performance charts for both applications indicated similar consistency as in TC9. The microservice application used 14.646 mCPU and 41.191 MiB memory for every 10,000 requests. The monolithic application processed 180,799 user requests in total with an average consumption of 174.226 mCPU and 48.936 MiB memory.

TC11

In TC11, both applications experienced fairly high failure rates, nearly 20% for the microservice system and 30% for the monolithic application. One likely reason to the high failure rates could be the applications were not able to scale up fast enough to handle the increasing number of requests and the accumulated requests caused concurrency issues to the applications. Overall, the microservice application recorded a total number of 239,286 requests, and the average CPU and memory usage for every 10,000 requests was 152.119 mCPU and 142.176 MiB. The monolithic application processed 41,276 requests with an average CPU and memory usage of 1238 mCPU and 242.431 MiB for every 10,000 requests.

TC12

In TC12, the microservice application reported similar patterns in the charts as in TC11. The failures started to appear only 4 minutes into the test run, and continued until the end of the test case, causing the failure rate to increase to 29%. The application processed

in total 366,689 user requests and the CPU and memory used by the microservice system were 119.992 mCPU and 178.927 MiB respectively. The Total Requests and Response Times charts of the monolithic application showed irregular and unpredictable patterns in the data. The application restarted 4 times during the test run, causing dramatic peaks and valleys in the charts. The downtime after each restart lasted for around 1 minute. The failures caused by concurrency issues did not appear in this test run, possibly due to the frequent restarts, so the failure rate was at only 10%, lower than the microservice system's. The average response time reached the highest record in all tests at 12,736 ms. The number of users for peak concurrency dropped to 879 in the middle of the test. In the end, with an average CPU and memory usage of 1119.619 mCPU and 237.373 MiB, the monolithic application processed a total number of 36,173 requests.

4.2.2 Application Performance and Resource Consumption

The tests were divided into three categories based on their scaling policy and under each category two Lucust Tasks (LT1 and LT2) were tested with two user groups (UG1 and UG2 or UG2 and UG3). The first two test cases under each scaling category tested LT1 of a simple HTTP GET request with two different user groups. The other two test cases tested LT2 of multiple HTTP GET and POST requests. The following summary of application performance and resource consumption is based on this structure:

No Scaling

In test cases TC1 - TC4, no scaling policy was configured, and no resource limit was set for the applications, so the applications could use as much of the available resources as needed. The applications were allocated with enough resources based on prior experimental tests. The test results showed that the available resources were much higher than what was used in the tests.

- LT1: Simple Task

In TC1 and TC2, both applications successfully handled nearly all the requests. The microservice delivered better performance results in terms of average requests per second, average response time, and total number of requests. The total number of requests handled by the microservice application was 4 times of what the monolithic application processed.

The average CPU usage for every 10,000 requests by the microservers was only 12% of what was used by the monolithic system. However, the average memory usage of the microservice application was higher, over 150% of the monolithic system's consumption.

- LT2: Complex Task

In TC3 and TC4, both applications reported failures in handling the requests. In TC4, the monolithic application restarted two times which caused spikes in the performance charts; the microservice system restarted once and was slower at the beginning to respond to the requests; the total number of users for peak concurrency dropped from 500 to 200 for the monolith and 300 for the microservices. Overall, the microservice application outperformed its monolith counterpart in average requests per second, average response time, total number of requests, and success rate. In TC3, the microservice system handled 7 times more requests, and in TC4, it handled 6 times more requests. However, the resource consumption of both CPU and memory by the microservice system was lower. The microservice system's CPU and memory usage was ~20% and ~67% respectively of the monolith's consumption in TC3. In TC4, the microservice system's CPU and memory usage was ~16% and ~63% respectively of the monolith's consumption.

Horizontal Scaling

In test case TC5 - TC8, the Horizontal Pod Autoscaler was enabled, and the resource requests and the maximum number of pod replicas were carefully calculated to ensure the same amount of resources in total for the two applications.

- LT1: Simple Task

In TC5 and TC6, the microservice application returned a 100% success rate while the monolithic system reported insignificant failure rates, < 0.001 in TC5 and ~0.003 in TC6. Based on the requests and response times chats, the microservice system was able to scale out fast to respond to increasing traffic without showing noticeable fluctuations in the total number of requests per second and average response time throughout the whole test duration. It performed better in terms of scalability, stability, consistency, responsiveness, and total number of requests. The microservice system's average CPU consumption was 12% in TC5 and 10% in TC6 of the monolith's usage. However the microservice system's memory usage was higher, 135% in TC5 and 102% in TC6 of the monolith's usage.

- LT2: Complex Task

In TC7 and TC8, both applications experienced failures caused by concurrency issues. However the microservice system was able to repair itself during the test period to resolve the concurrency issue, thus it reported lower failure rates, around 50% lower in TC7 and 30% lower in TC8 compared to the monolith. In addition, the microservice application was faster in its scaling efforts and its performance graphs were less volatile. The total number of requests handled by the microservice system was 3.5 times and 4.1 times of what the monolithic application processed in TC7 and TC8. The microservice system's average CPU and memory consumption was 18% and 43% in TC7, and 17% and 36% in TC8 of the monolithic application's usage.

Vertical Scaling

In test cases TC9 - TC12, the Vertical Pod Autoscaler was enabled. The total amount of resources was the same for the two applications.

- LT1: Simple Task

In TC9 and TC10, both applications reported a 100% success rate. the performance charts of the microservice application indicated consistent stability while the monolithic software displayed two small erratic valleys in the Total Requests charts for unknown reasons. Despite both applications reporting satisfactory results, the microservice system returned better statistics in total number of requests, average response time, and average requests per second; it responded to ~2 times more user requests in both test cases. For resource expenditures, the average CPU consumption by the microservice system was only ~8% of the monolith's, and its memory usage was 9% of the monolithic application's usage.

- LT2: Complex Task

In TC11, both software systems encountered concurrency issues, which led to high failure rates, 20% for the microservice application and 30% for the monolithic application. On the whole, the microservice system handled 5 times more user requests with an average CPU and memory consumption of 12% and 59% of the monolith's usage. In TC12, the monolithic application experienced 3-4 restarts during the 10-minute test run and presented volatile performance charts, its average response time reached 12,736 ms, the highest in

all the test cases; while the microservice application faced the same concurrency issues as reported in previous test cases just 4 minutes into the test run, which resulted in a nearly 30% high failure rate. All in all, the performance results of the microservice system were still better as it handled over 100,000 more requests successfully despite its high failure rate with an average usage of 10% CPU and 75% memory of the monolithic application's consumption.

5 Discussion

5.1 Interpretation of Findings

The results of the load tests performed in the benchmarking process showed that for the simple task, both applications were able to deliver satisfactory results with nearly 100% success rate, but the microservice system returned much higher numbers in average requests per second and total number of requests. For the complex task, both applications experienced failures in test cases with all three different scaling scenarios. The failures were caused by concurrency issues in the applications which happened possibly due to the complex task requiring longer response time and increasing traffic causing a high accumulation of request pileup. All in all, the application in a microservice architecture delivered better performance results in metrics of the total number of requests, response times, and success rates. The total number of requests processed by the microservice application was numerous times over the monolithic application handled in nearly all the test cases. The charts of total requests and average response times of the microservice application in most test cases showed higher consistency and stability. In some test cases, the microservice system was able to resolve the concurrency issues by quickly restarting the problematic subsystem and lowering its failure rates. As for resource consumption, the microservice system consumed less CPU resources in all 12 test cases. The memory usage of the microservice application was slightly higher than the monolith's consumption in 4 out of 12 test cases, otherwise lower in the rest of the test cases.

5.2 Comparison with Previous Research

Based on the literature review results, monolithic applications required fewer resources and delivered better results in terms of throughput, latency, and response times for handling lower concurrent users. The monolithic architecture was proven to be a better choice for small and simple applications that do not have demanding requirements on high-traffic load handling. The microservice systems on the other hand were found to have advantages in handling high traffic loads; when deployed in cloud environments, they showed benefits in scalability, responsiveness, reliability, and stability. Some research

articles stated that microservice-based systems generated higher resource efficiency. The microservice architecture was reported to be more suitable for large-scale applications with complex business logic or software systems that target high numbers of concurrent users.

The load test results from the benchmarking process indicated similar conclusions as in some of the reviewed literature that applications in microservice architectures outperformed in handling heavy traffic loads, especially when deployed in cloud environments where they benefit from advantages such as scalability, efficiency, and resource utilization effectiveness. However, the load test statistics also revealed that the microservice-based system performed far better when handling both simple and complex tasks with various concurrent user loads and had much lower CPU consumption in all test cases and lower memory usage in most of the test cases, conflicting with the findings from some of the existing research results that monolithic applications showed advantages in simple task handling and resource requirements, especially for applications of small scale and low complexity, which are considered the characteristics of the applications used in the load testing.

5.3 Limitations

It is worth mentioning that during the search process for articles, it was found that the number of research papers on this topic was very limited. All the papers that included empirical research on this topic were selected for the literature review in this thesis. Some of the papers were just a few pages long without detailed documentation on the research tools and processes, and lacked descriptions of test environments, resource allocation, or test scenarios. Most of the tests were performed in environments with limited amounts of resources. A big portion of the applications used in the research papers were small demo projects developed solely for research purposes. In addition, the applications used for the load tests in the benchmarking experiment of this thesis are relatively small with a limited number of components and the most basic functionalities. The microservice-based program contains as few as five subsystems. The applications were developed for research on software architectures instead of load testing. Therefore the application designers and developers may not have taken into account concurrency issues that may appear during high traffic loads. The environment setup and resource allocation were limited by the free trial plan offered by GKE. All those factors can to some extent reduce the validity of the test results. Therefore the research results may have discrepancies from real-life

applications.

6 Conclusions

Despite being more conventional, the monolithic architecture is believed to be more advantageous than the microservice architecture, particularly for smaller projects and prototyping efforts. Microservice architecture has been gaining popularity in recent years, especially with the advancement of cloud technologies. A growing number of applications are being developed in this architecture. Many legacy software systems have been migrated to use this architecture. However, which one is a better architecture may depend on various factors. The purpose of this thesis was to study and compare the differences in application performance and resource consumption between two different software architectures, namely the traditional monolithic architecture and the relatively novel microservice architecture.

To fulfill the research purpose of this thesis, a systematic literature review was performed with previous research papers on this topic to understand the existing research results. In addition, a benchmarking process with load testing was implemented on two software systems with identical functionalities using the two above-mentioned software architectures.

The results of the literature review showed that monolithic programs normally required fewer resources and performed better in terms of throughput, latency, and response times when managing fewer concurrent users. It has been demonstrated that monolithic design is a preferable option for small-scale systems without strict demands on handling heavy traffic loads. On the other hand, it was discovered that microservice systems are more efficient at handling large amounts of traffic; they demonstrated advantages in terms of scalability, responsiveness, stability, and reliability in cloud environments. Microservice-based systems were reported to produce improved resource efficiency in some study studies. The overall findings suggested that the microservice architecture suites better software systems intended for huge numbers of concurrent users or large-scale applications with intricate business logic.

The outcomes of the load tests demonstrated that applications in both architectures could produce 100% successful results in handling simple tasks under different numbers of concurrent users from 100 to 1,000. Both applications encountered failures in test cases under all three of the scaling scenarios for the complex task. However, the overall test results showed that the microservice system delivered better statistics concerning the total num-

ber of requests, average response times, and success rates in nearly all test cases as it handles exponentially more requests than the monolithic application did. The application performance charts also indicated that the microservice-based system was more advantageous in terms of scalability, responsiveness, and stability. When it comes to resource consumption, the application with a microservice architecture reported much higher CPU resource efficiency as it handled substantially greater numbers of requests with a small fraction of the CPU usage by the software in a monolithic architecture. For memory resources, in 9 out of 12 test cases, the microservice system's usage was lower, and in the rest of the test cases, the memory consumption of the microservice system was slightly higher than the monolithic application. Therefore it can be concluded that for the two applications used in this benchmarking experiment, the microservice-based system delivered much better results in terms of both application performance and resource efficiency.

The test results in the benchmarking process echoed the research findings from some of the reviewed literature that microservice systems deliver better results in terms of throughput, efficiency, scalability, reliability, and stability as they are more advantageous at handling heavy traffic and saving costs on infrastructure resources, especially in cloud environments. However, the monolithic application used in the benchmarking process failed to show the advantages reported in some of the existing research papers, even though it delivered satisfactory performance outcomes. In its overall form, the research findings of this thesis demonstrated that both architectures have advantages and drawbacks. Applications in different architectures can perform well and deliver desired achievements when properly implemented. Nevertheless, the research outcomes indicated the performance and resource consumption of applications created using different software architectures depend on a wide range of factors, especially application scale and complexity, deployment environments, and user traffic.

Bibliography

Abbas, R., Sultan, Z., and Bhatti, S. N. (2017). "Comparative analysis of automated load testing tools: Apache JMeter, Microsoft Visual Studio (TFS), LoadRunner, Siege". In: *2017 International Conference on Communication Technologies (ComTech)*, pp. 39–44. DOI: 10.1109/COMTECH.2017.8065747.

Adrio, K., Tanzil, C. N., Lianto, M. C., and Rasjid, Z. E. (2023). "Comparative Analysis of Monolith, Microservice API Gateway and Microservice Federated Gateway on Web-based application using GraphQL API". In: *2023 10th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, pp. 654–660. DOI: 10.1109/EECSI59885.2023.10295809.

Almeida, J. F. and Silva, A. R. (2020). "Monolith Migration Complexity Tuning Through the Application of Microservices Patterns". In: *Software Architecture*. Ed. by A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, and O. Zimmermann. Cham: Springer International Publishing, pp. 39–54. ISBN: 978-3-030-58923-3.

Benavente, V., Yantas, L., Moscol, I., Rodriguez, C., Inquilla, R., and Pomachagua, Y. (2022). "Comparative Analysis of Microservices and Monolithic Architecture". In: *2022 14th International Conference on Computational Intelligence and Communication Networks (CICN)*, pp. 177–184. DOI: 10.1109/CICN56167.2022.10008275.

Bengtsson, P. and Bosch, J. (1999). "Architecture level prediction of software maintenance". In: *Proceedings of the Third European Conference on Software Maintenance and Reengineering (Cat. No. PR00090)*, pp. 139–147. DOI: 10.1109/CSMR.1999.756691.

Blinowski, G., Ojdowska, A., and Przybyłek, A. (2022). "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation". In: *IEEE Access* 10, pp. 20357–20374. DOI: 10.1109/ACCESS.2022.3152803.

Chen, T.-H., Syer, M. D., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., and Flora, P. (2017). "Analytics-Driven Load Testing: An Industrial Experience Report on Load Testing of Large-Scale Systems". In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 243–252. DOI: 10.1109/ICSE-SEIP.2017.26.

De Lauretis, L. (2019). "From Monolithic Architecture to Microservices Architecture". In: DOI: 10.1109/ISSREW.2019.00050.

Al-Debagy, O. and Martinek, P. (2018). "A Comparative Review of Microservices and Monolithic Architectures". In: *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 000149–000154. DOI: 10.1109/CINTI.2018.8928192.

Dobrica, L. and Niemela, E. (2002). "A survey on software architecture analysis methods". In: *IEEE Transactions on Software Engineering* 28.7, pp. 638–653. DOI: 10.1109/TSE.2002.1019479.

Esposito, C., Castiglione, A., and Choo, K.-K. R. (2016). "Challenges in Delivering Software in the Cloud as Microservices". In: *IEEE Cloud Computing* 3.5, pp. 10–14. DOI: 10.1109/MCC.2016.105.

Flygare, R. (2017). "Performance characteristics between monolithic and microservice-based systems". In: *Bachelor Thesis Software Engineering urn:nbn:se:bth-14888 06-2017*. URL: https://www.diva-portal.org/smash/get/diva2:1119785/FULLTEXT03.pdf.

Gong, C., Liu, J., Zhang, Q., Chen, H., and Gong, Z. (2010). "The Characteristics of Cloud Computing". In: *2010 39th International Conference on Parallel Processing Workshops*, pp. 275–279. DOI: 10.1109/ICPPW.2010.45.

Gos, K. and Zabierowski, W. (2020). "The Comparison of Microservice and Monolithic Architecture". In: *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pp. 150–153. DOI: 10.1109/MEMSTECH49584.2020.9109514.

Grafana, L. (2024). In: URL: https://grafana.com/load-testing/types-of-load-testing/.

Iris, B., Manuel, H., and Robert, M. (2019). "IMPLEMENTATION OF A WEB-BASED AUDIENCE RESPONSE SYSTEM AS MICROSERVICE APPLICATION VS. MONOLITHIC APPLICATION". In: *IADIS digital library*, pp. 27–34.

Islam, R., Patamsetti, V., Gadhi, A., Gondu, R. M., Bandaru, C. M., Kesani, S. C., and Abiona, O. (2023). "The future of cloud computing: benefits and challenges". In: *International Journal of Communications, Network and System Sciences* 16.4, pp. 53–65.

Jadeja, Y. and Modi, K. (2012). "Cloud computing - concepts, architecture and challenges". In: *2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET)*, pp. 877–880. DOI: 10.1109/ICCEET.2012.6203873.

Jatkiewicz, P. and Okrój, S. (2023). "Differences in performance, scalability, and cost of using microservice and monolithic architecture". In: *Proceedings of the 38th ACM/SIGAPP*

Symposium on Applied Computing. SAC '23. Tallinn, Estonia: Association for Computing Machinery, pp. 1038–1041. ISBN: 9781450395175. DOI: 10.1145/3555776.3578725. URL: https://doi.org/10.1145/3555776.3578725.

Kazman, R., Bass, L., Abowd, G., and Webb, M. (1994). "SAAM: a method for analyzing the properties of software architectures". In: *Proceedings of 16th International Conference on Software Engineering*, pp. 81–90. DOI: 10.1109/ICSE.1994.296768.

Lauwren, A. J. (2022). "MICROSERVICE AND MONOLITH PERFORMANCE COMPARISON IN TRANSACTION APPLICATION". PhD thesis. Universitas Katholik Soegijapranata Semarang. URL: http://repository.unika.ac.id/id/eprint/28277.

Linthicum, D. S. (2016). "Practical Use of Microservices in Moving Workloads to the Cloud". In: *IEEE Cloud Computing* 3.5, pp. 6–9. DOI: 10.1109/MCC.2016.114.

Mangwani, P. and Tokekar, V. (2022). "Container Based Scalability and Performance Analysis of Multitenant SaaS Applications". In: *2022 13th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pp. 1–6. DOI: 10.1109/ICCCNT54827.2022.9984214.

Mario, V., Oscar, G., Lina, O., Harold, C., Lorena, S., Mauricio, V., Rubby, C., Santiago Giland Carlos, V., Angee, Z., and Mery, L. (2017). "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures". In: *Service Oriented Computing and Applications* 11.2, pp. 233–247. DOI: 10.1007/s11761-017-0208-y. URL: https://doi.org/10.1007/s11761-017-0208-y.

Microsoft (2024). "Migrate a monolithic application to microservices using domain-driven design". In: URL: https://learn.microsoft.com/en-us/azure/architecture/microservices/migrate-monolith.

Microsoft, A. (2024). "what-is-the-cloud". In: URL: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-the-cloud.

Mohamed, N., Moussa, S., Badr, N., and Tolba, M. (Feb. 2021). "Enhancing Test Cases Prioritization for Internet of Things based systems using Search-based Technique". In: *International Journal of Intelligent Computing and Information Sciences* 21, pp. 84–94. DOI: 10.21608/ijicis.2021.69462.1076.

Montazerolghaem, A., Yaghmaee, M. H., and Leon-Garcia, A. (2020). "Green Cloud Multimedia Networking: NFV/SDN Based Energy-Efficient Resource Allocation". In: *IEEE Transactions on Green Communications and Networking* 4.3, pp. 873–889. DOI: 10.1109/TGCN.2020.2982821.

Namiot, D. and sneps-sneppe, M. (Sept. 2014). "On Micro-services Architecture". In: *Interenational Journal of Open Information Technologies* 2, pp. 24–27.

Napawit, T. and Waraporn, V. (2021). "Performance Comparison between Monolith and Microservice Using Docker and Kubernetes". In: *International Journal of Computer Theory and Engineering* 13.3, pp. 91–95.

Nayim, N. N., Karmakar, A., Ahmed, M. R., Saifuddin, M., and Kabir, M. H. (2023). "Performance Evaluation of Monolithic and Microservice Architecture for an E-commerce Startup". In: *2023 26th International Conference on Computer and Information Technology (ICCIT)*, pp. 1–5. DOI: 10.1109/ICCIT60459.2023.10441241.

OpenAI (2024). URL: https://chat.openai.com.

Plessis, S. du and Correia, N. (2021). "A Comparative Study of Software Architectures in Constrained Device IoT Deployments". In: *2021 IEEE International Conference on Internet of Things and Intelligence Systems (IoTaIS)*, pp. 35–41. DOI: 10.1109/IoTaIS53735.2021.9628703.

Ponce, F., Márquez, G., and Astudillo, H. (2019). "Migrating from monolithic architecture to microservices: A Rapid Review". In: pp. 1–7. DOI: 10.1109/SCCC49216.2019.8966423.

Poonam, M., Niti, M., and Sachin, M. (2023). "Evaluation of a Multitenant SaaS Using Monolithic and Microservice Architectures". In: *SN Computer Science* 4 (2), p. 185. ISSN: 2661-8907. DOI: 10.1007/s42979-022-01610-2. URL: https://doi.org/10.1007/s42979-022-01610-2.

Raharjo, A. B., Andyartha, P. K., Wijaya, W. H., Purwananto, Y., Purwitasari, D., and Juniarta, N. (2022). "Reliability Evaluation of Microservices and Monolithic Architectures". In: *2022 International Conference on Computer Engineering, Network, and Intelligent Multimedia (CENIM)*, pp. 1–7. DOI: 10.1109/CENIM56801.2022.10037281.

Ray, P. P. (2018). "An Introduction to Dew Computing: Definition, Concept and Implications". In: *IEEE Access* 6, pp. 723–737. DOI: 10.1109/ACCESS.2017.2775042.

Saransig, A. and Tapia Leon, F. (Jan. 2019). "Performance Analysis of Monolithic and Micro Service Architectures – Containers Technology: Proceedings of the 7th International Conference on Software Process Improvement (CIMPS 2018)". In: pp. 270–279. ISBN: 978-3-030-01170-3. DOI: 10.1007/978-3-030-01171-0_25.

Tapia, F., Mora, M. Á., Fuertes, W., Aules, H., Flores, E., and Toulkeridis, T. (2020). "From Monolithic Systems to Microservices: A Comparative Study of Performance". In: *Applied Sciences* 10.17. ISSN: 2076-3417. DOI: 10.3390/app10175797. URL: https://www.mdpi.com/2076-3417/10/17/5797.

Velepucha, V. and Flores, P. (2021). “Monoliths to microservices - Migration Problems and Challenges: A SMS”. In: pp. 135–142. DOI: 10.1109/ICI2ST51859.2021.00027.

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., and Gil, S. (2015). “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud”. In: *2015 10th Computing Colombian Conference (10CCC)*, pp. 583–590. DOI: 10.1109/ColumbianCC.2015.7333476.

Wescott, B. (2013). *The Every Computer Performance Book , Chapter 6: Load Testing*. CreateSpace.

Zhang, H., Li, S., Jia, Z., Zhong, C., and Zhang, C. (2019). “Microservice Architecture in Reality: An Industrial Inquiry”. In: pp. 51–60. DOI: 10.1109/ICSA.2019.00014.

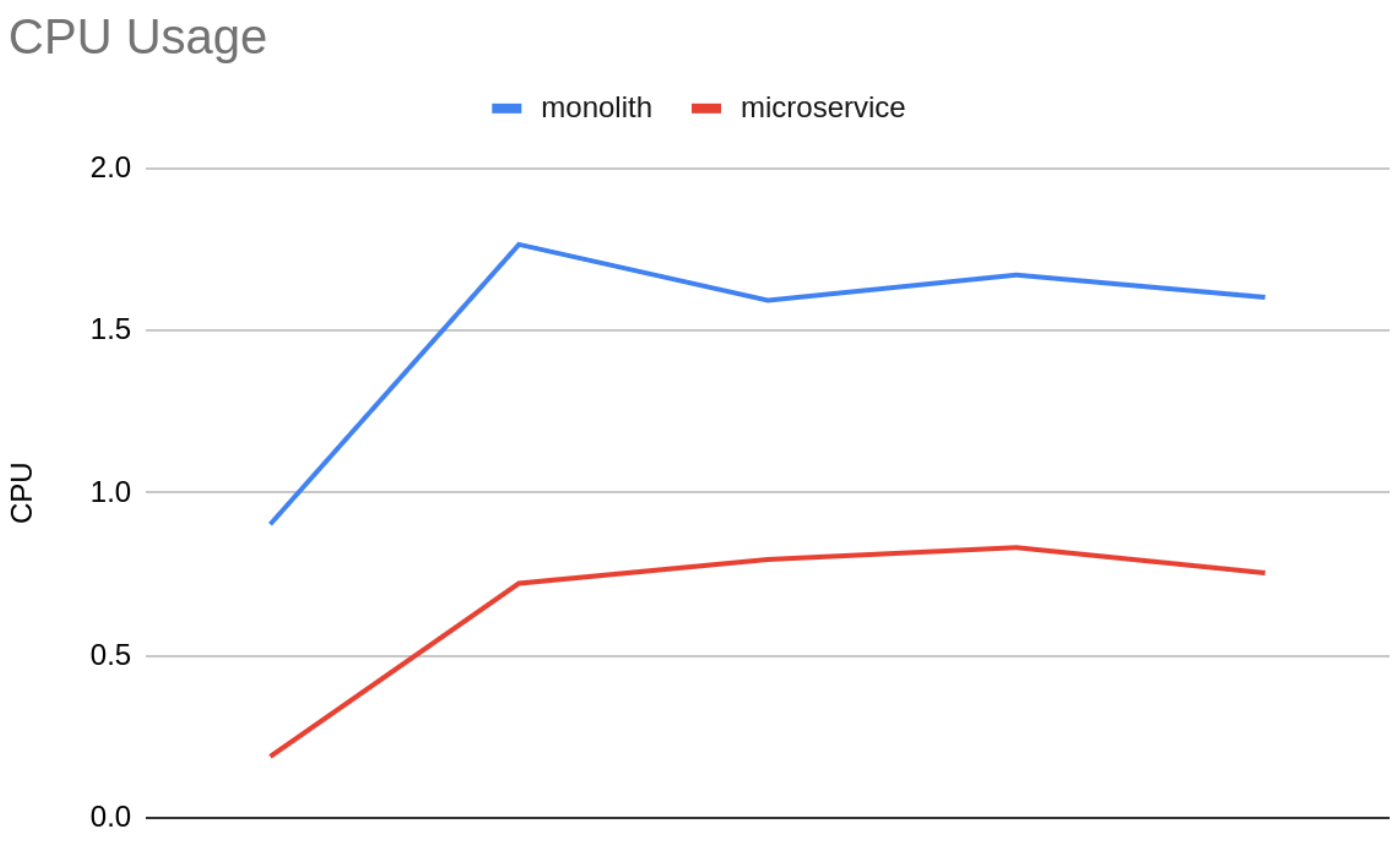
Appendix A Load Test Report

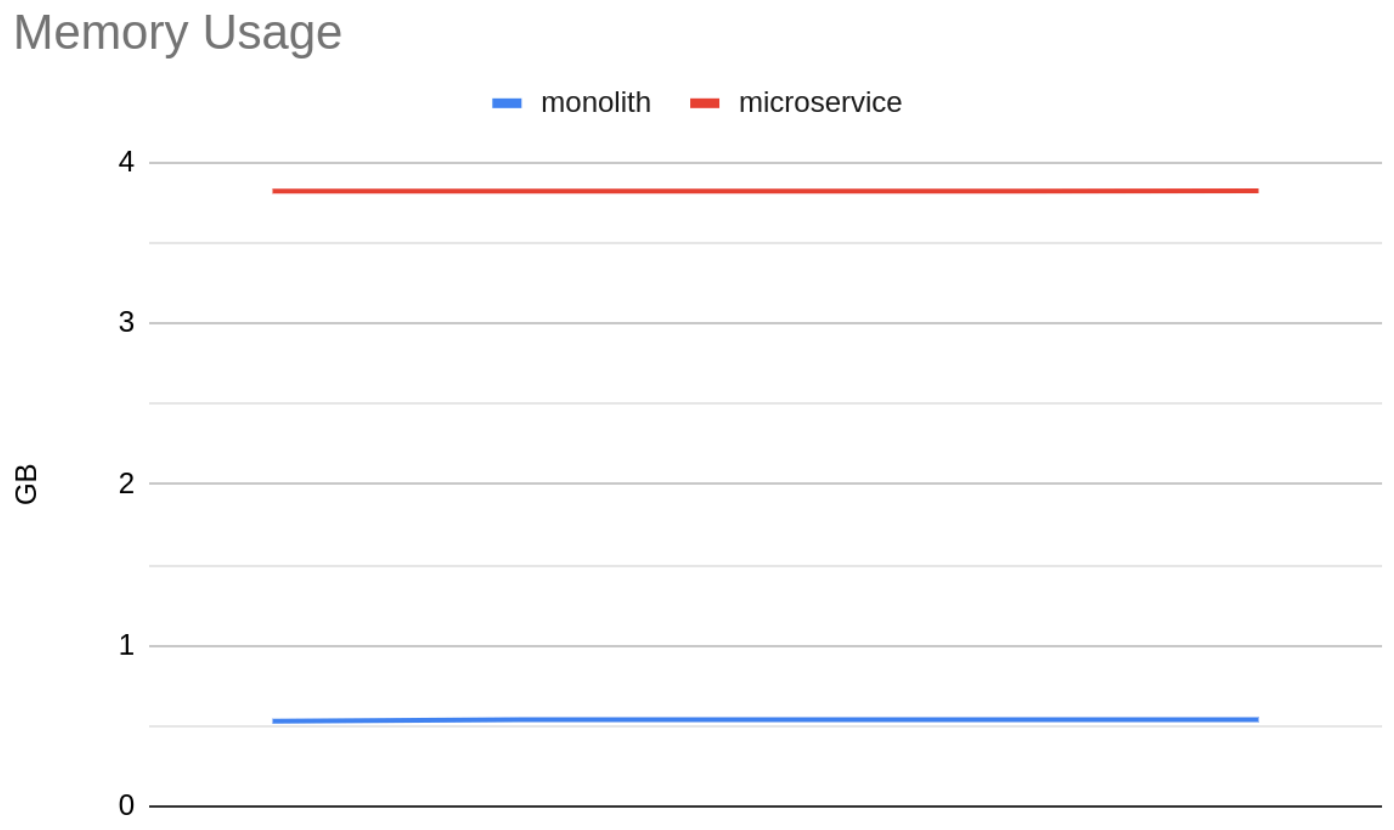
Charts of Resource Consumption and Application Performance

The CPU and memory consumption charts and Locust reports gathered from load tests are shown in the following PDF document: Resource Consumption and Application Performance.

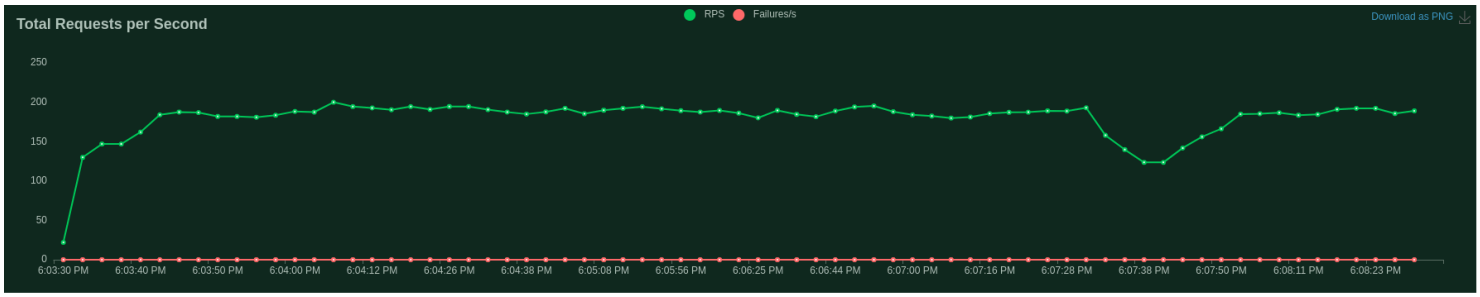
Resource Consumption and Application Performance

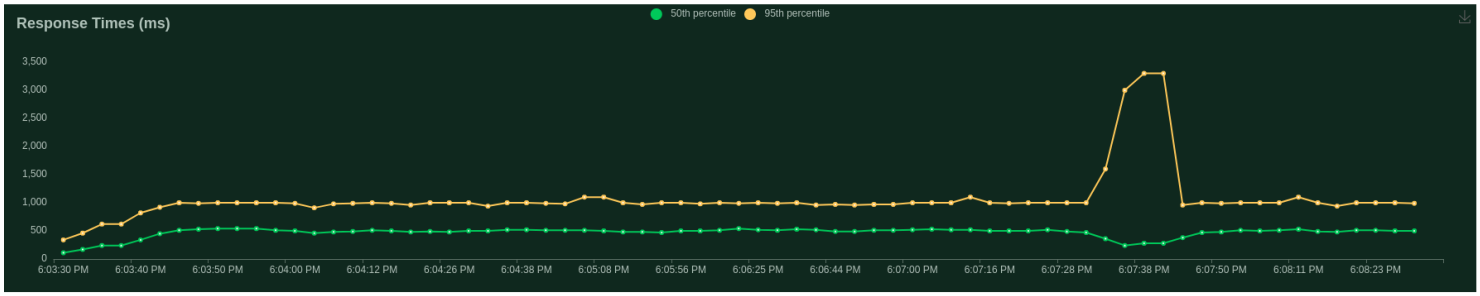
TC1

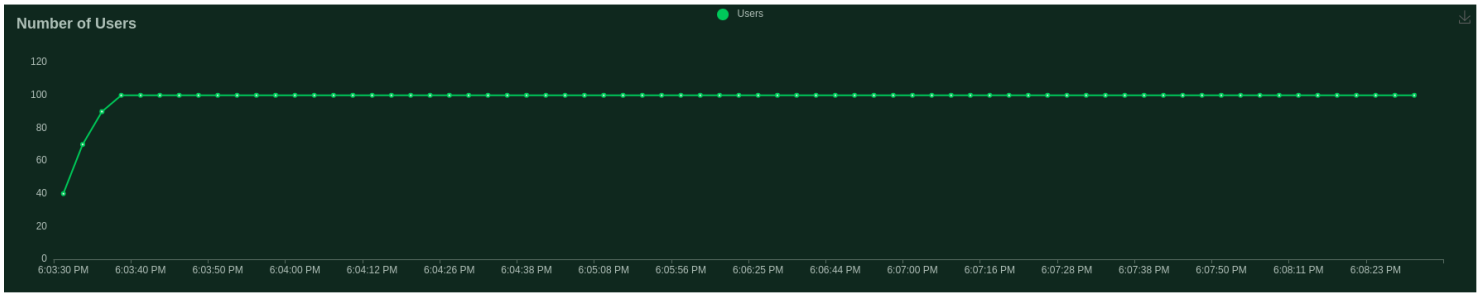




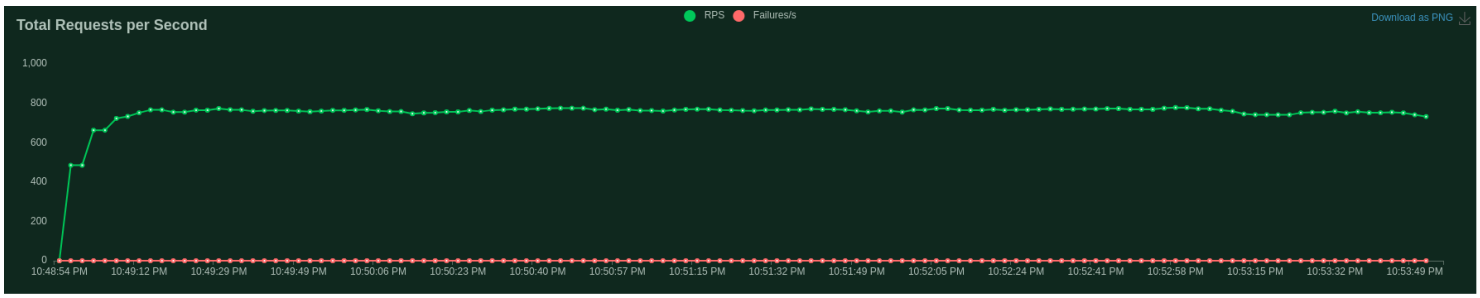
Locust Report: monolith

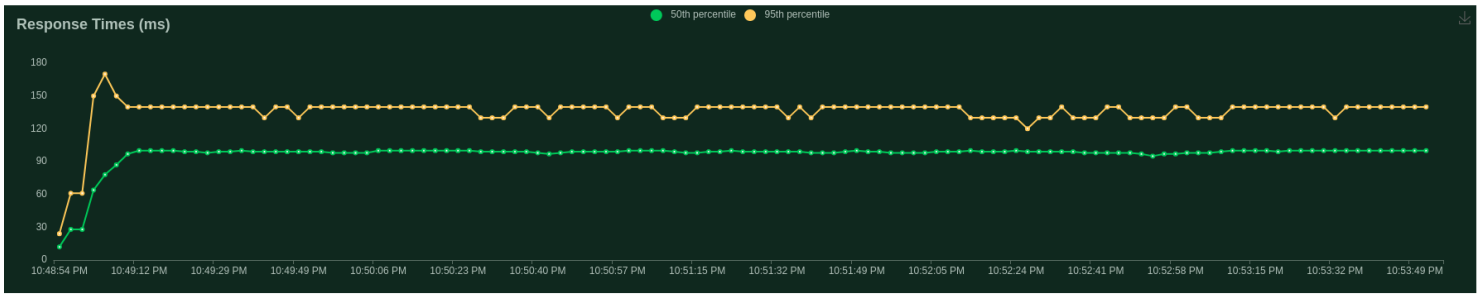


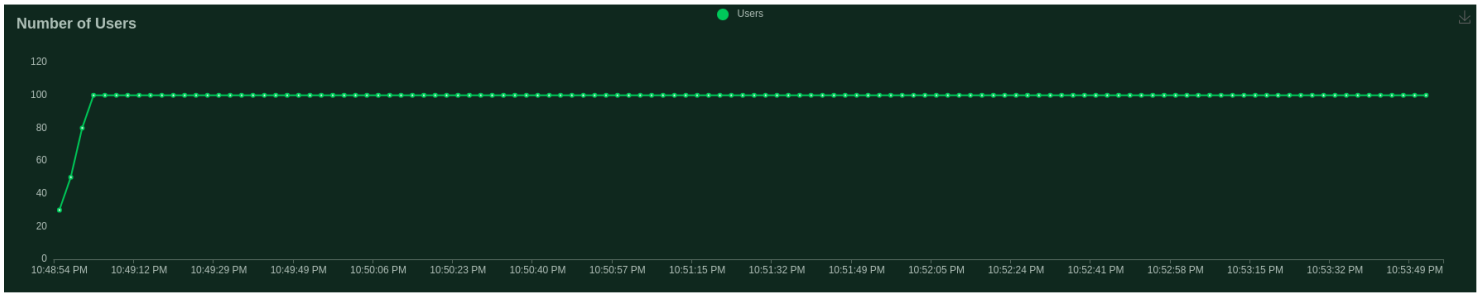




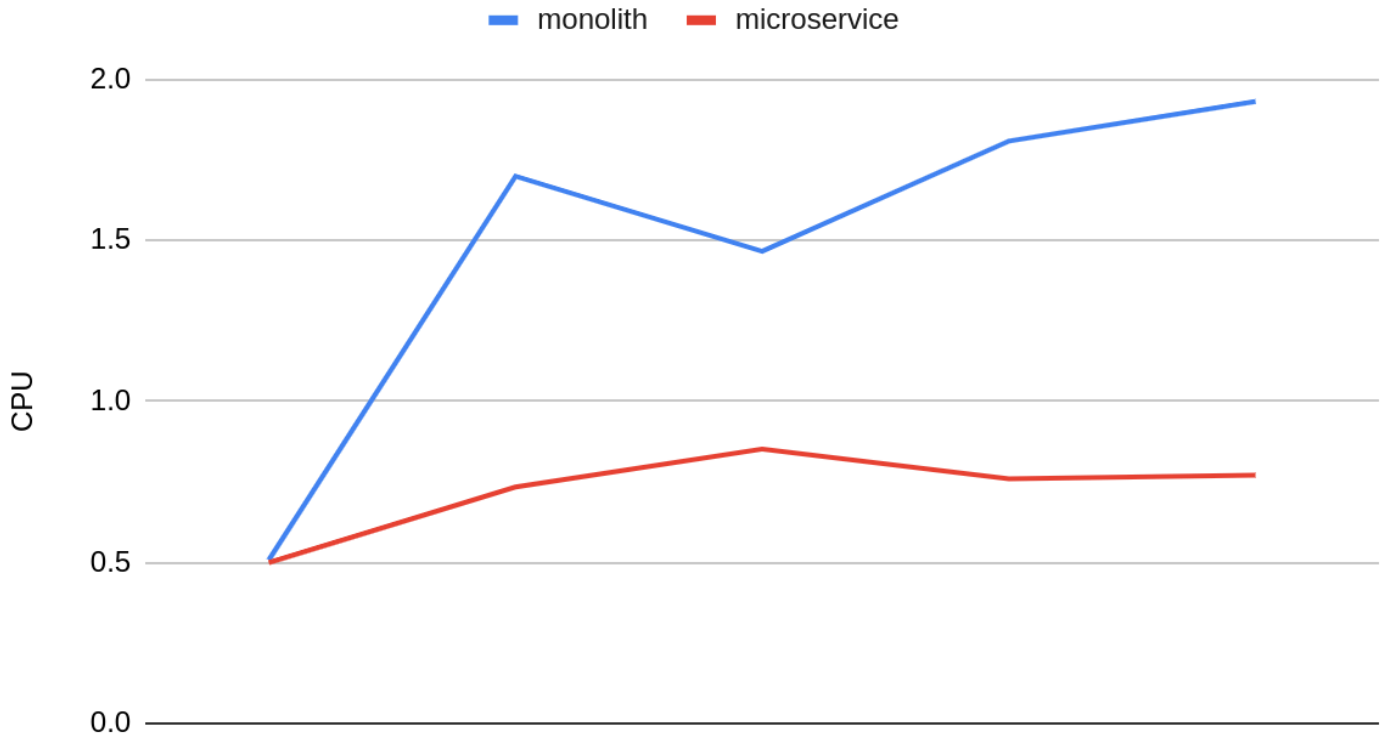
Locust Report: microservice



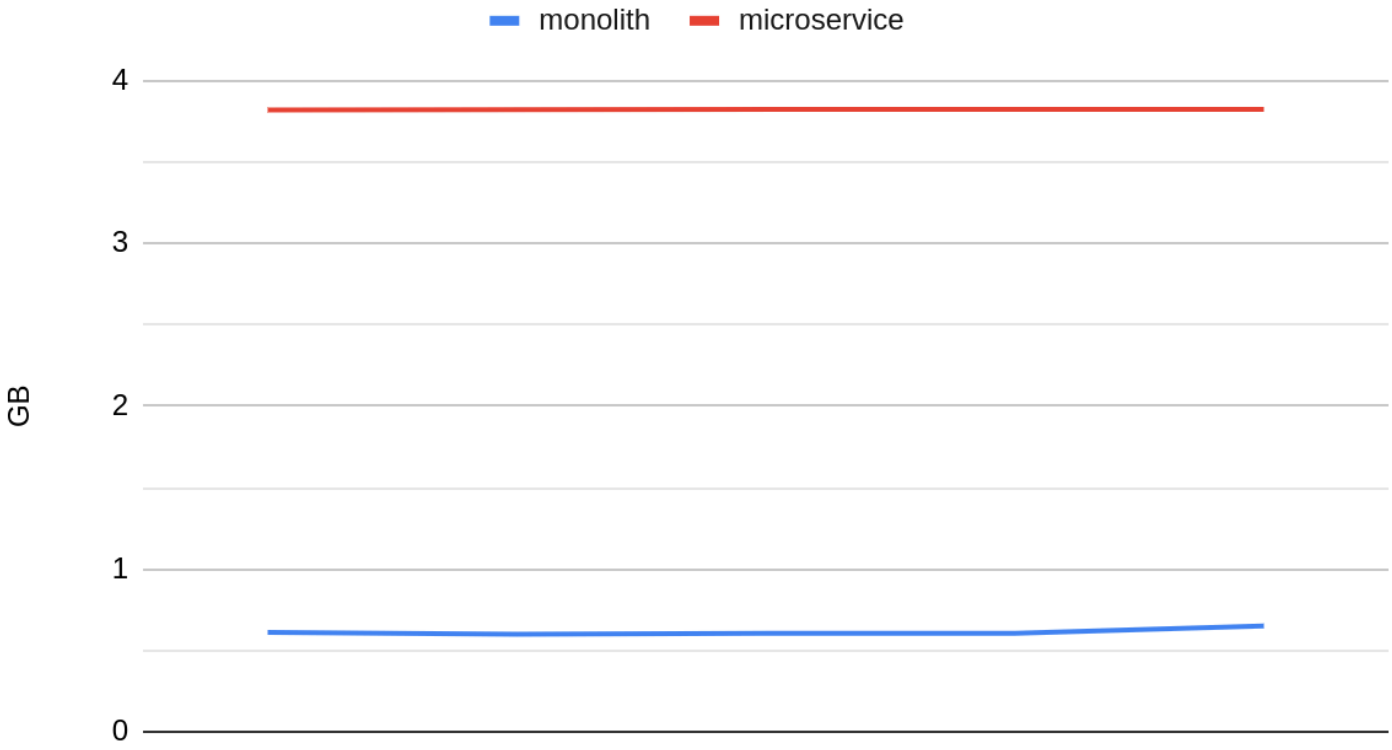




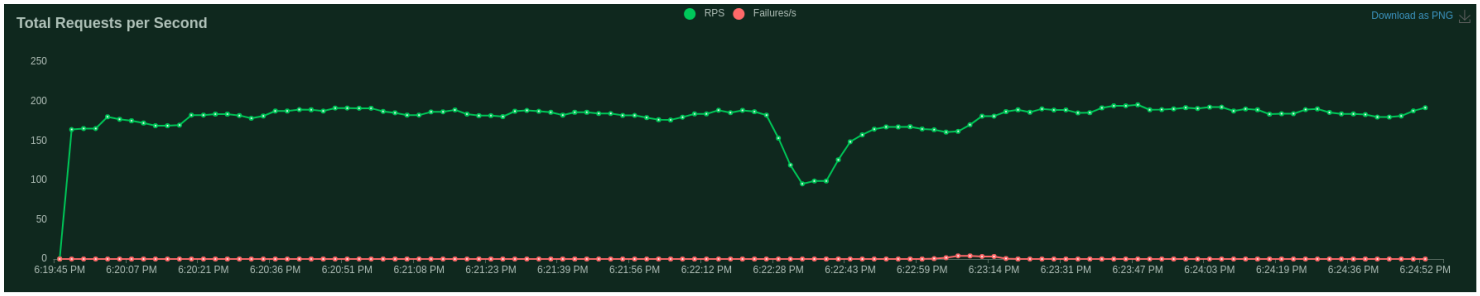
CPU Usage

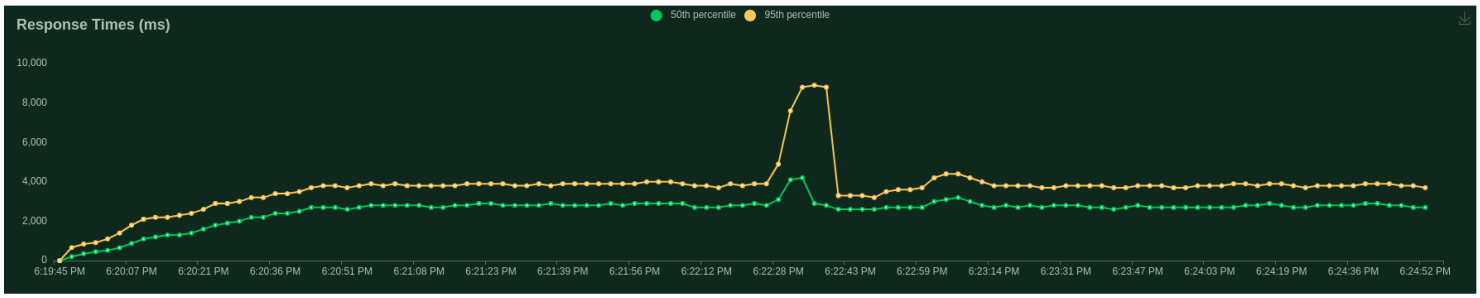


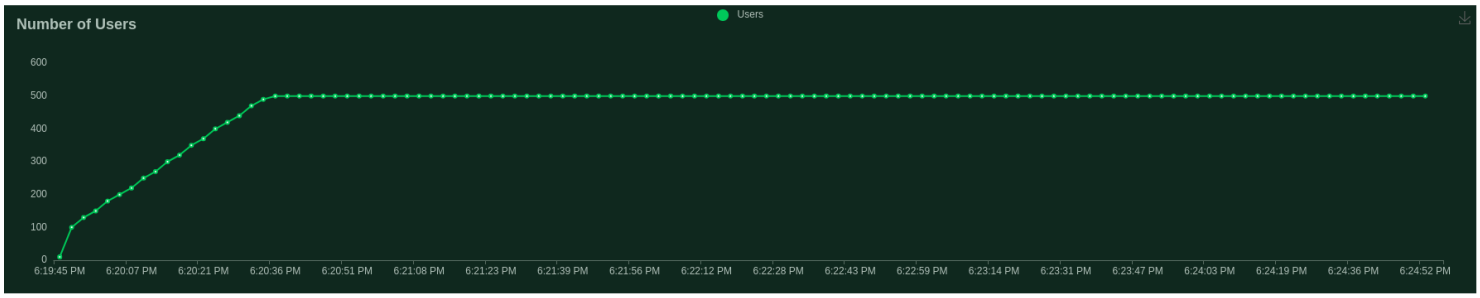
Memory Usage



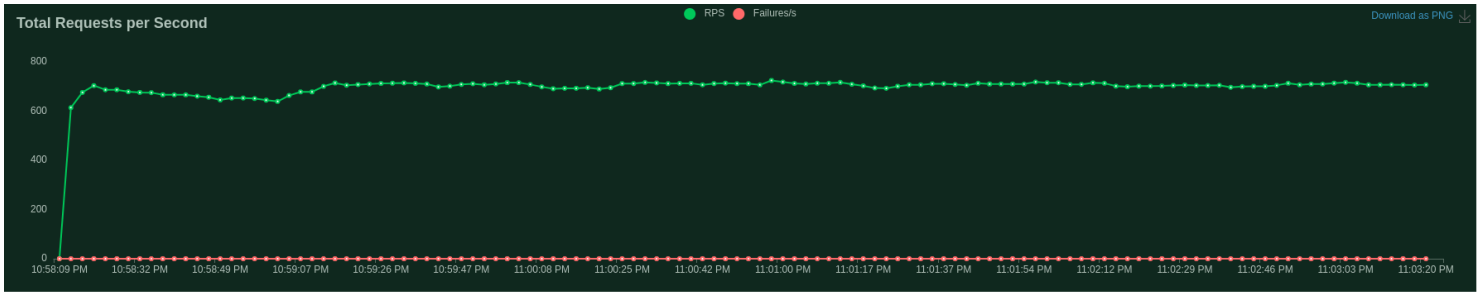
Locust Report: monolith

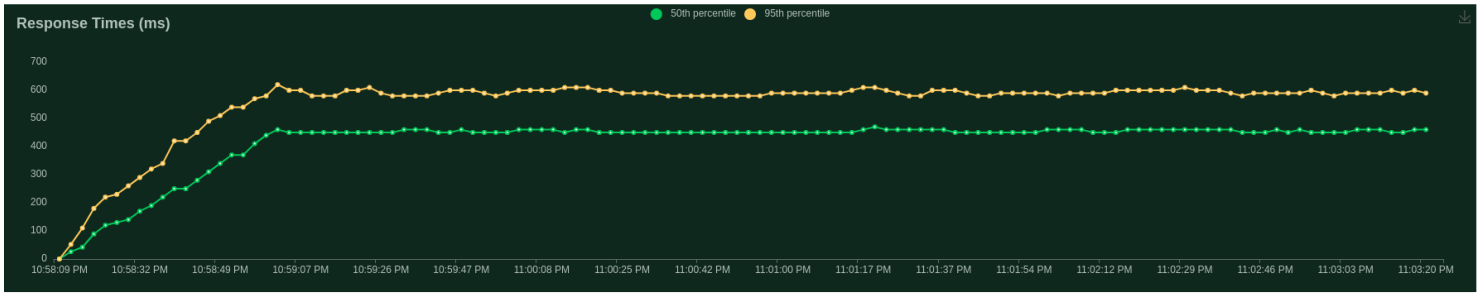


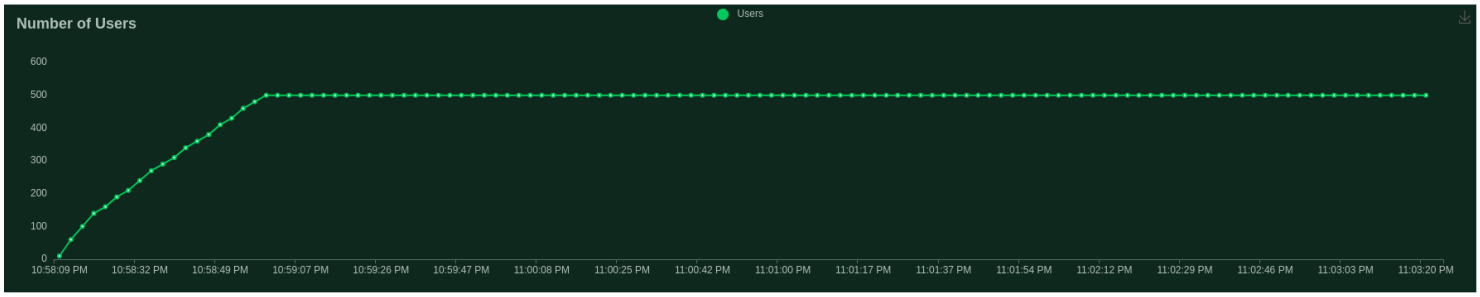




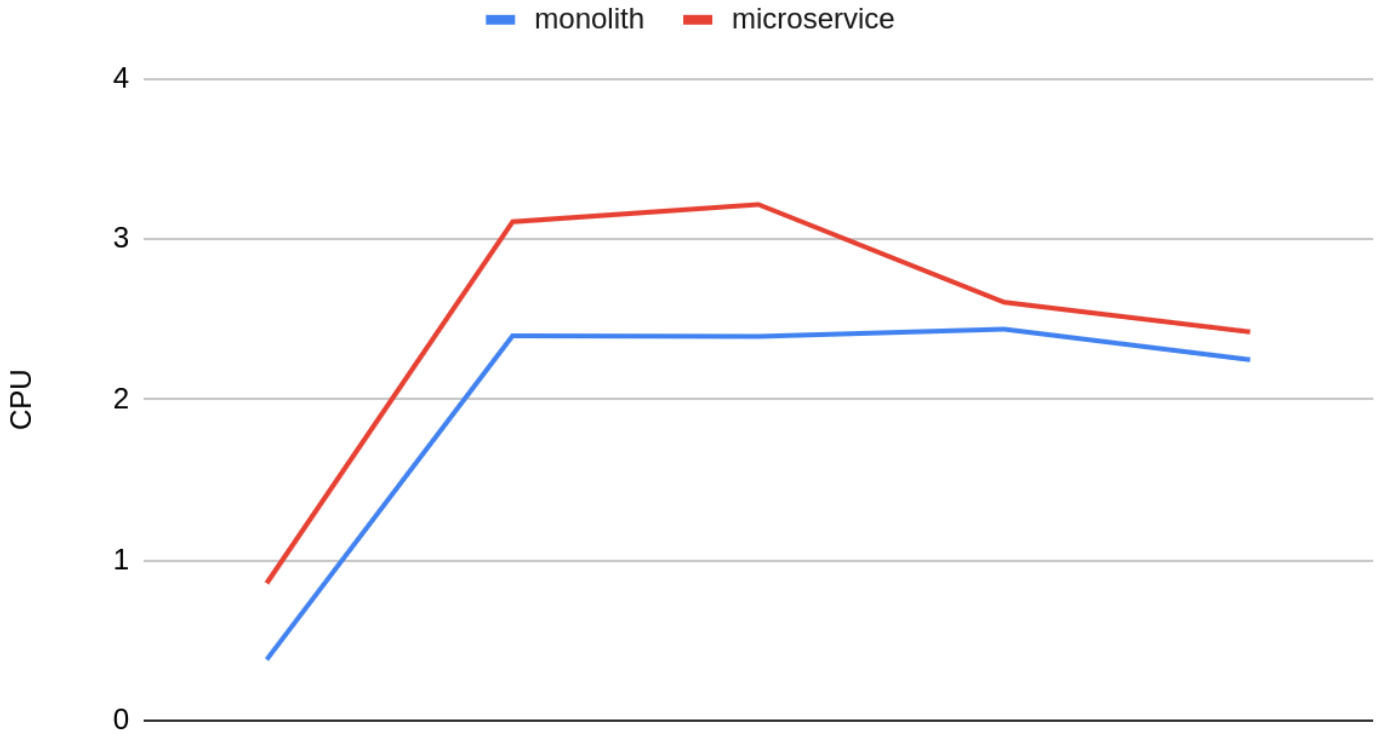
Locust Report: microservice



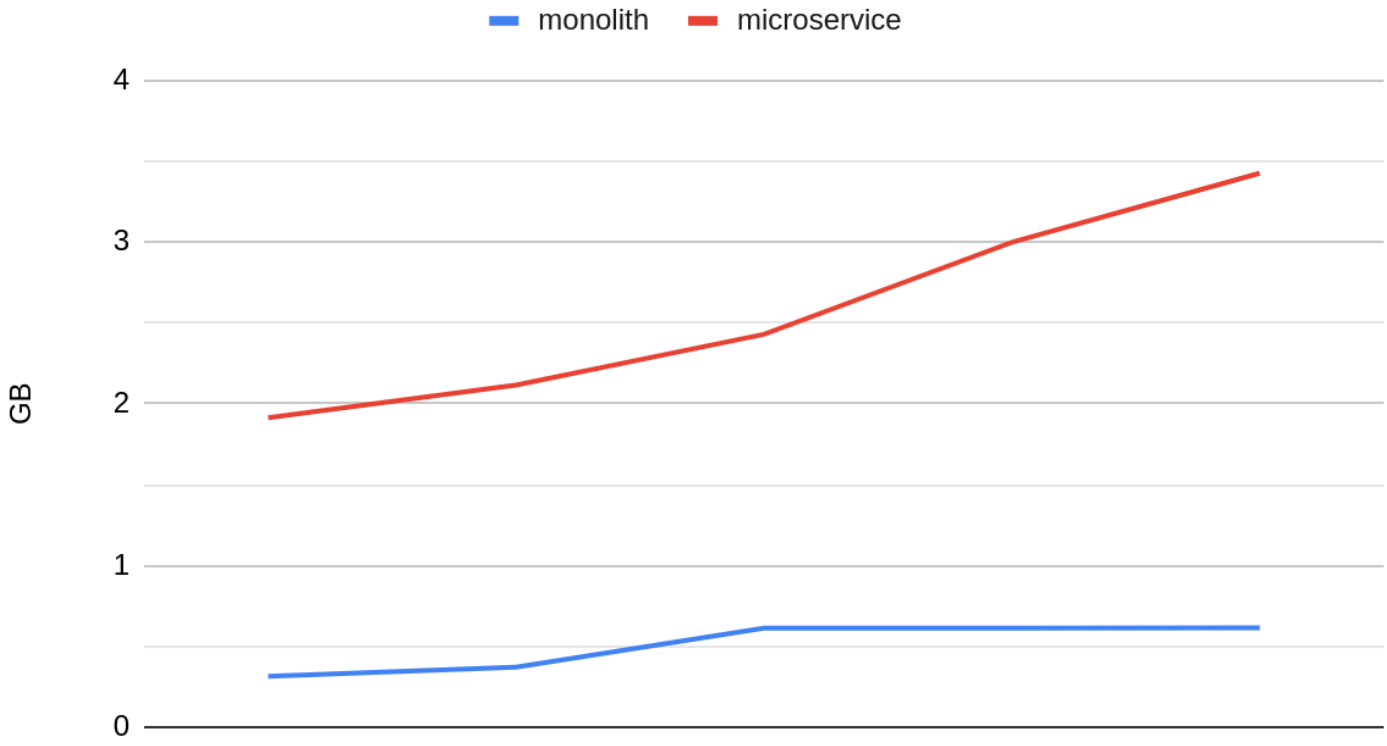




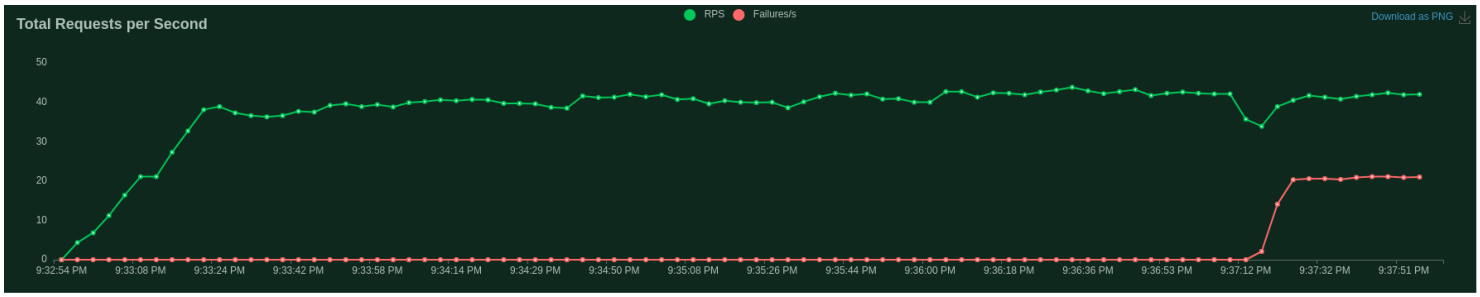
CPU Usage

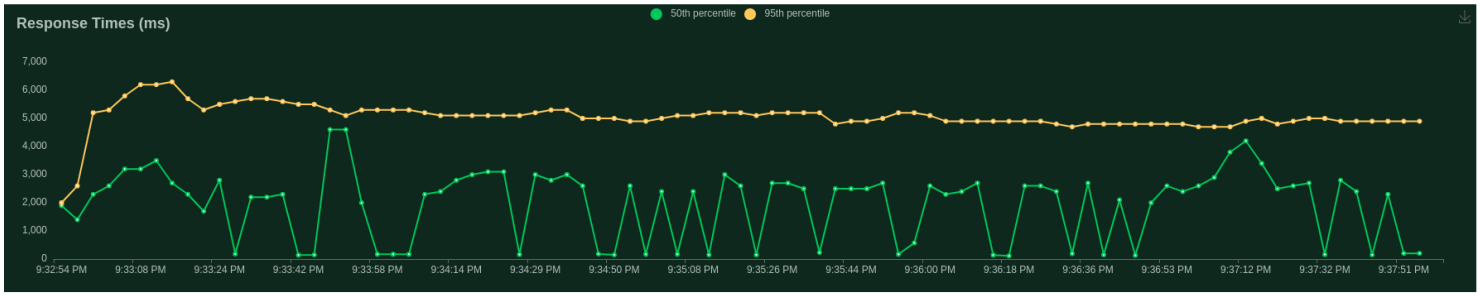


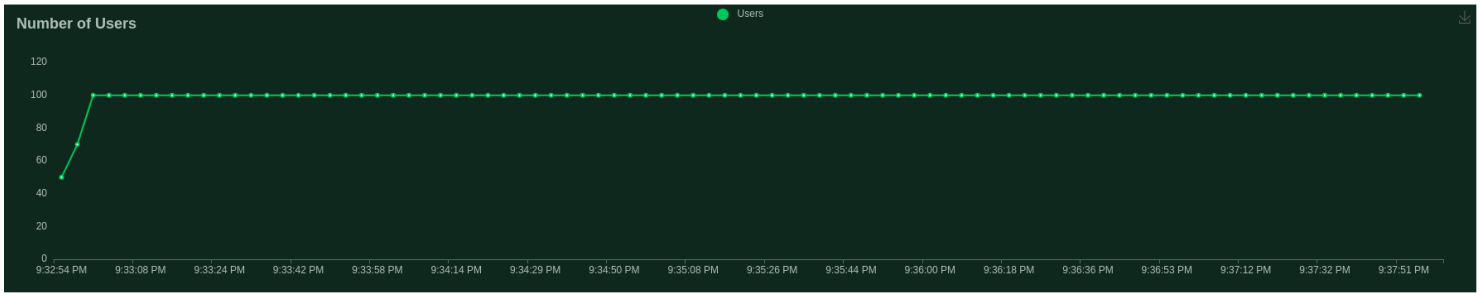
Memory Usage



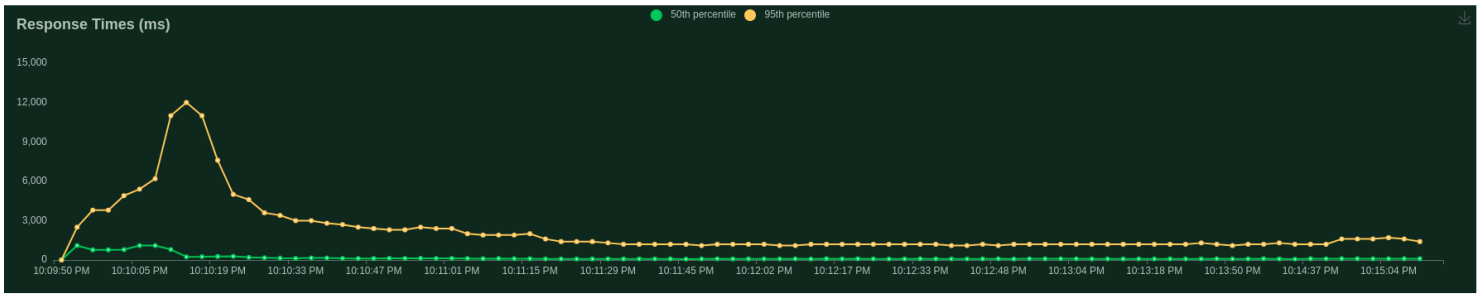
Locust Report: monolith

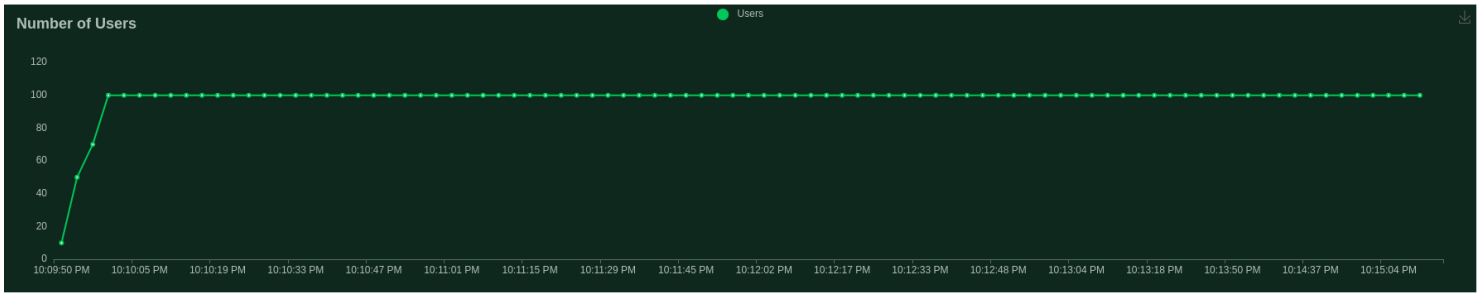




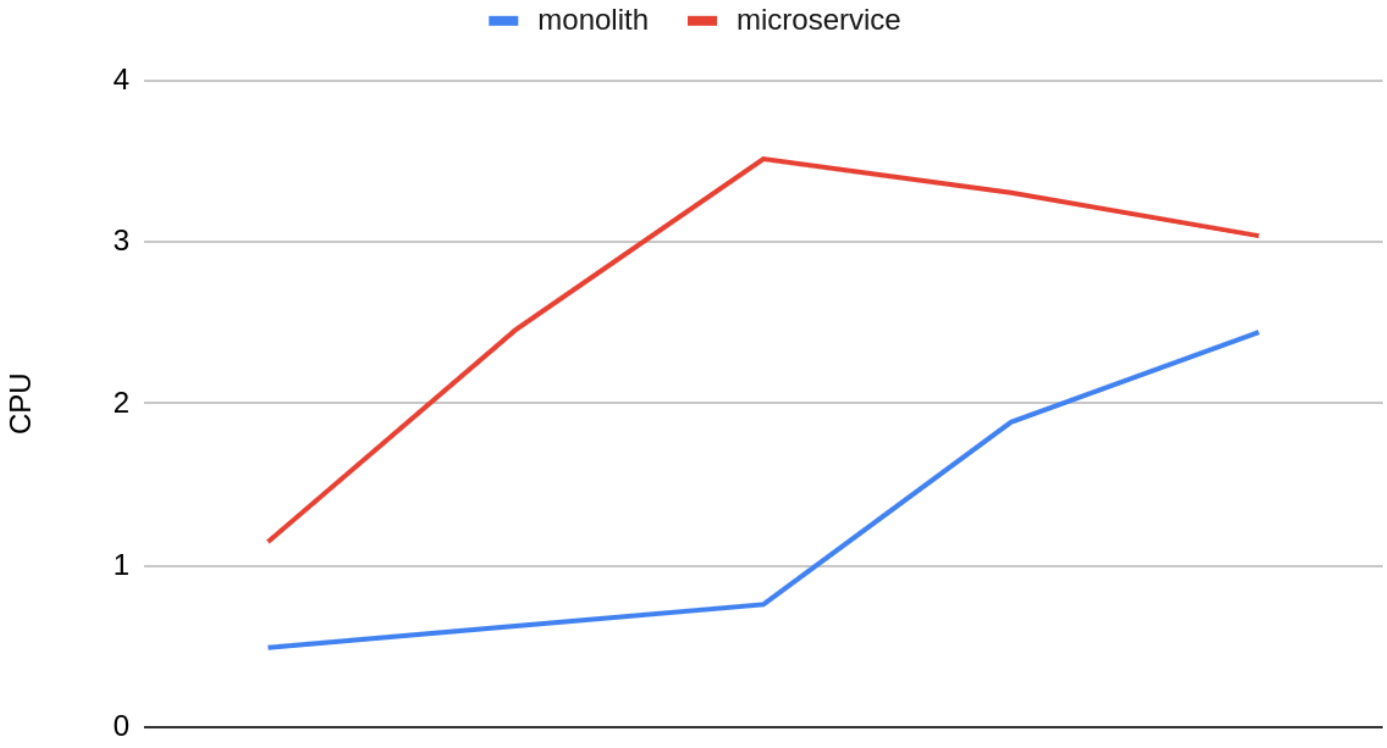


Locust Report: microservice

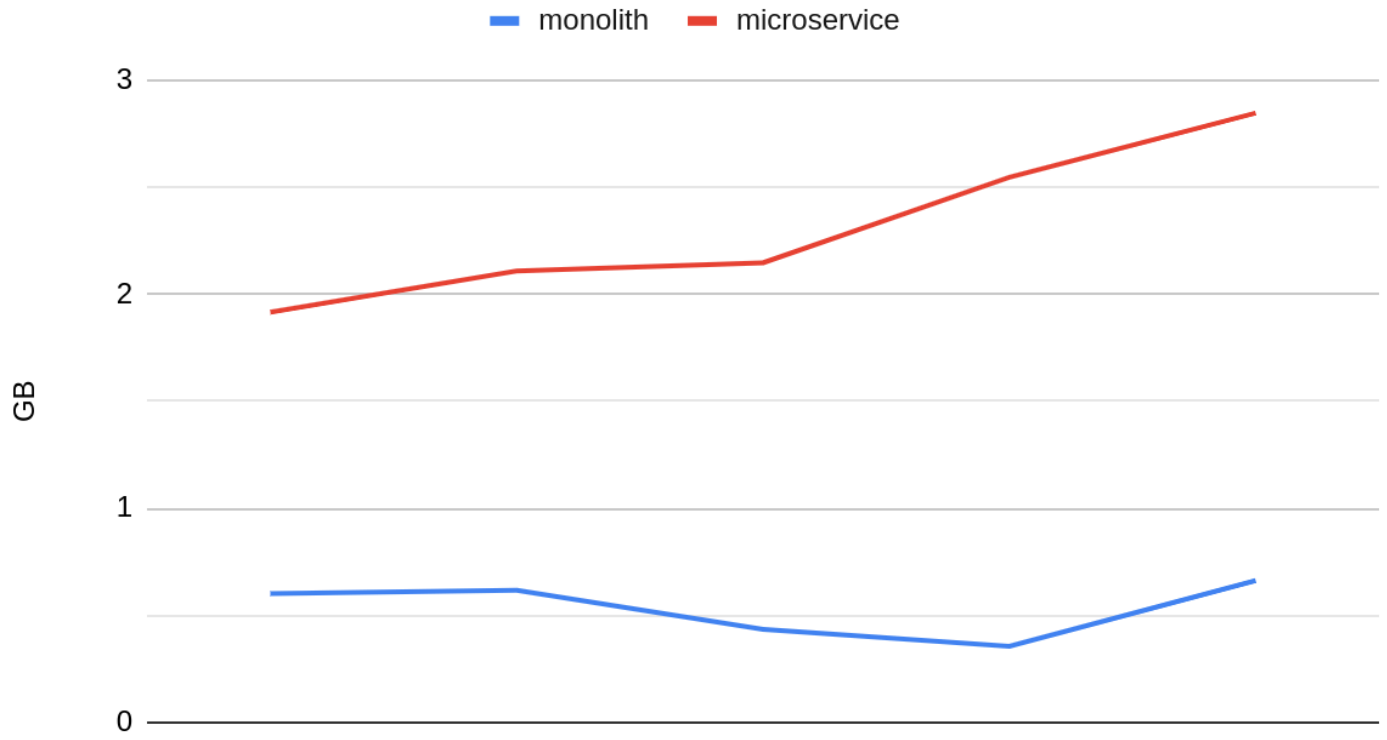




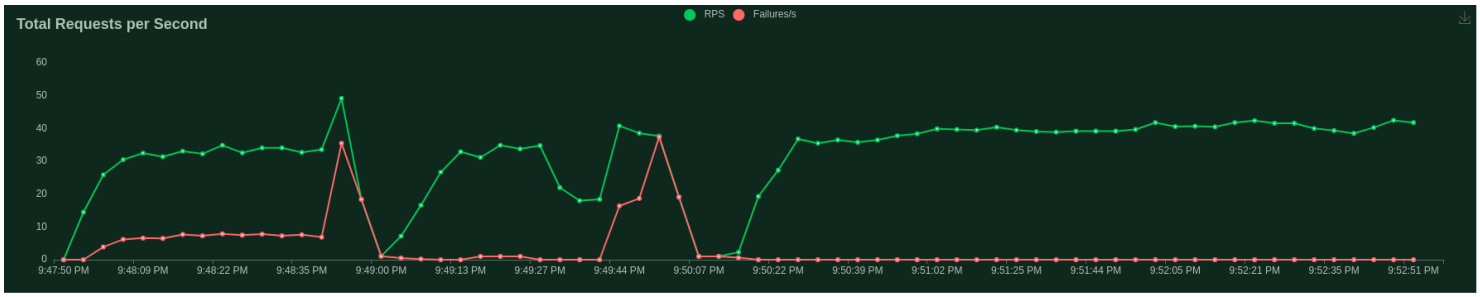
CPU Usage

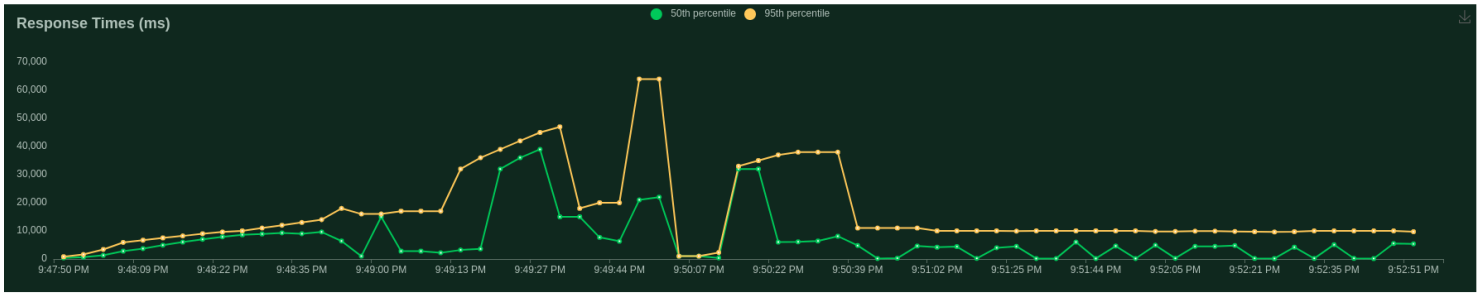


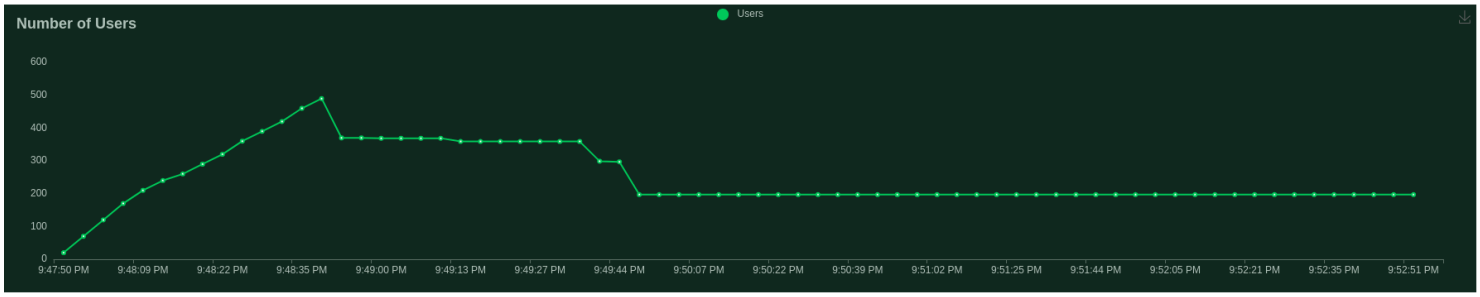
Memory Usage



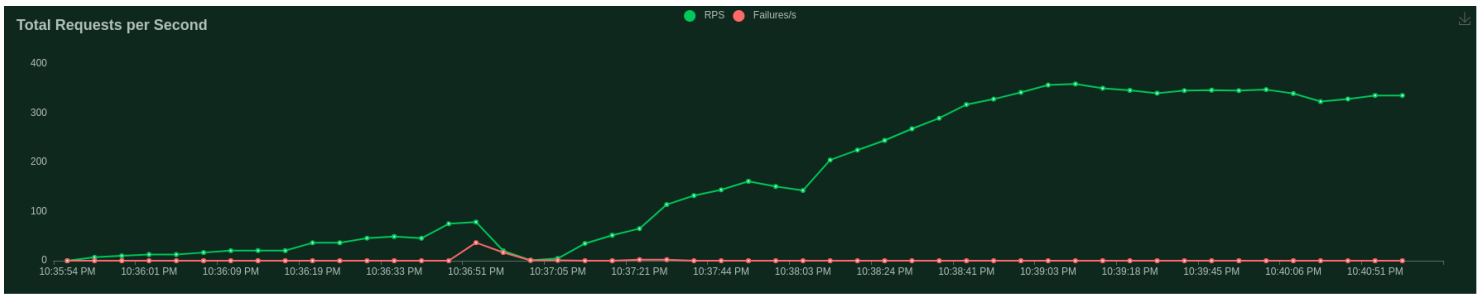
Locust Report: monolith

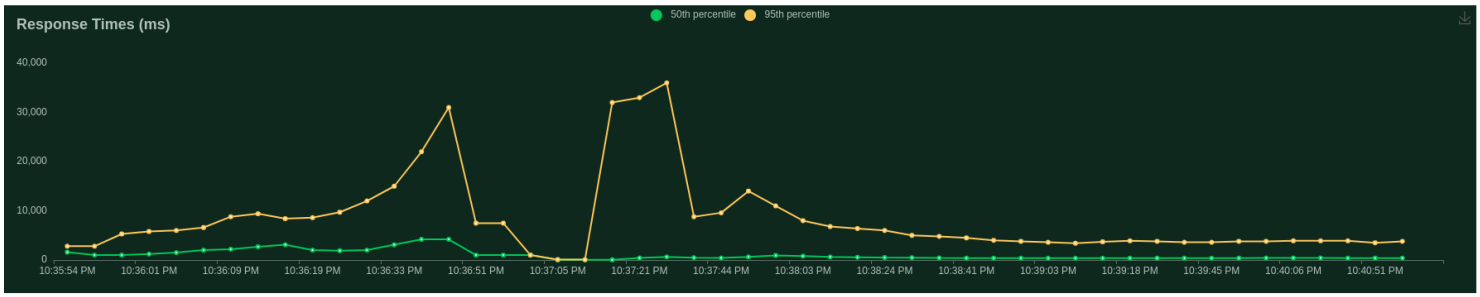


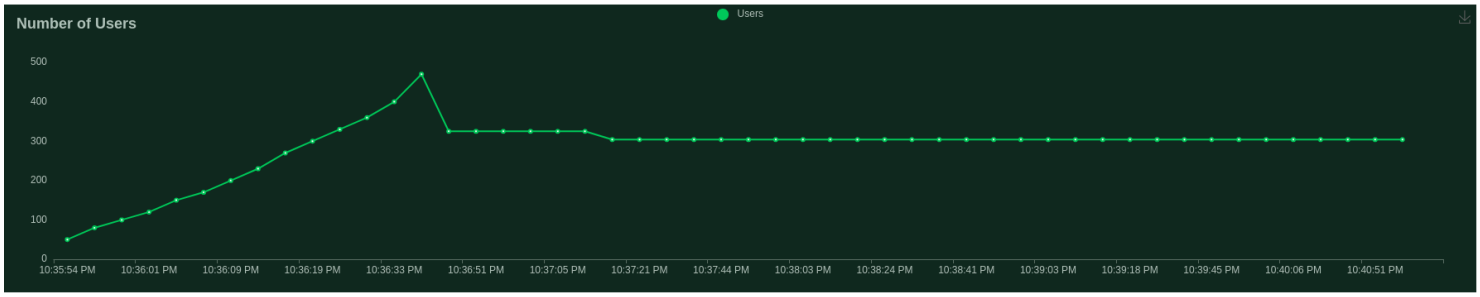




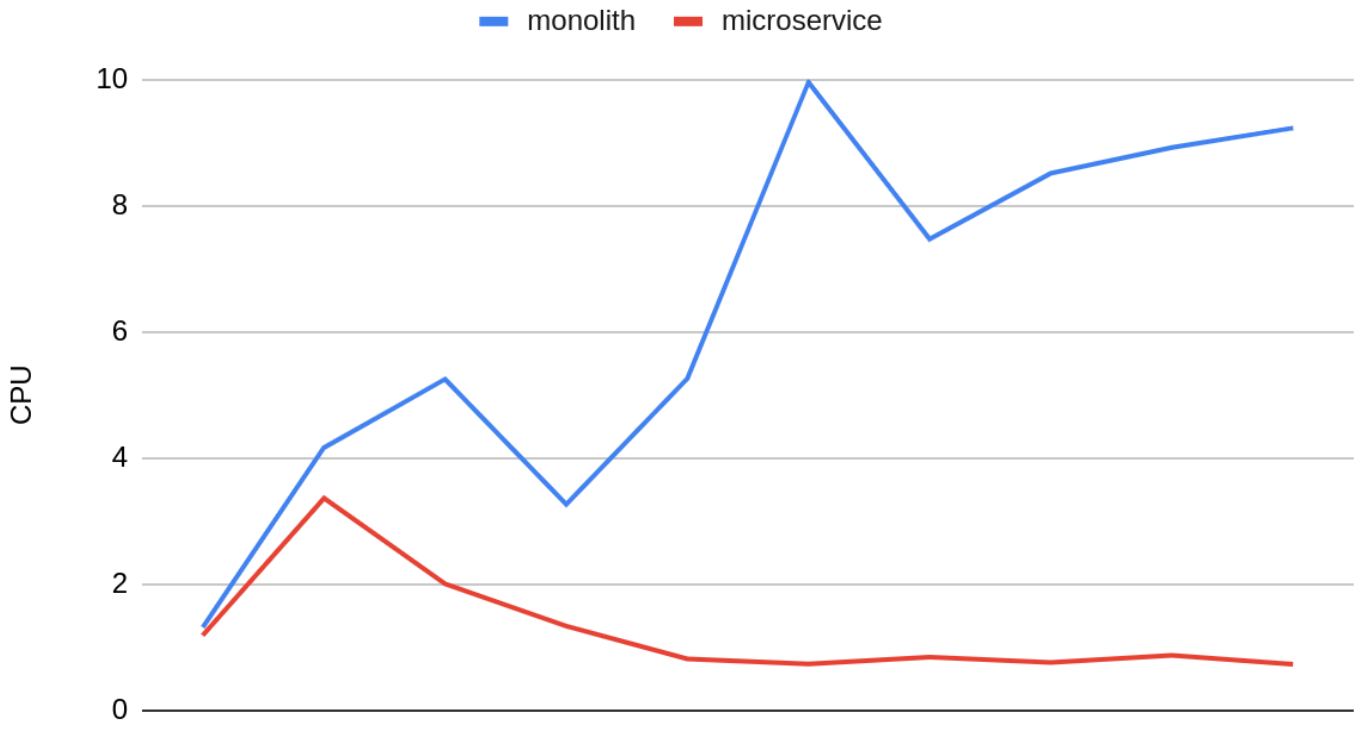
Locust Report: microservice



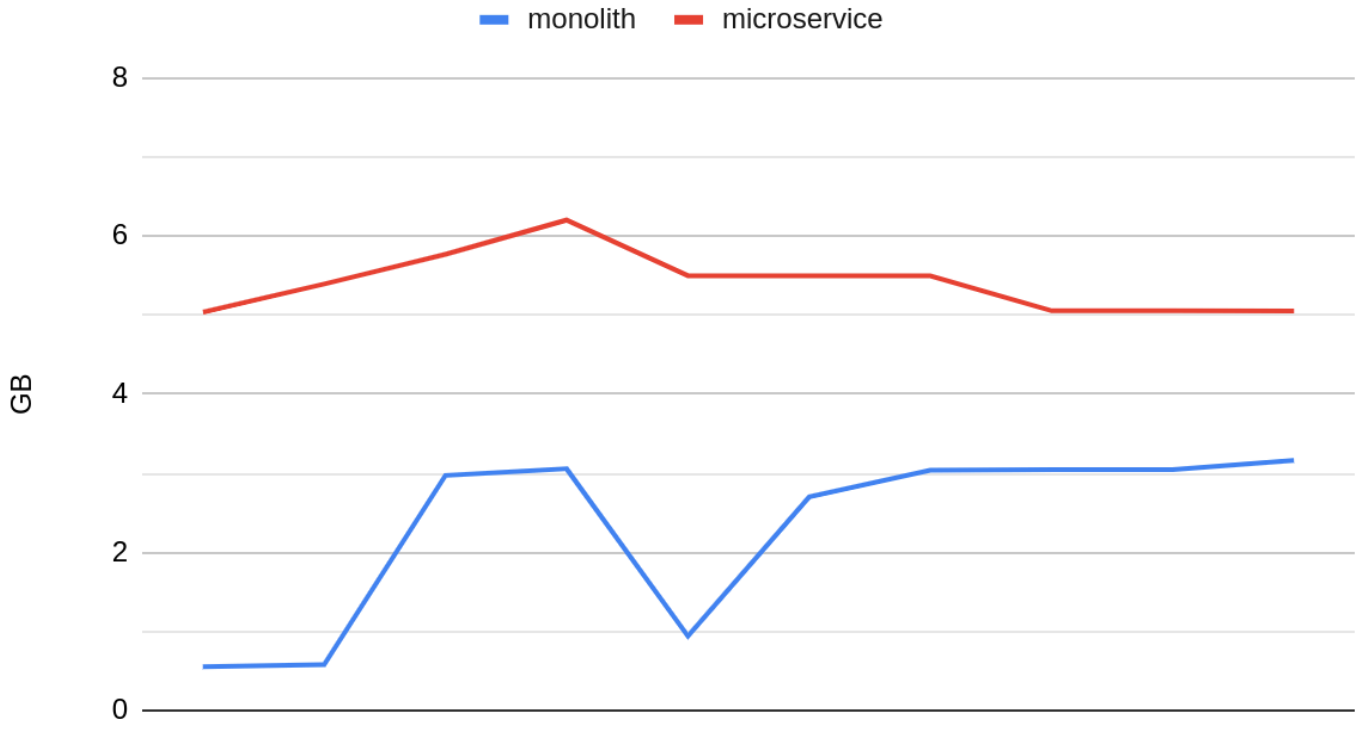




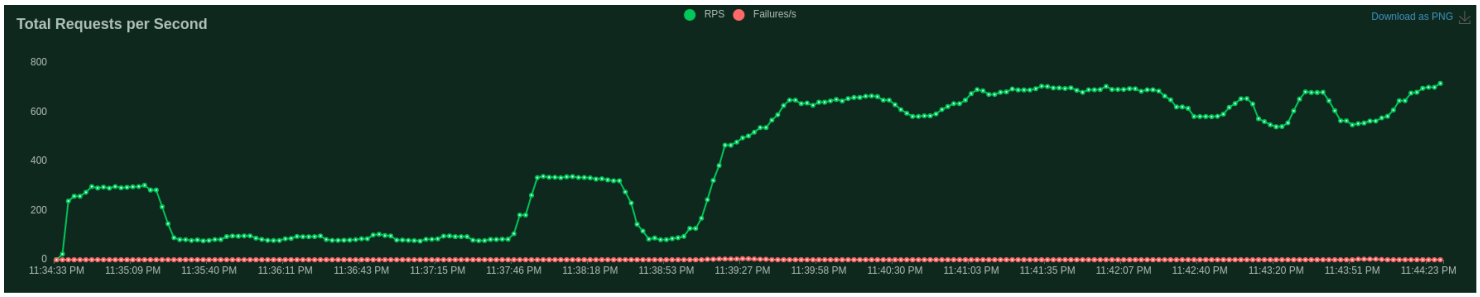
CPU Usage

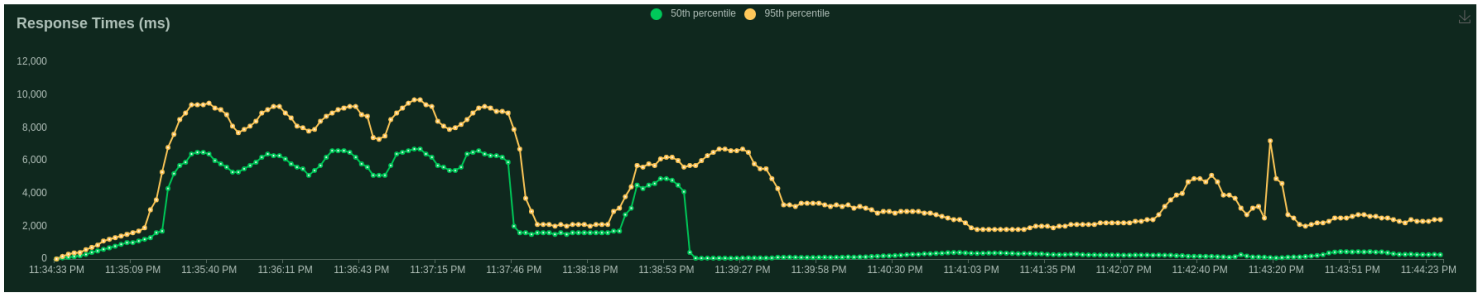


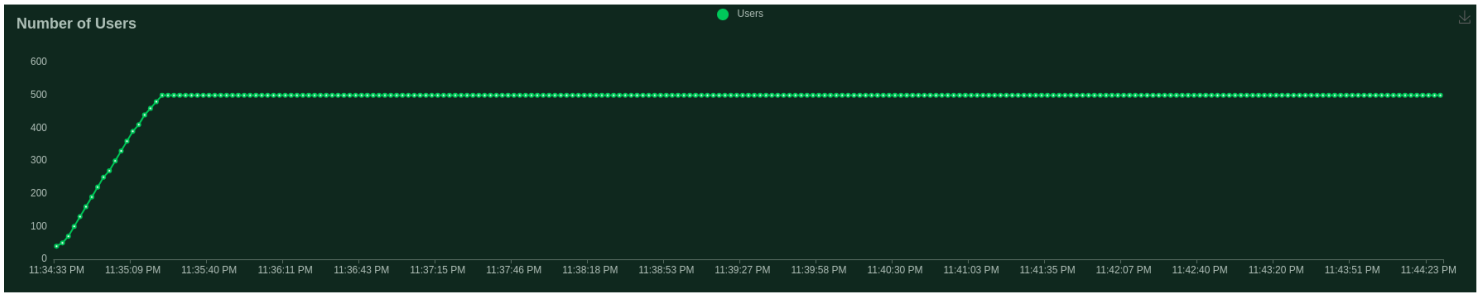
Memory Usage



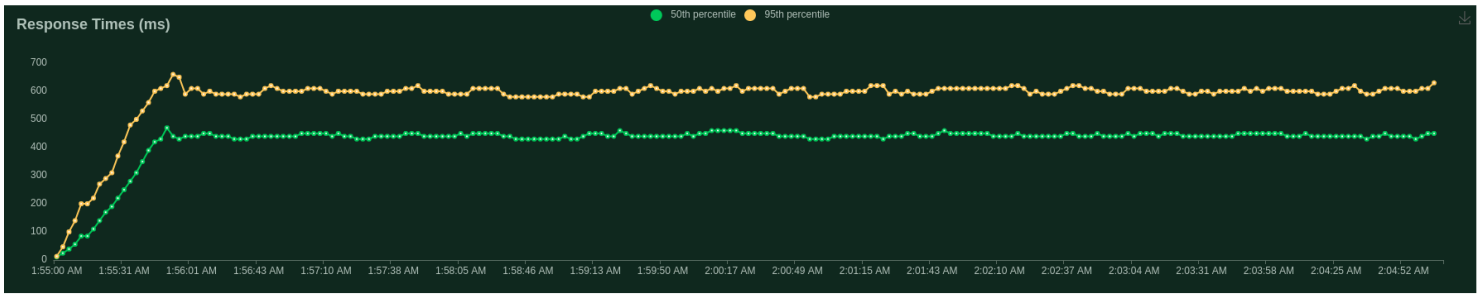
Locust Report: monolith

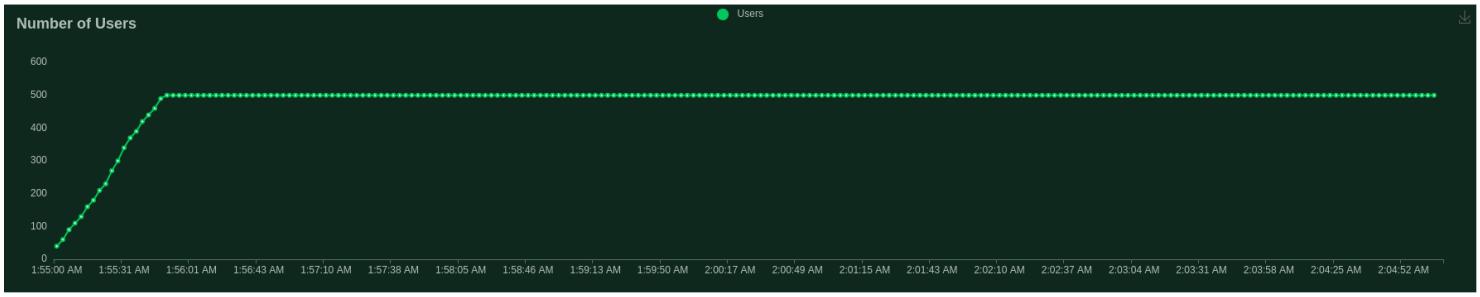




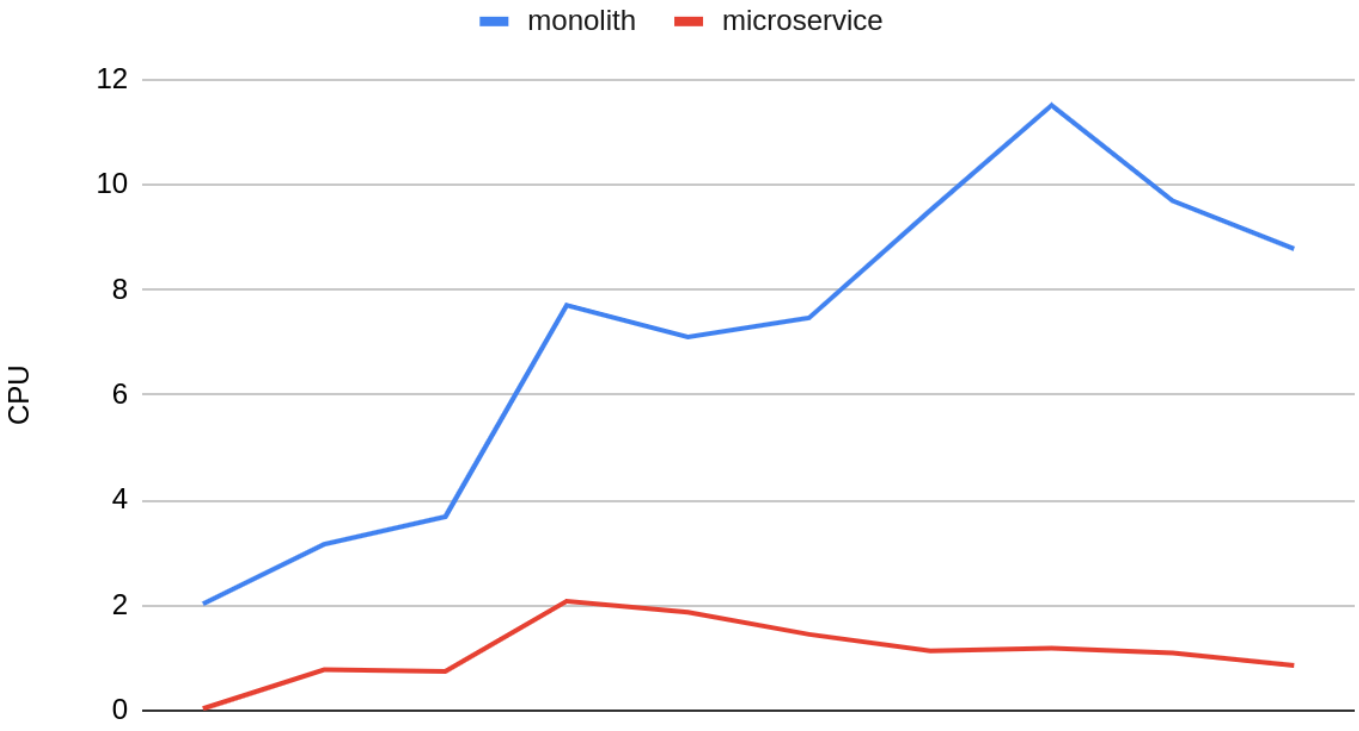


Locust Report: microservice

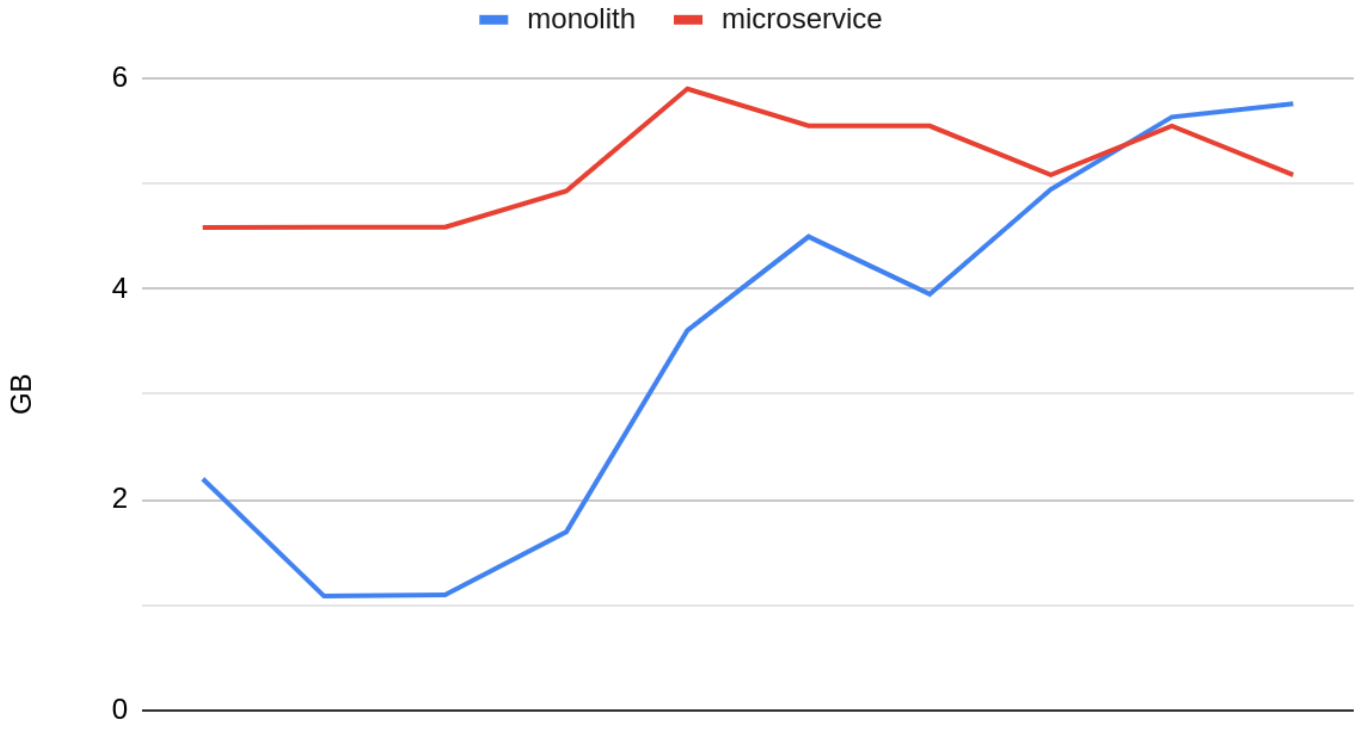




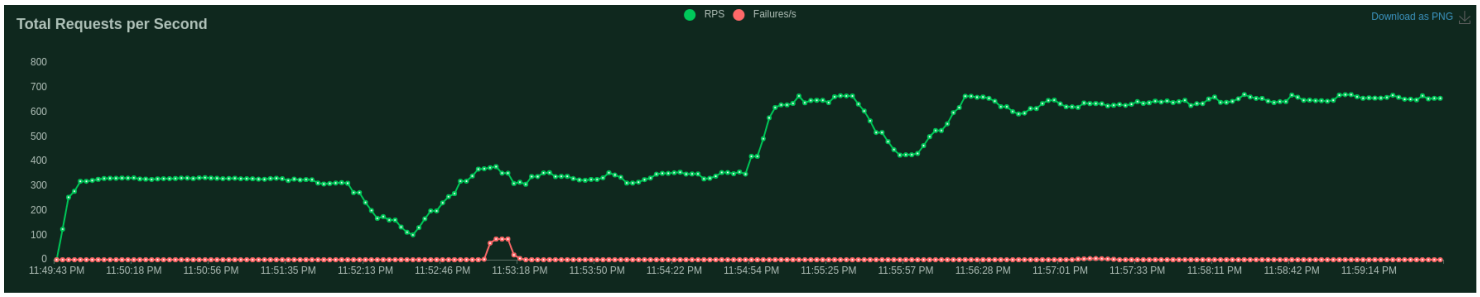
CPU Usage

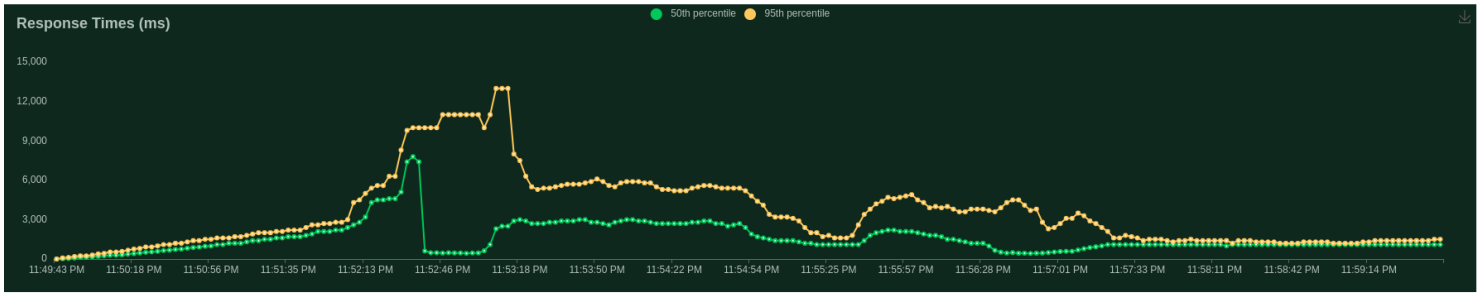


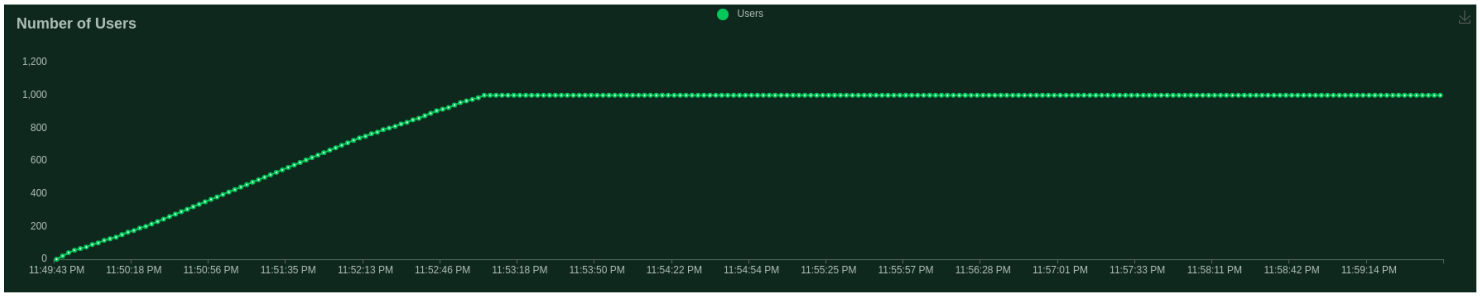
Memory Usage



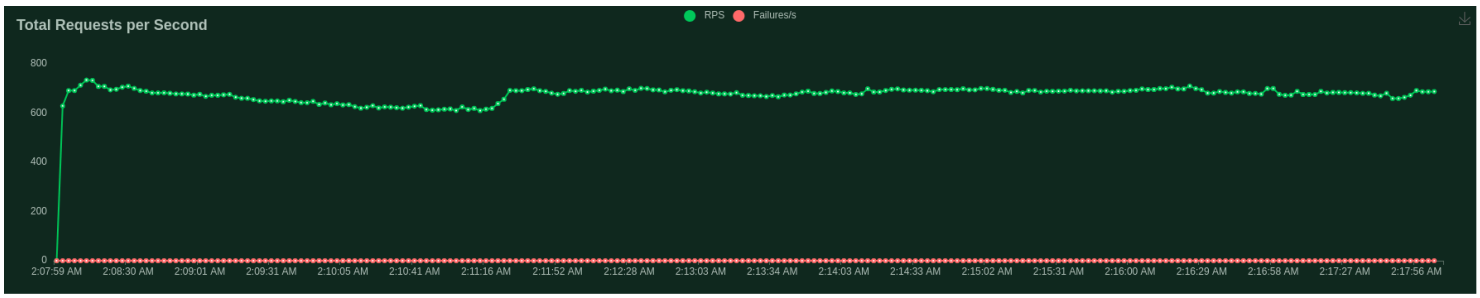
Locust Report: monolith

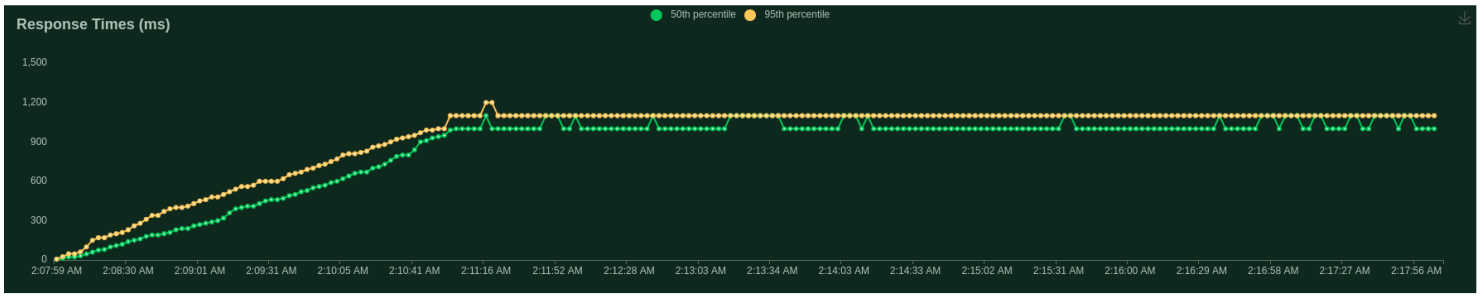


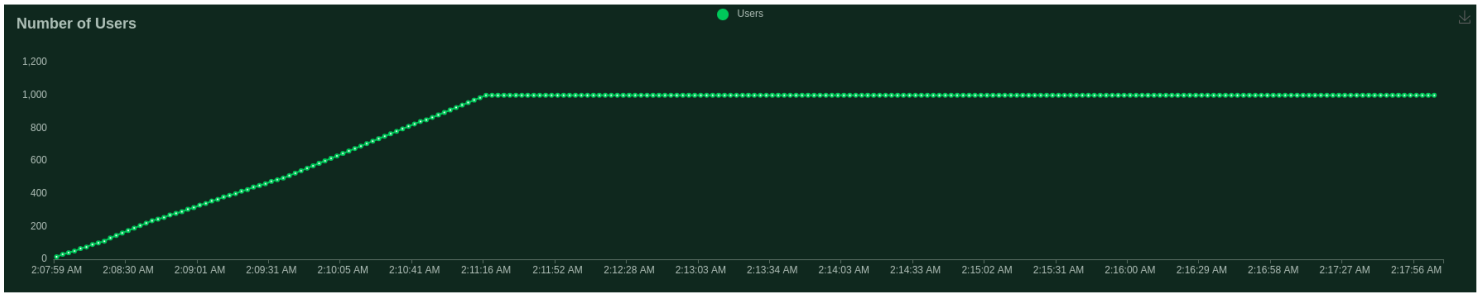




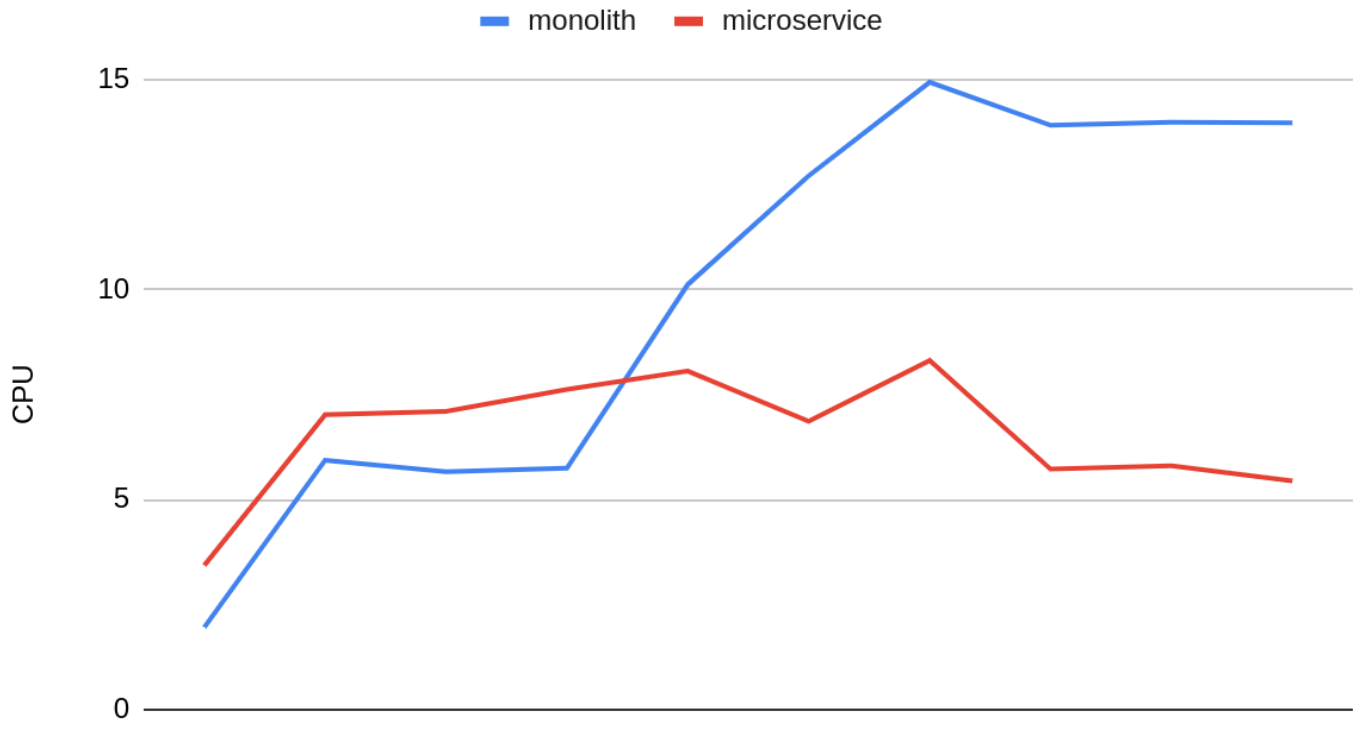
Locust Report: microservice



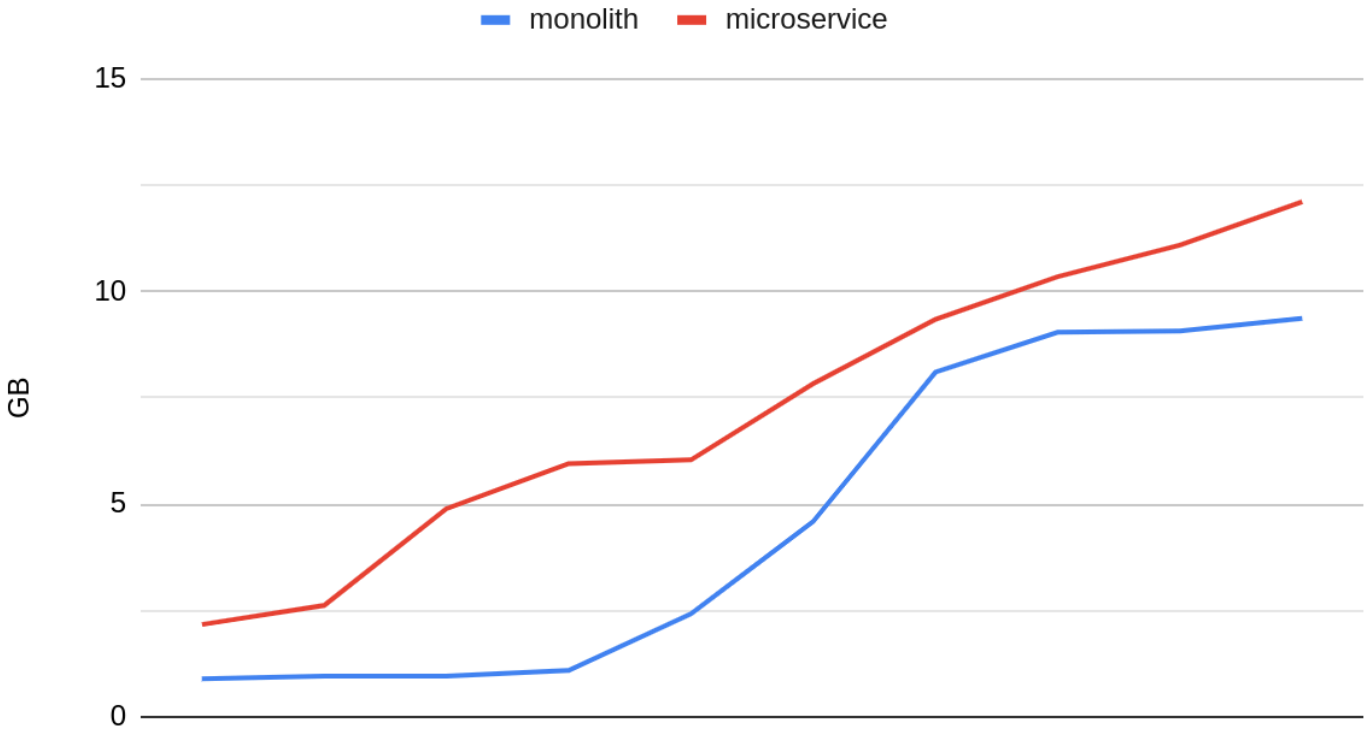




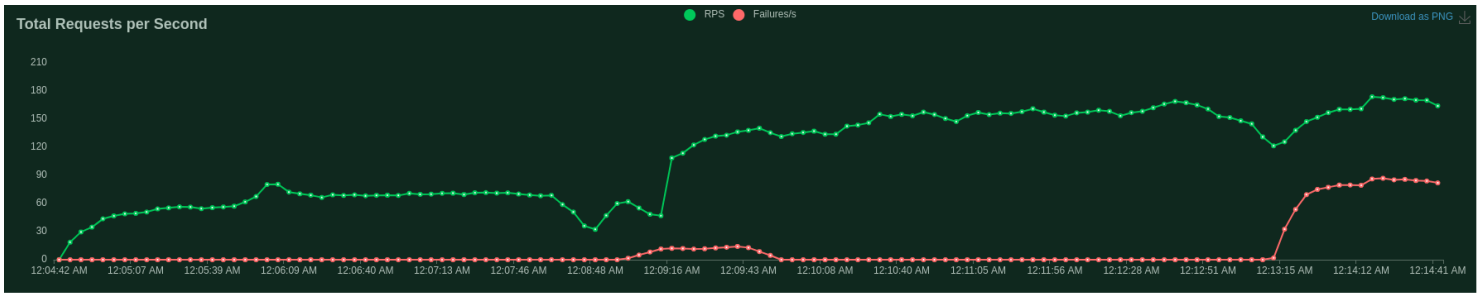
CPU Usage

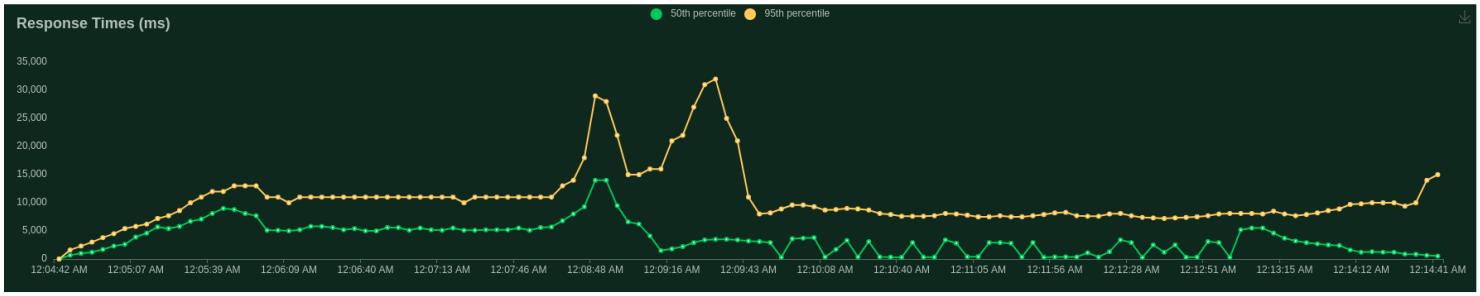


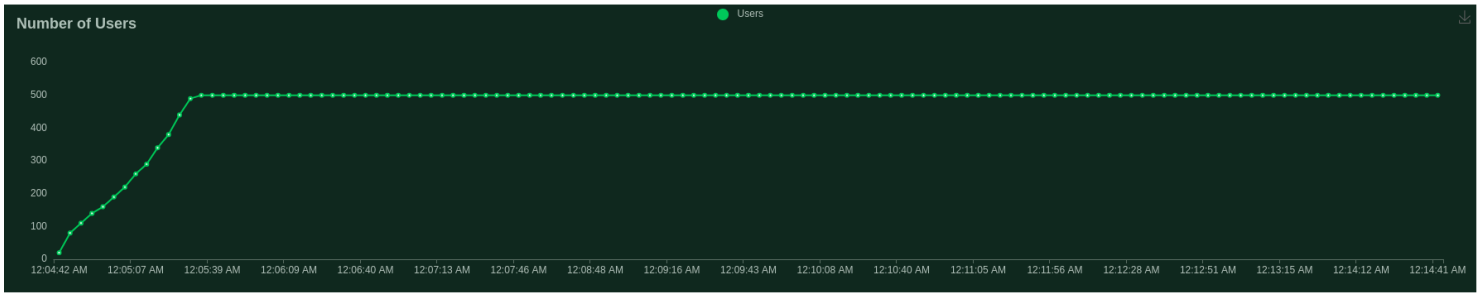
Memory Usage



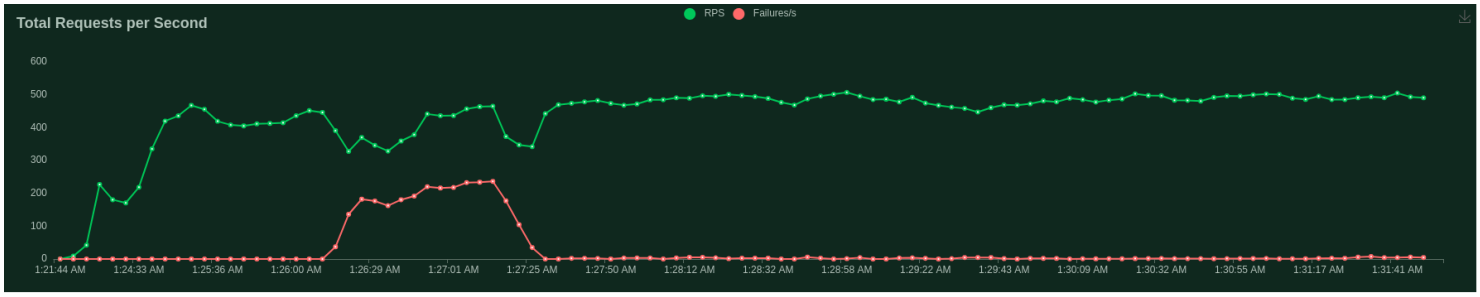
Locust Report: monolith

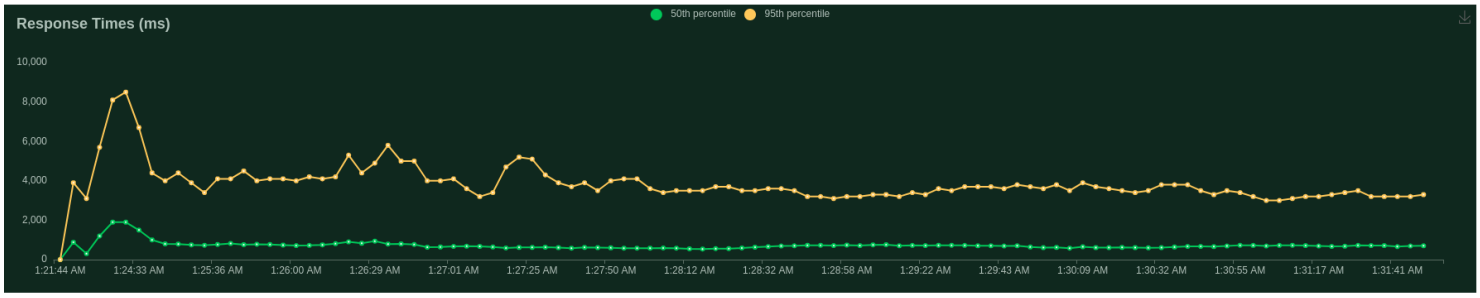


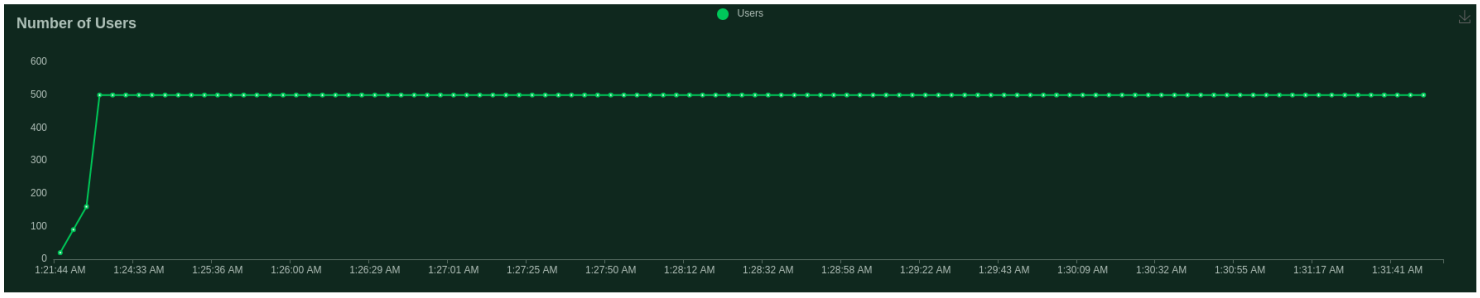




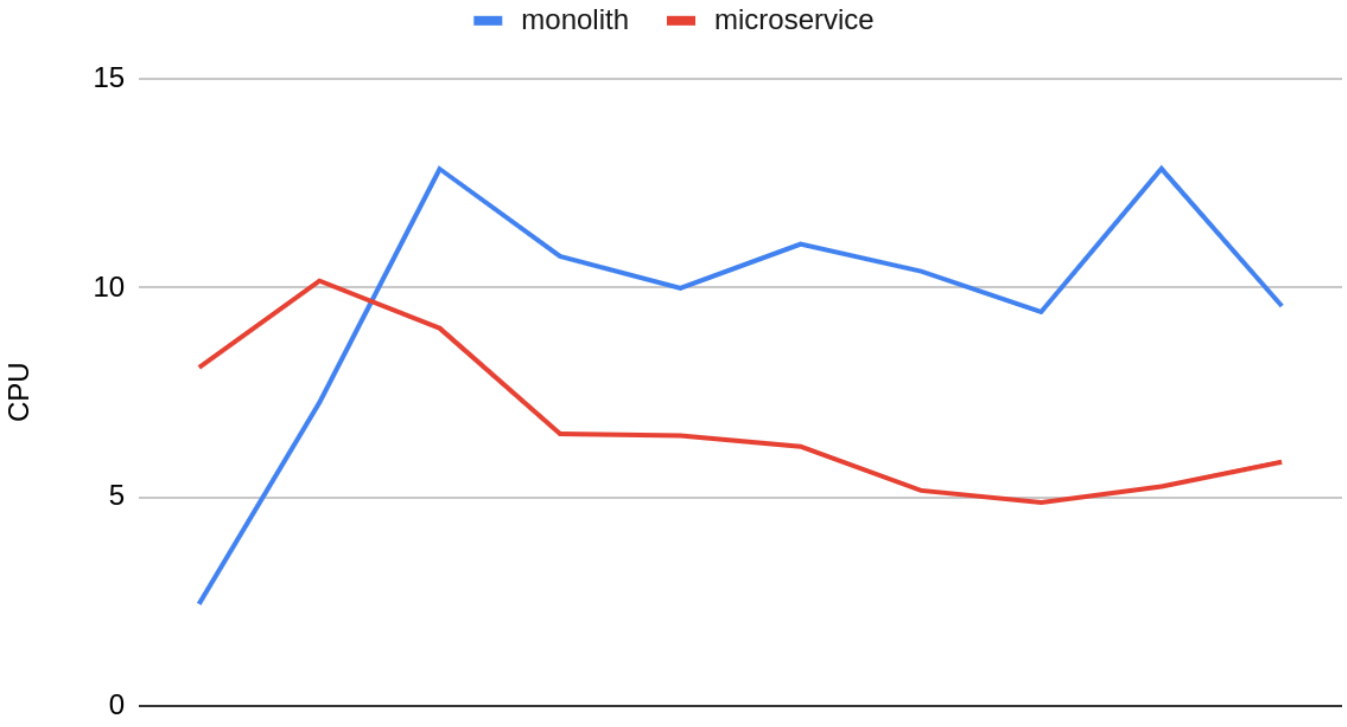
Locust Report: microservice



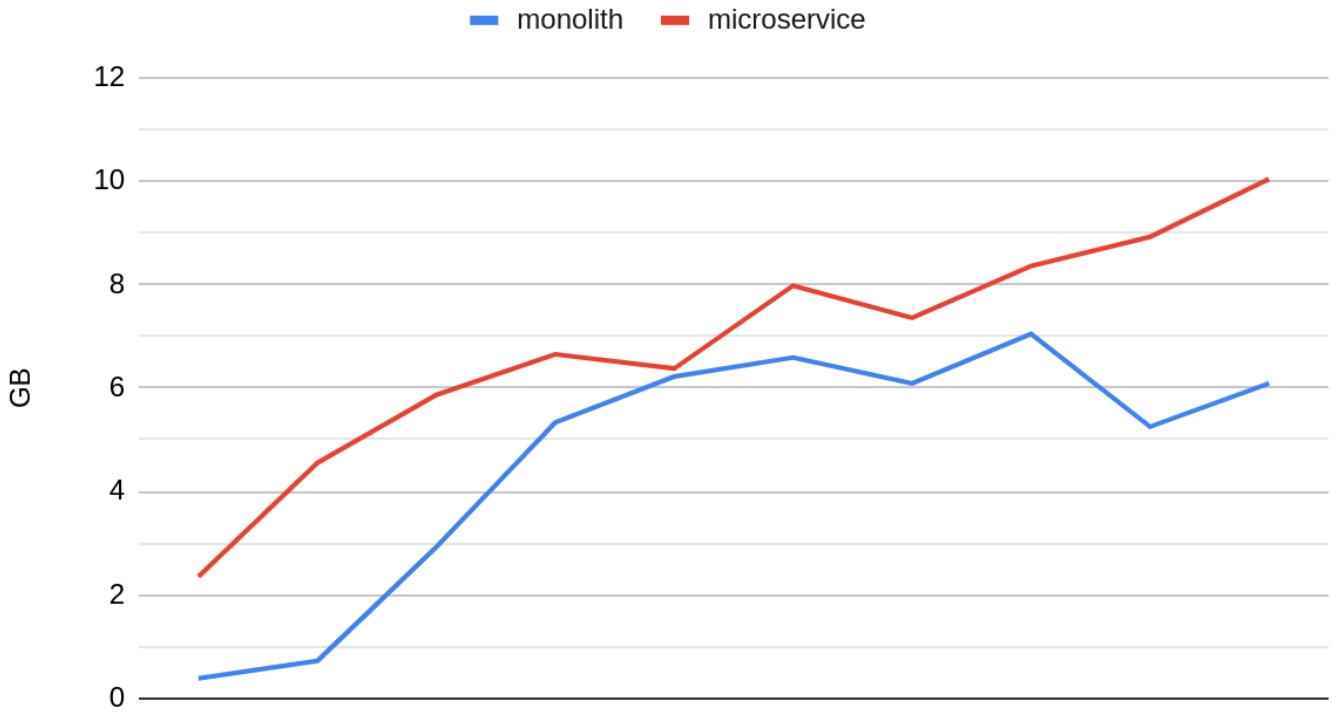




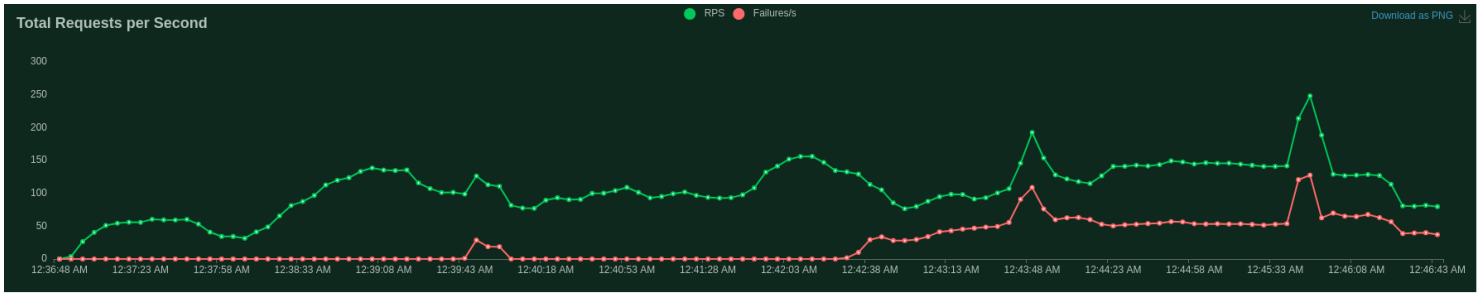
CPU Usage

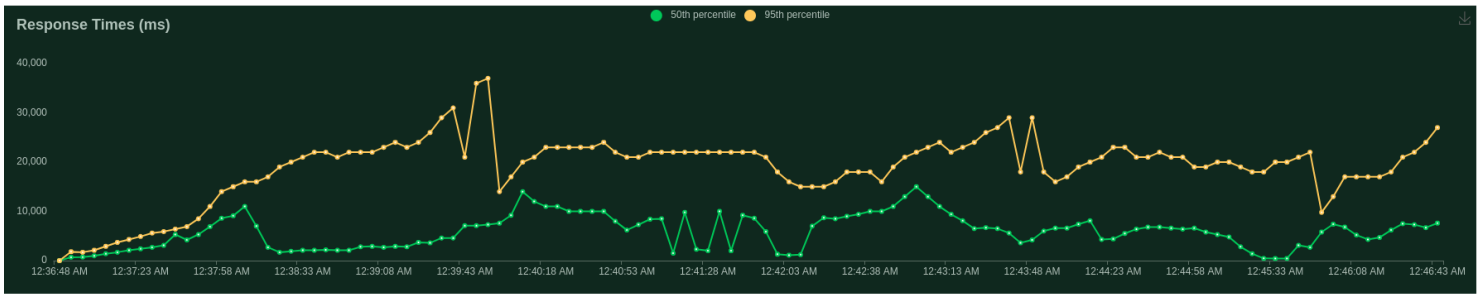


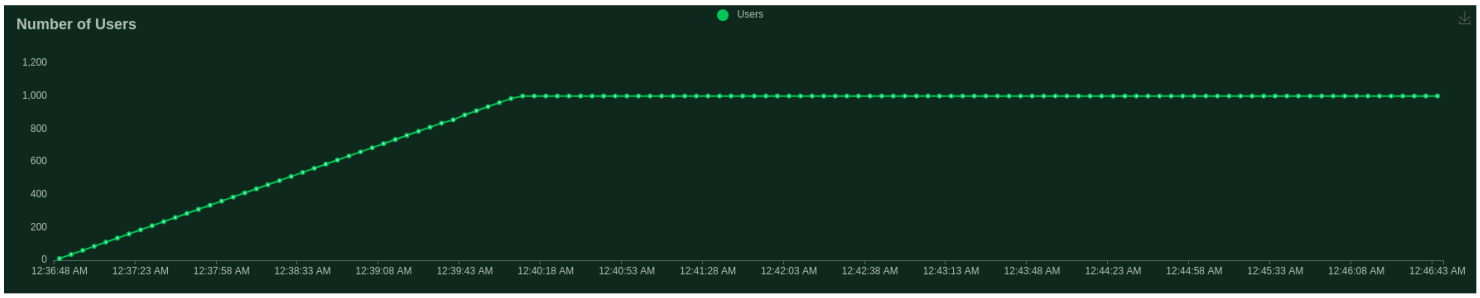
Memory Usage



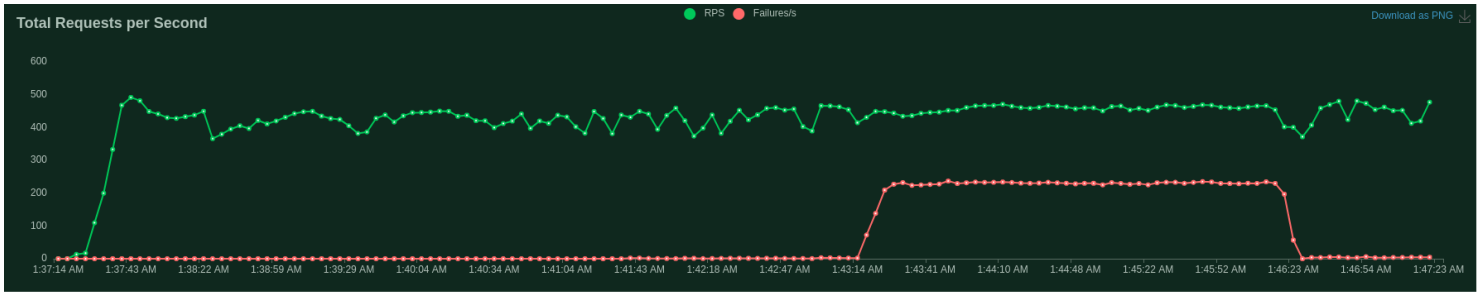
Locust Report: monolith

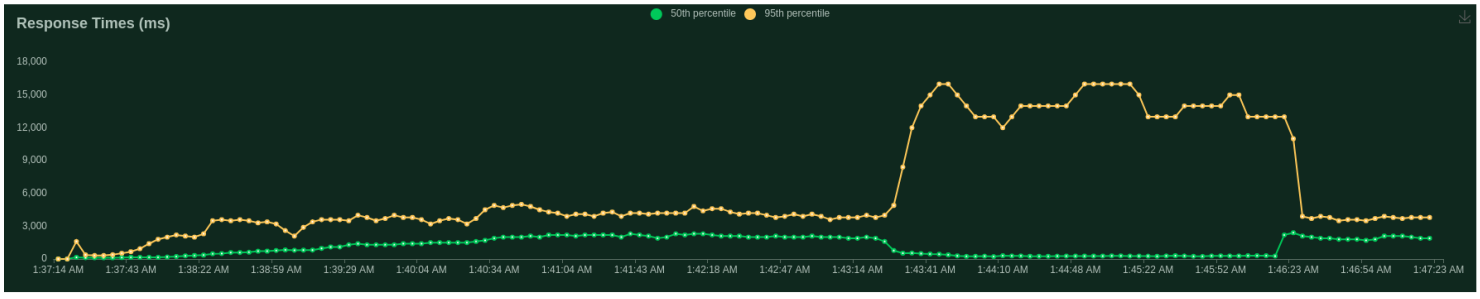


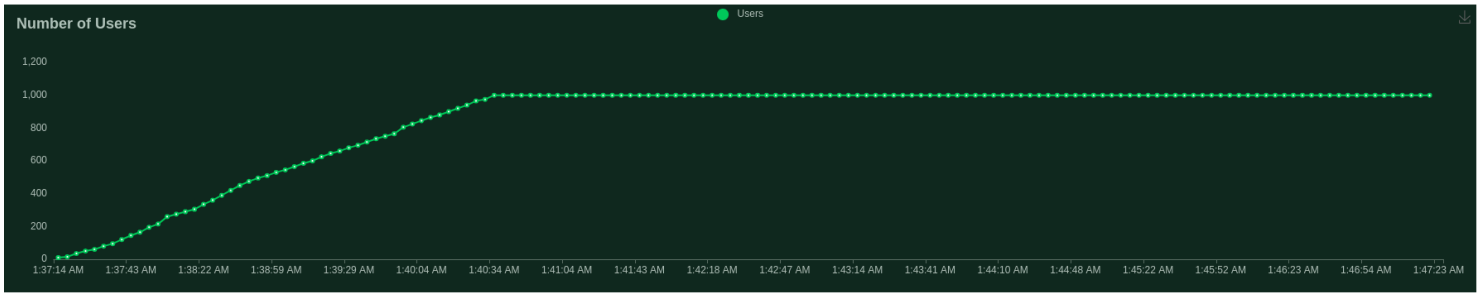




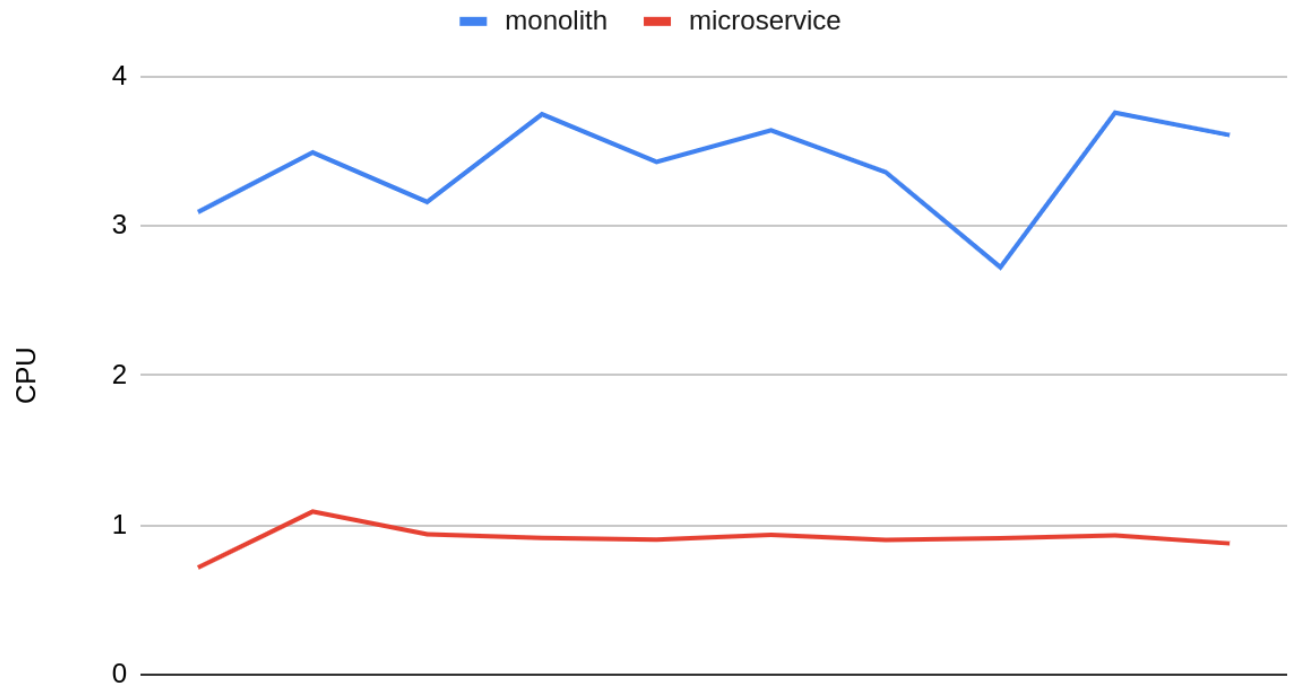
Locust Report: microservice



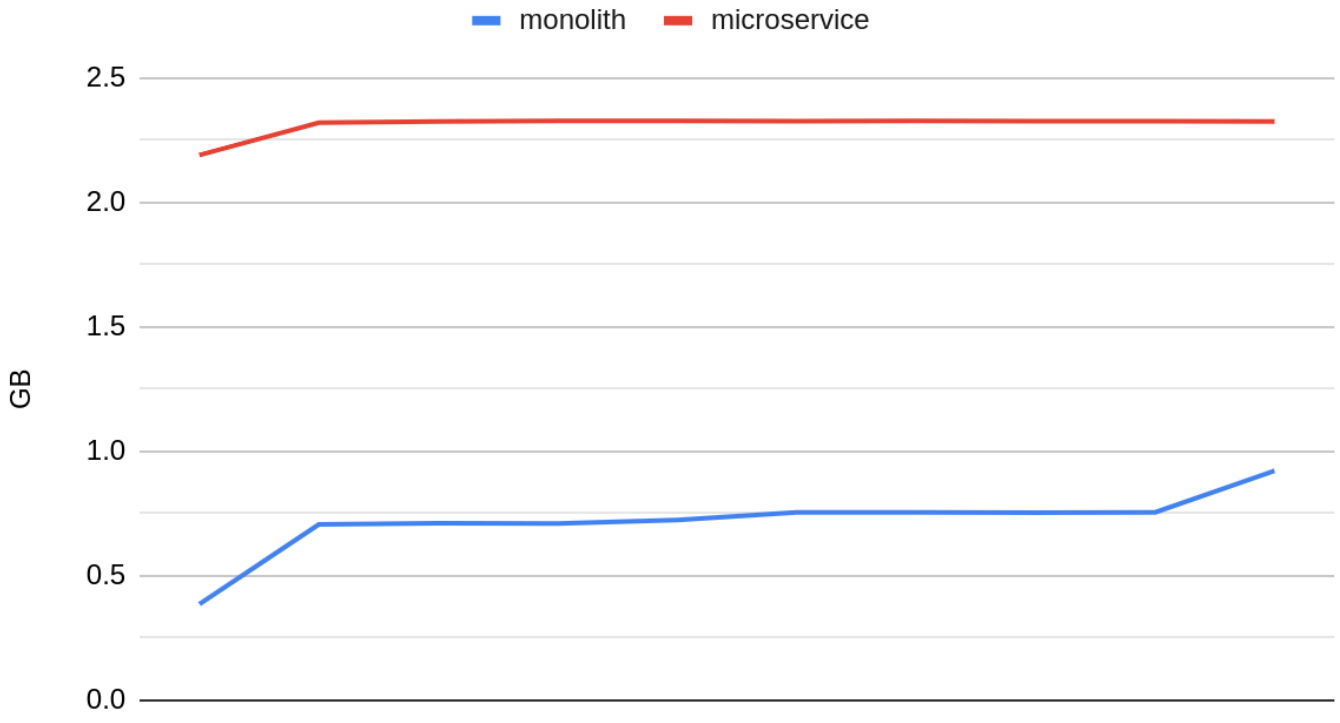




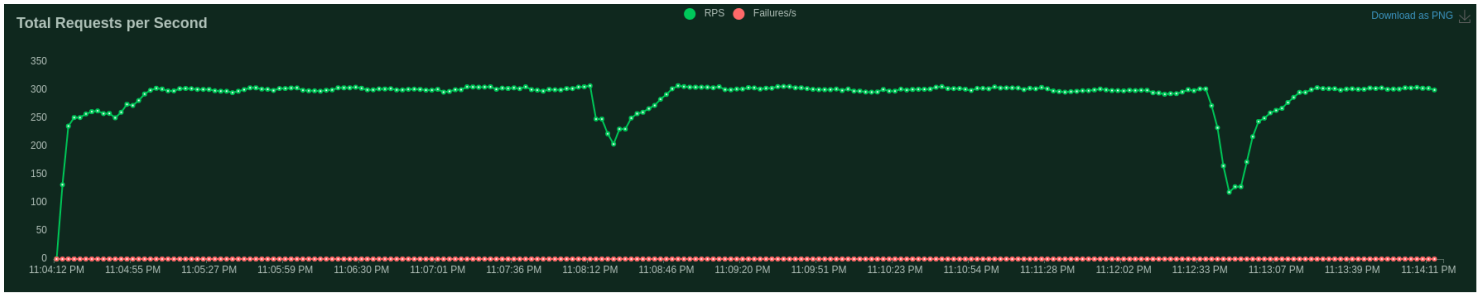
CPU Usage

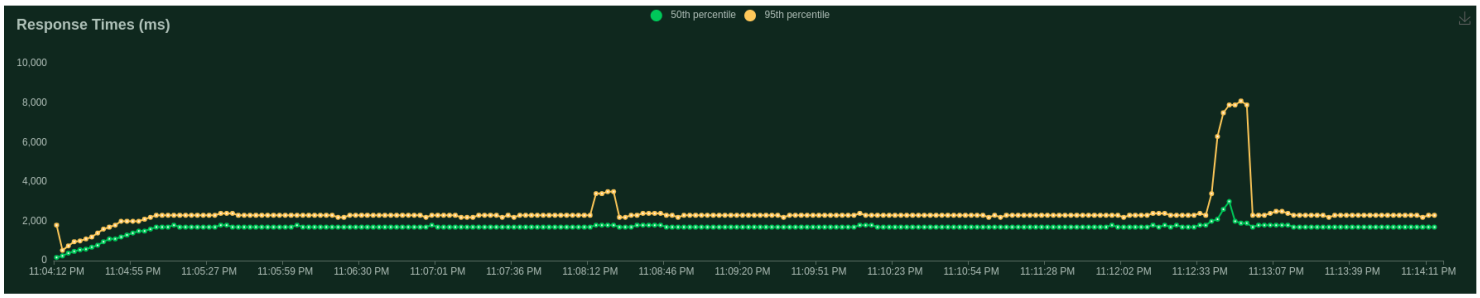


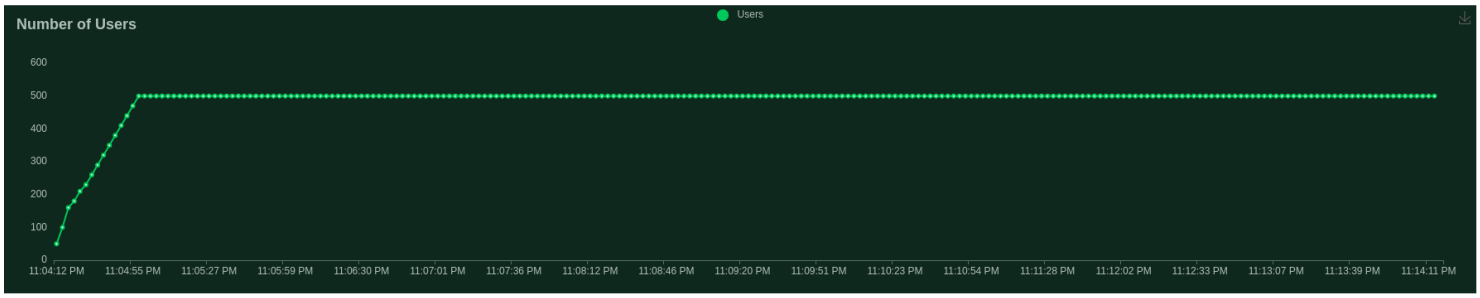
Memory Usage



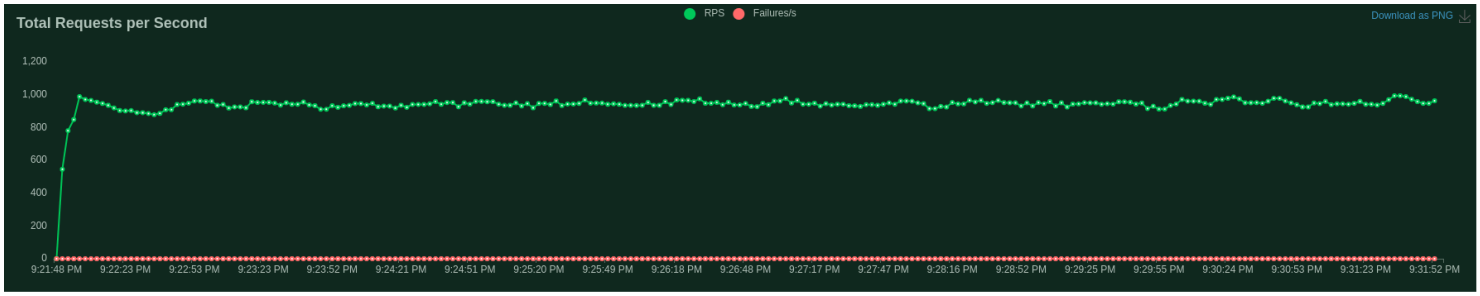
Locust Report: monolith

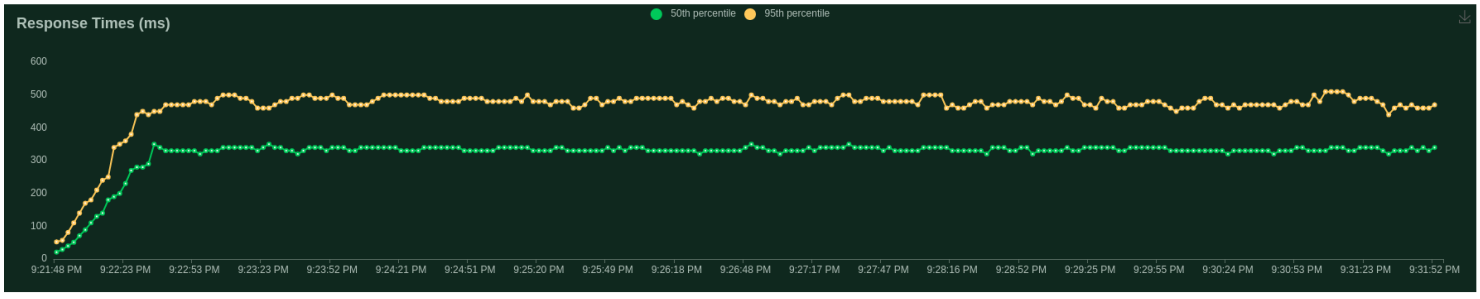


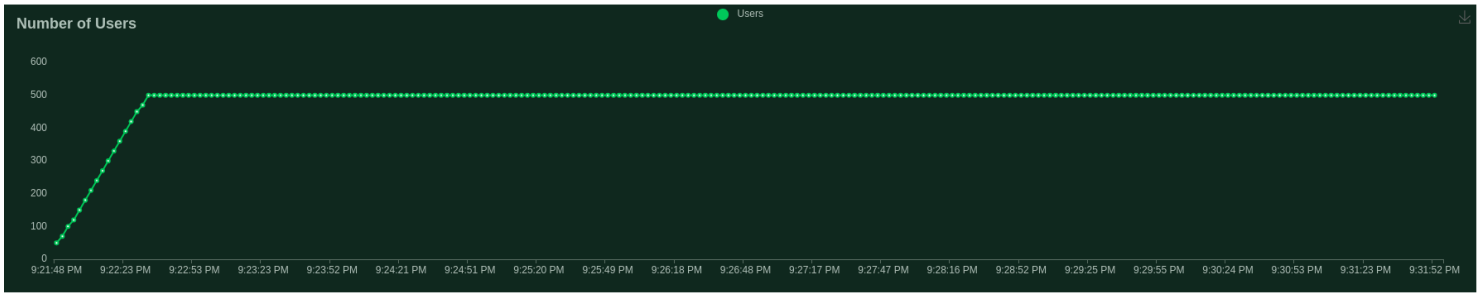




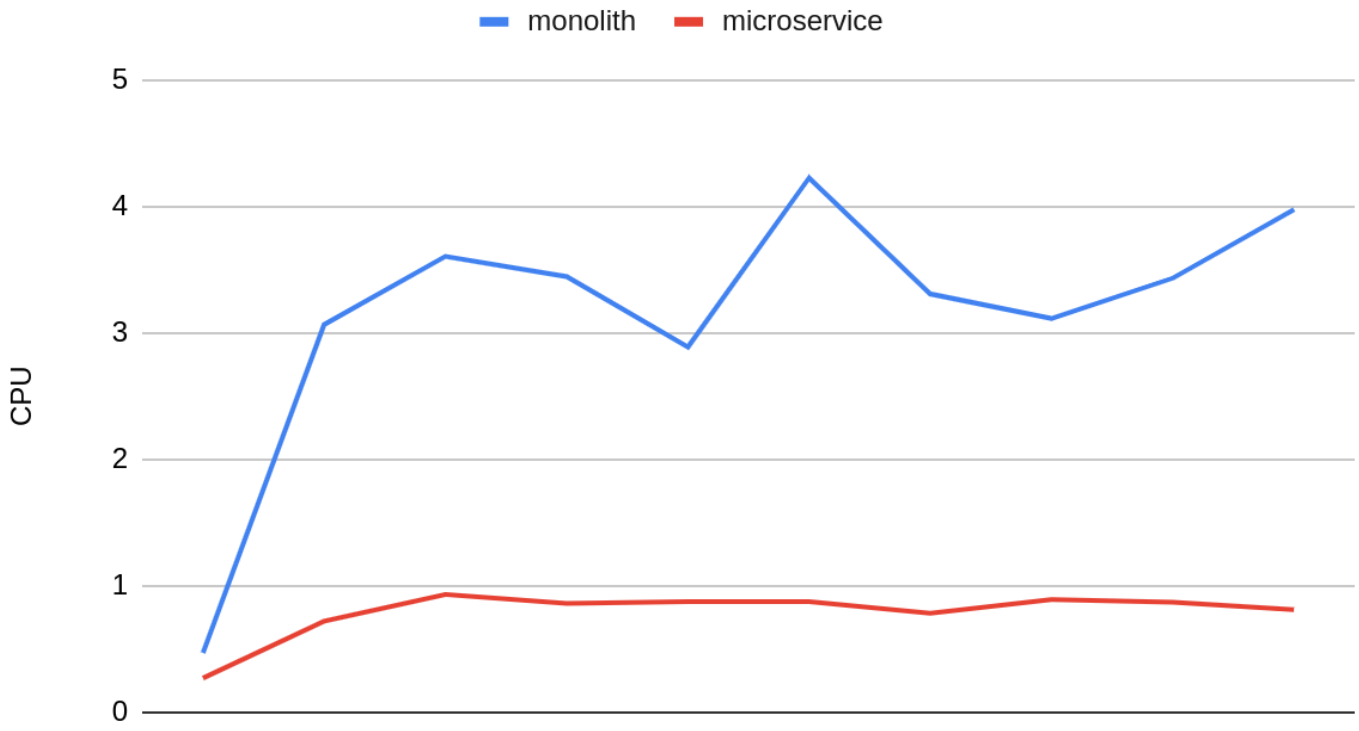
Locust Report: microservice



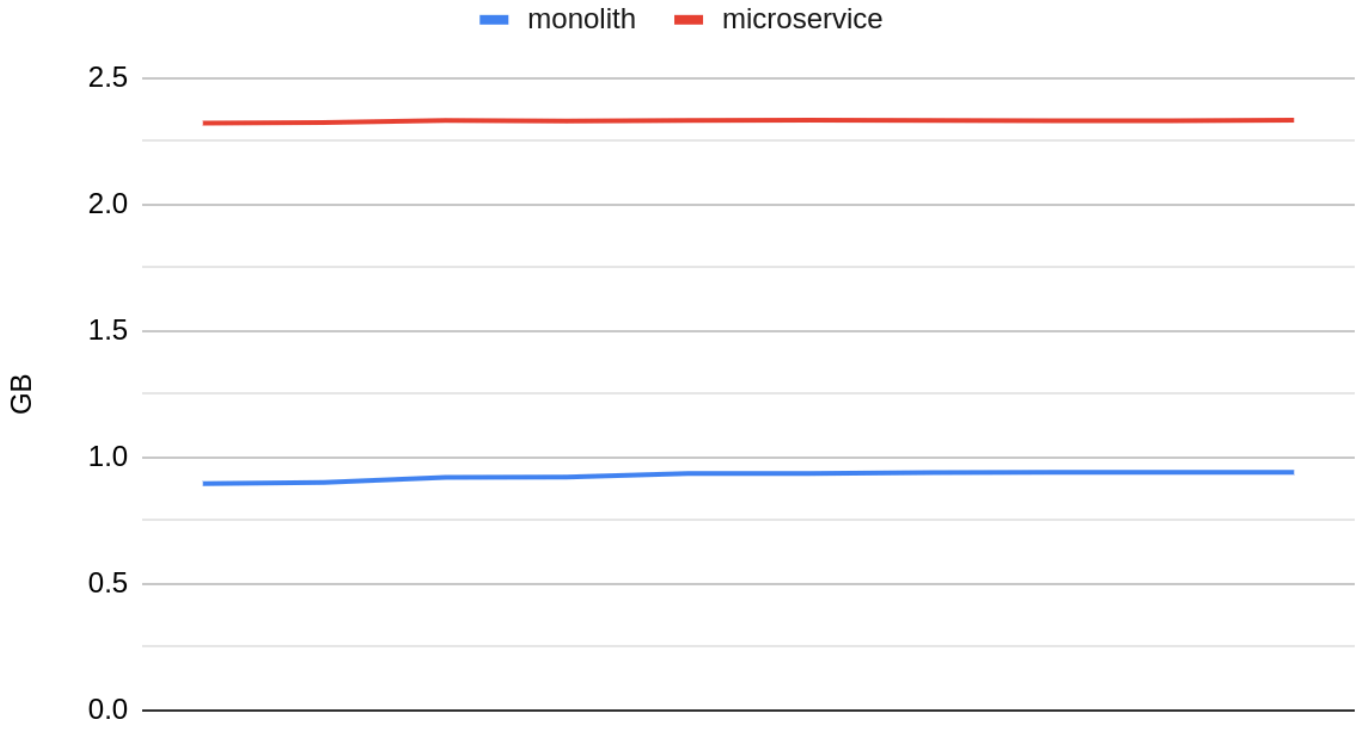




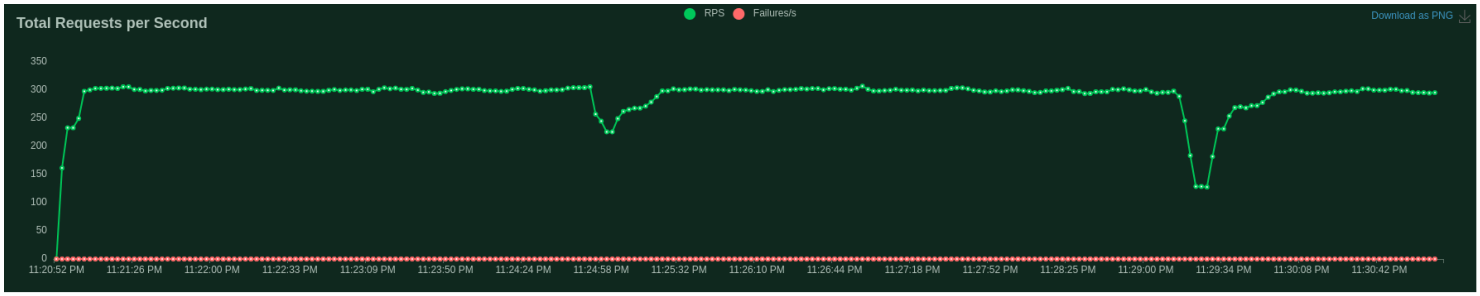
CPU Usage

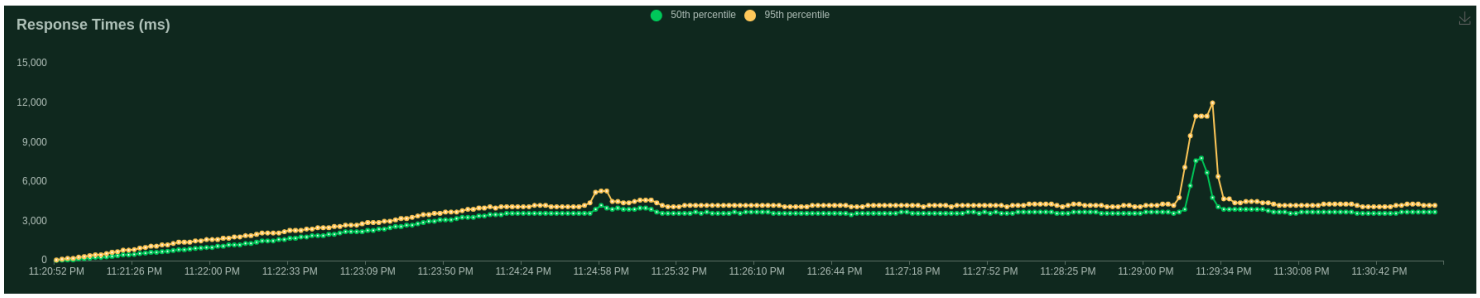


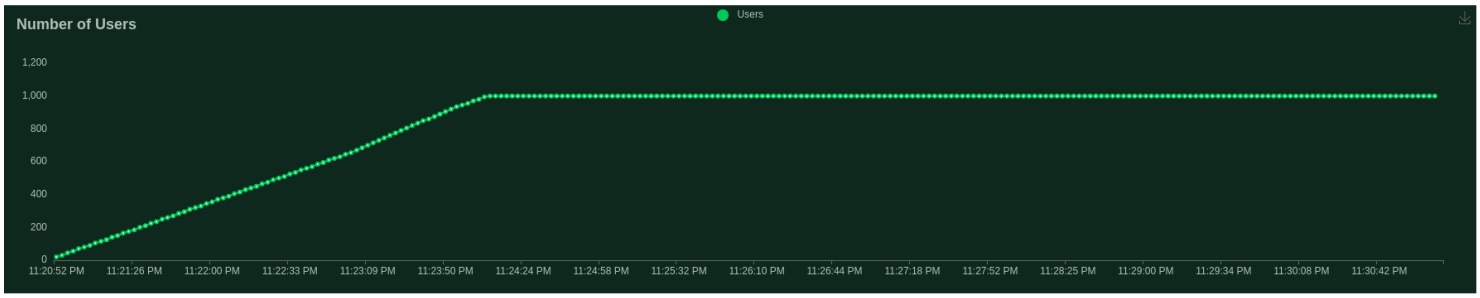
Memory Usage



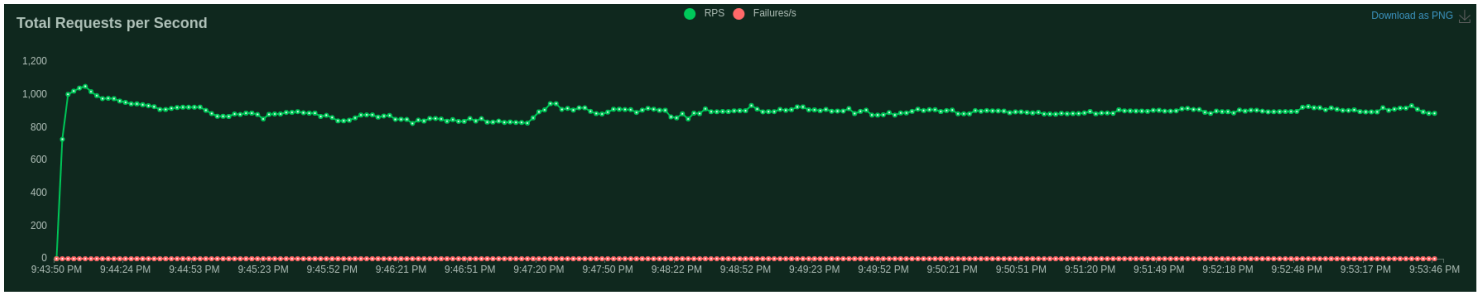
Locust Report: monolith

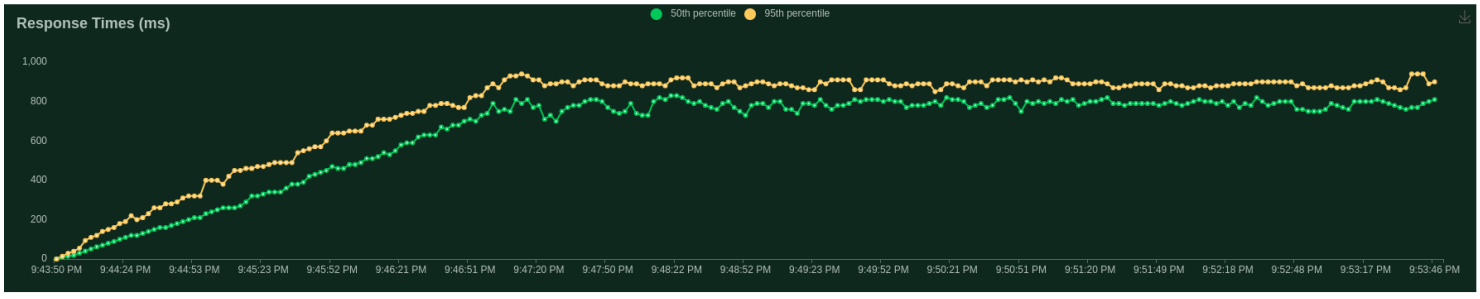


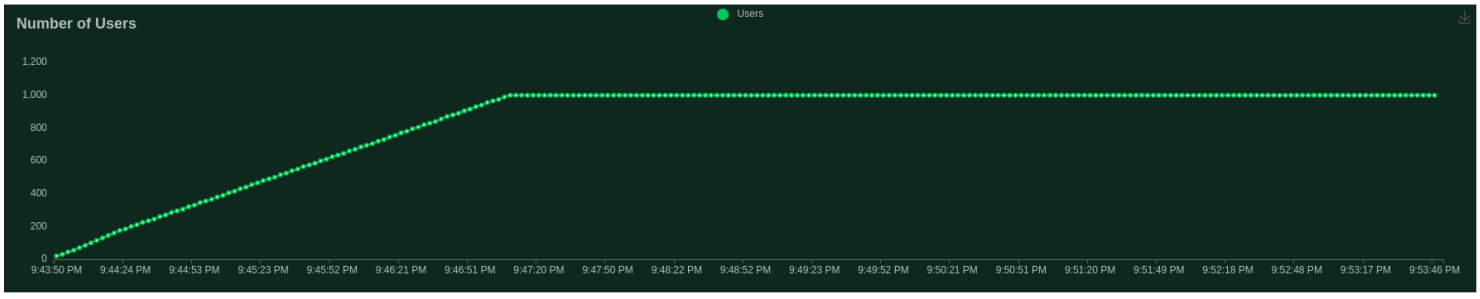




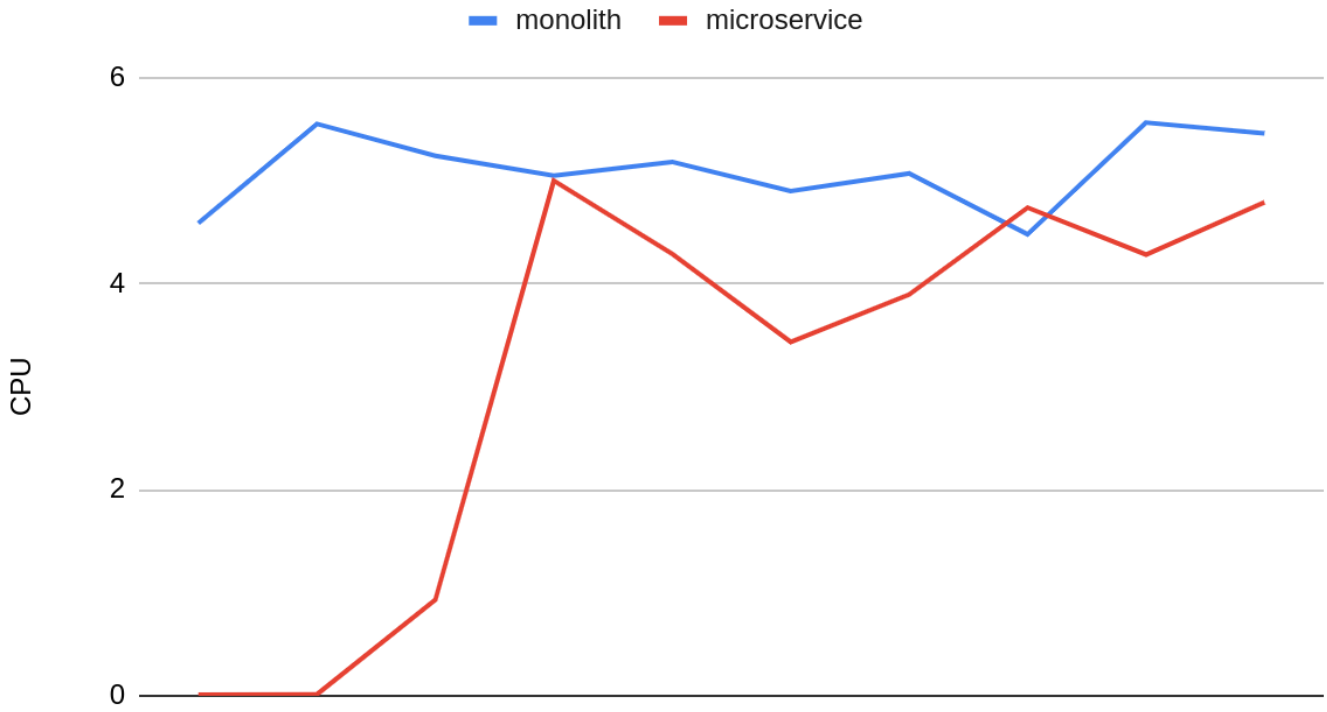
Locust Report: microservice



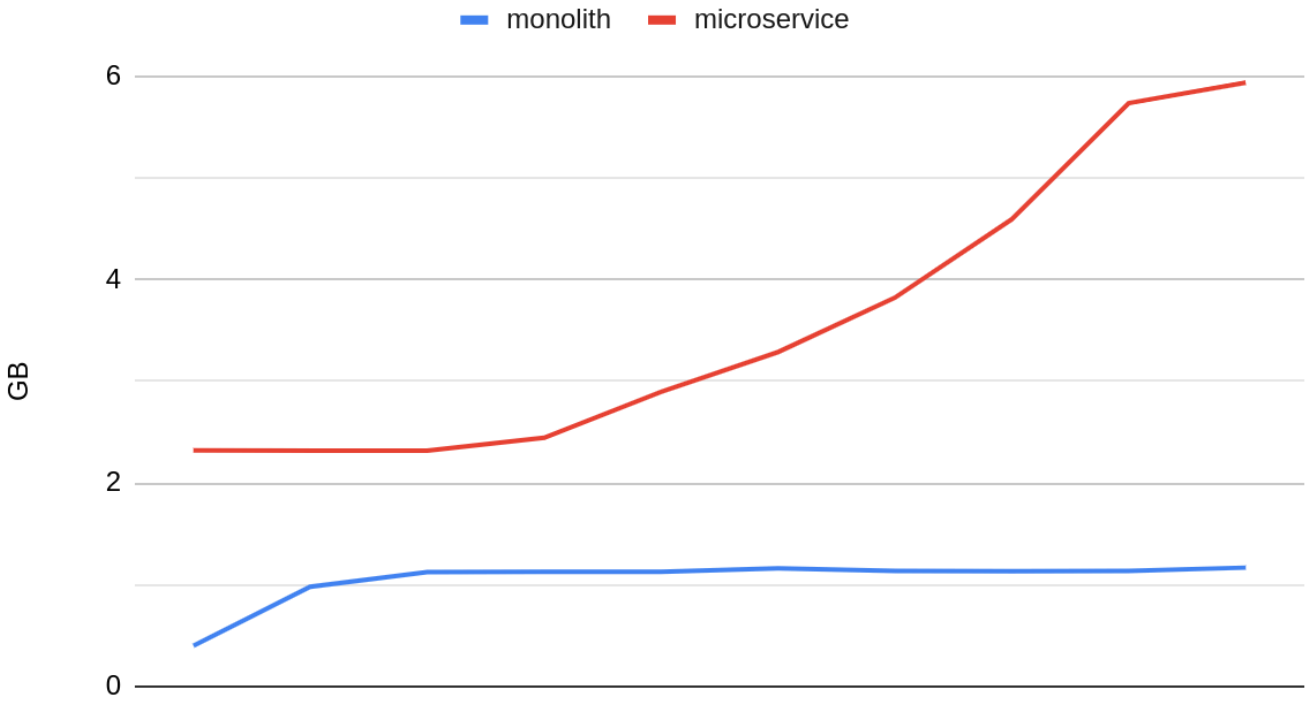




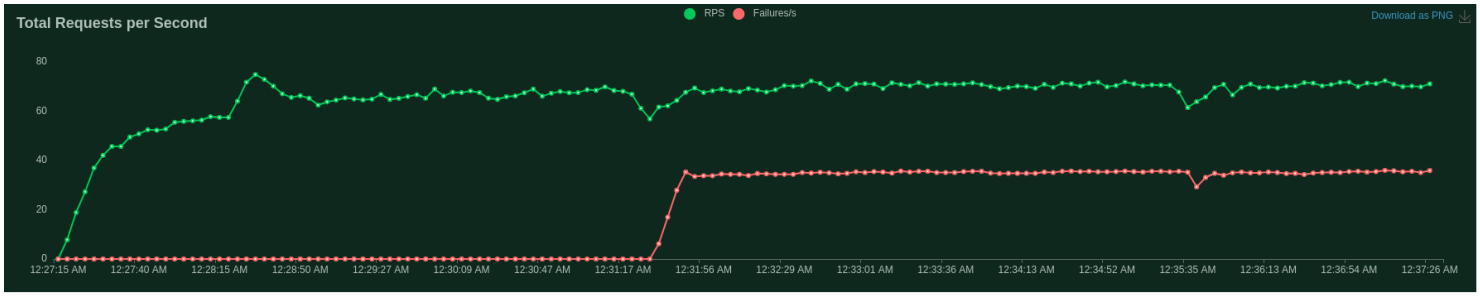
CPU Usage

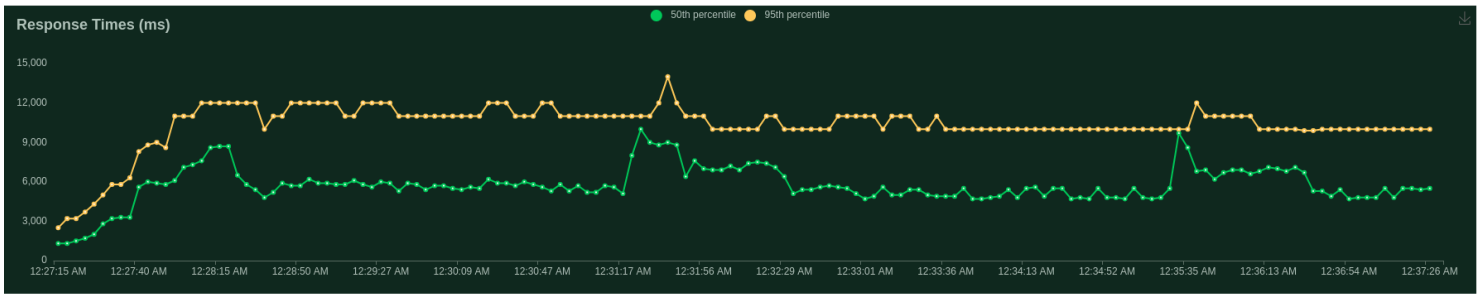


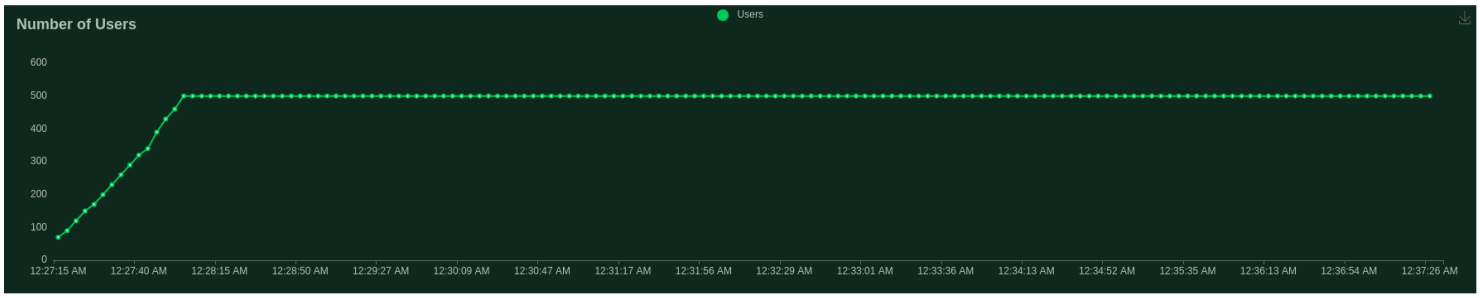
Memory Usage



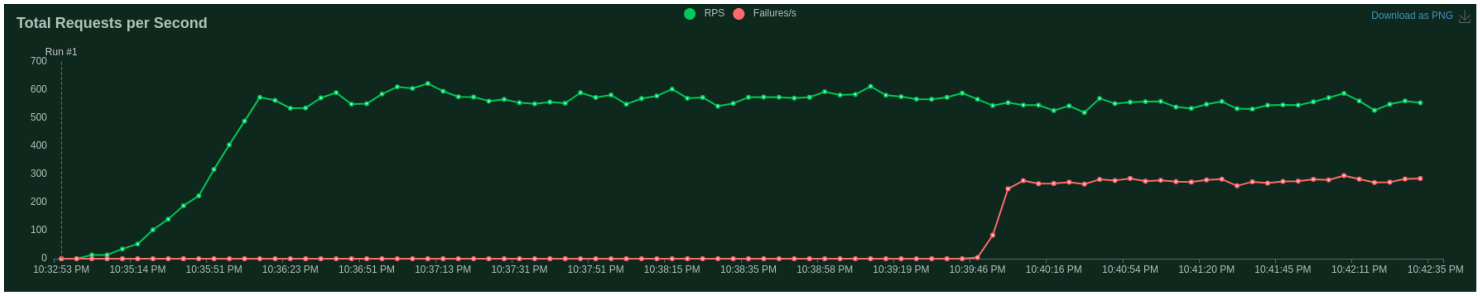
Locust Report: monolith

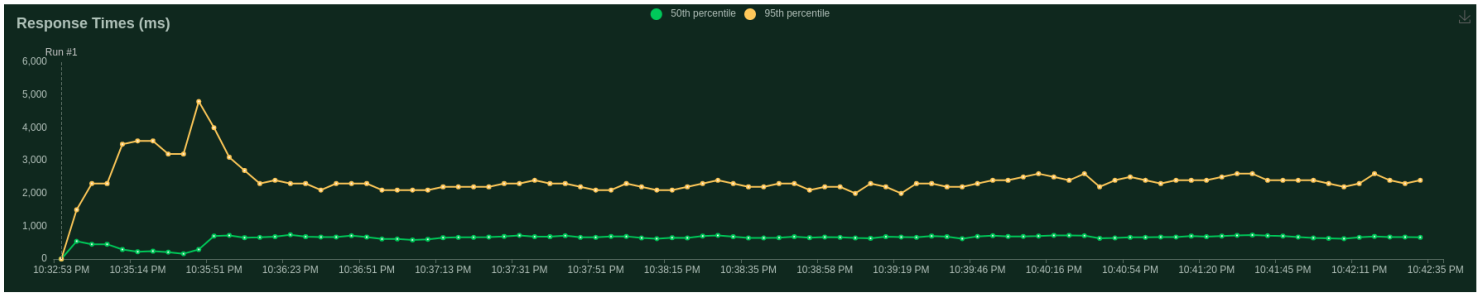


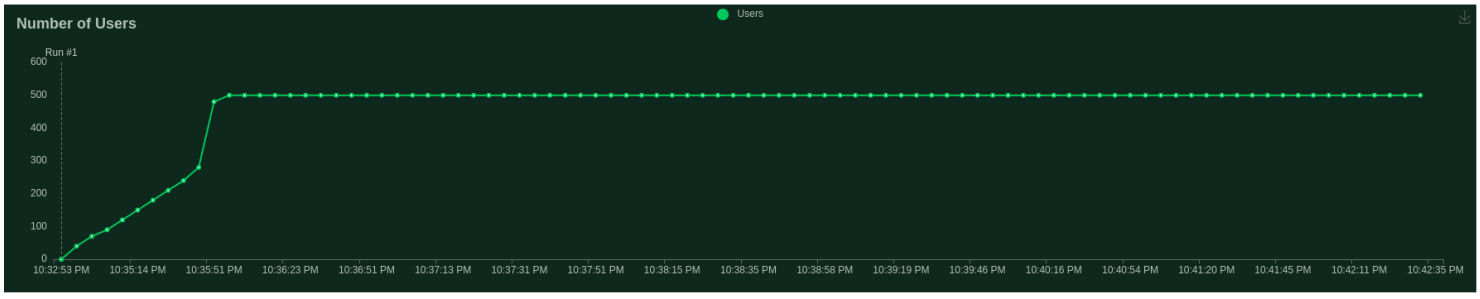




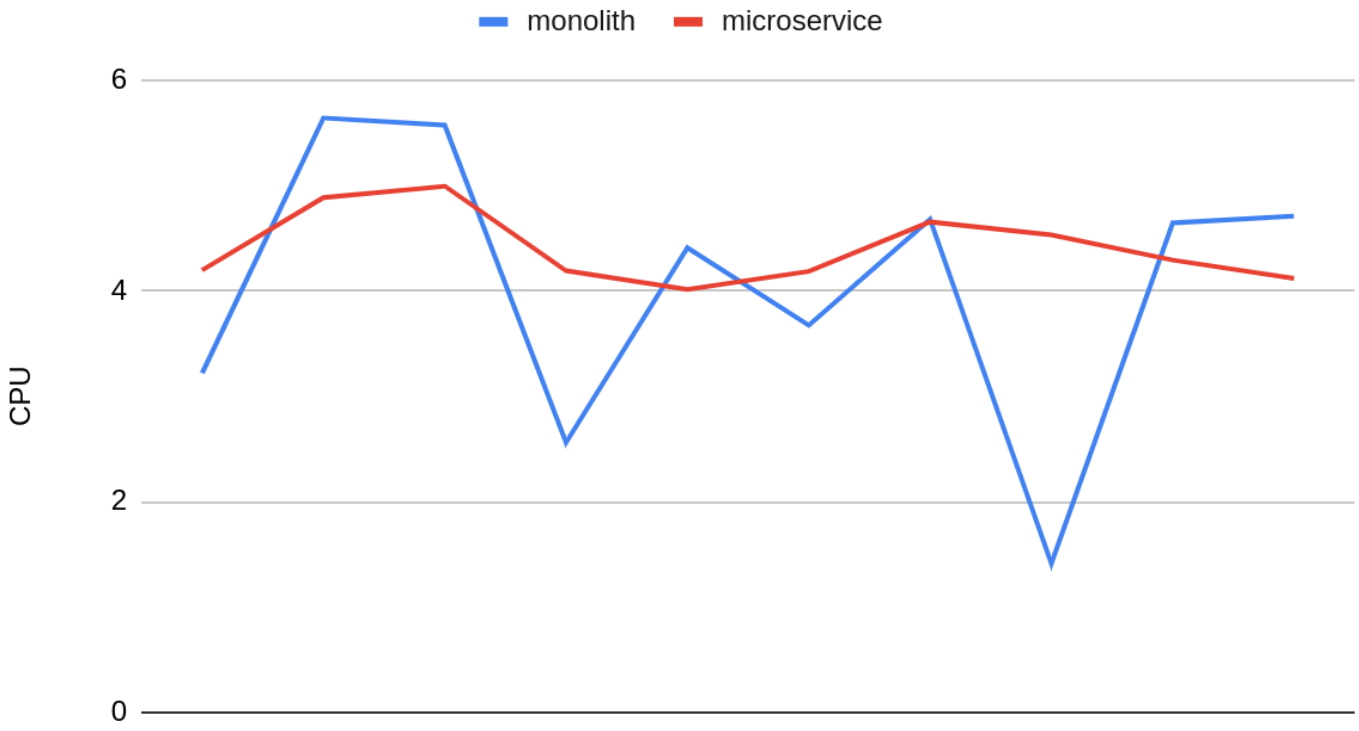
Locust Report: microservice



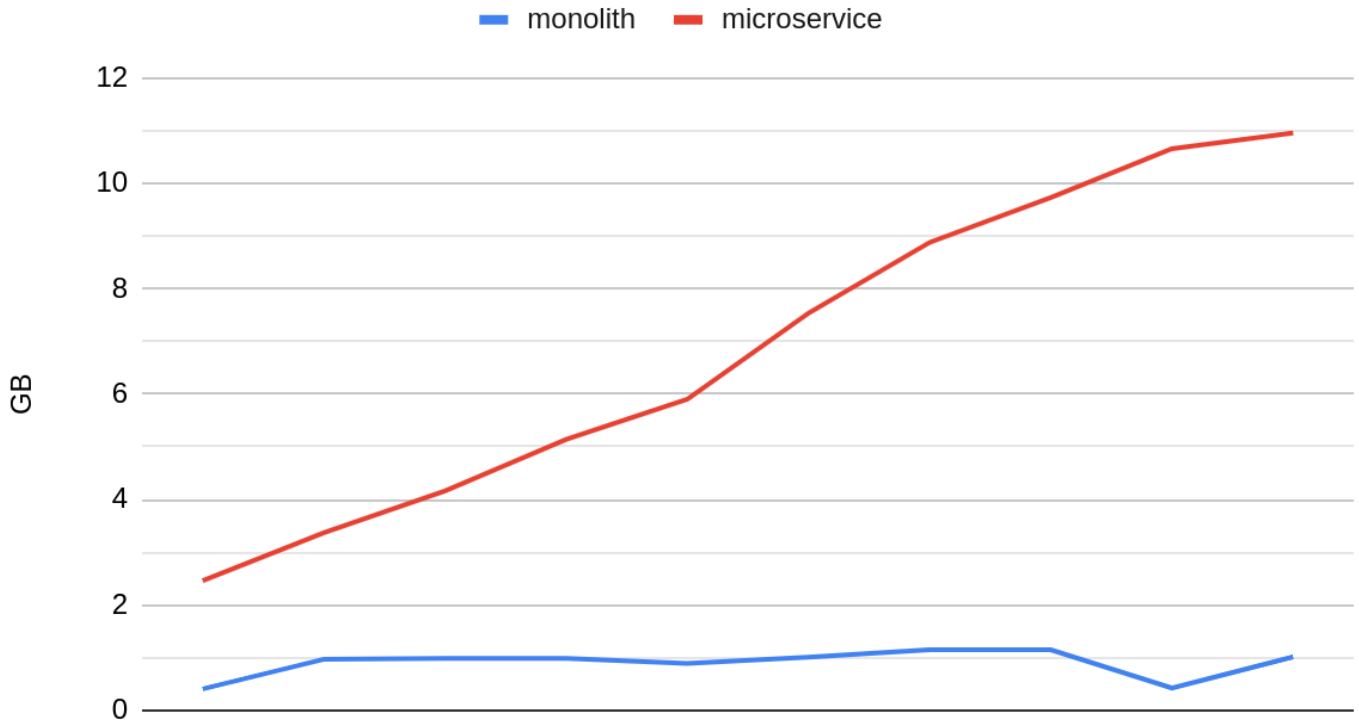




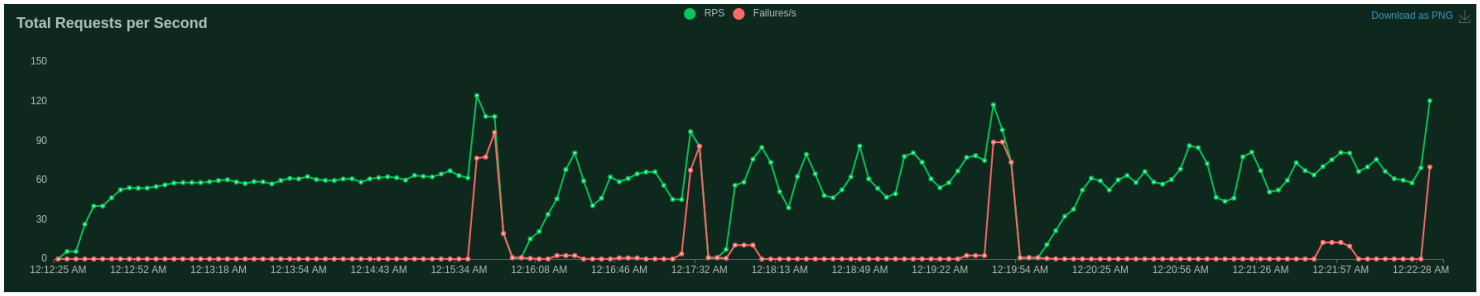
CPU Usage

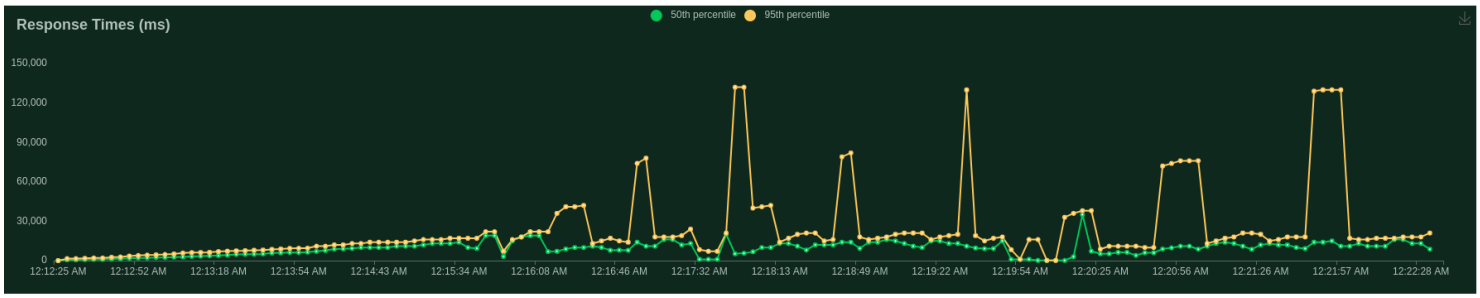


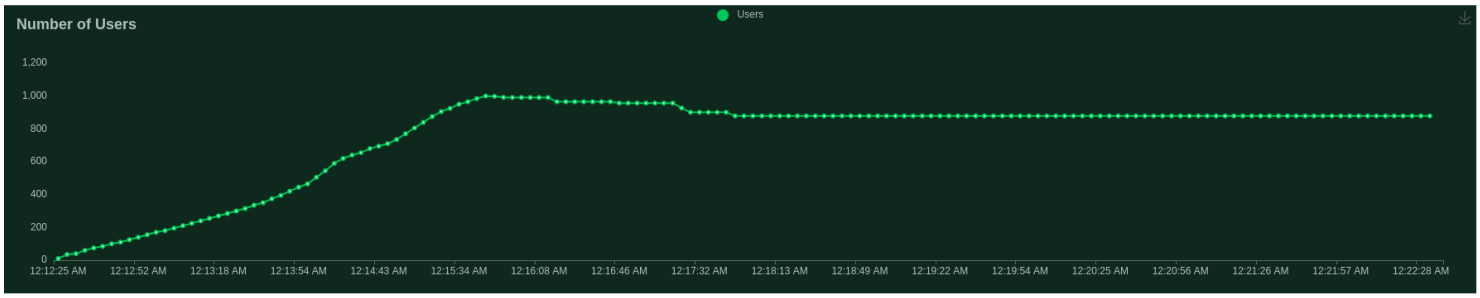
Memory Usage



Locust Report: monolith







Locust Report: microservice