Bachelor's Thesis

Benchmarking Microservice Applications in Cloud Computing

Pavlos Evgeniou

University of Cyprus

Department of Computer Science

December 2020

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Benchmarking Microservice Applications in Cloud Computing

Pavlos Evgeniou

Επιβλέπων Καθηγητής

George Pallis

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

December 2020

ACKNOWLEDGEMENTS

I would like to thank and express my appreciation to my supervisor, Professor George Pallis, who gave me the opportunity to work on a topic that is very important and useful. I also got to learn about a topic that is trending in Computer Science, but it is not in our curriculum. He supported me in everything I needed, and he was always available for questions.

My special thanks also go to Mr.Moysi Symeonidi and Mr.Zacharias Georgiou, who are part of LINC in the department of computer science at University of Cyprus. They supported me and helped me with a lot of problems I encountered. Their instructions and suggestions were important in the completion of the research. On top of that, their knowledge and experience helped in solving and avoiding problems making the work a lot easier. Finally, I would like to thank my family and friends for being supportive during my studies and especially during these hard times we have found ourselves into this past year.

Abstract

The microservice architecture is one of the biggest trends in Computer Science the last few years. A lot of massive corporations are moving away from the monolithic design of their applications and moving into the more modular microservice design, so the need for benchmarking the microservice applications has increased. In the most recent years, a lot of research articles have been released on different ways to benchmark them.

In our research we thought and created realistic scenarios to test microservice applications and through analyzing and monitoring them to be able to find some of their bottlenecks. Even though the method we used is simple, it helps you understand the behavior of the application and fix any problems that may occur. Through this research even someone with little experience with the microservice architecture, will be able to understand the basics on how to monitor a microservice application and recreate the same or different scenarios for a different application.

Table of Context

List of Figures

Chapter 1

Introduction

1.1 Motivation

In this day and age, thousands of programs, applications, services and websites are created daily. At the beginning, almost everything was built using the monolithic architecture, which meant that all the companies were building their applications as single units. After a while, it became apparent that this design was very hard and time consuming to maintain and update. The industry needed a way to switch away from the monolithic design. They needed a design which would be easier to develop, maintain and update since the cost of creating and maintaining a monolithic application was expensive and difficult. This led to the architecture known as microservices. Their modularity, ease to develop, and maintain, changed the market as we know it. During the last decade, a lot of massive corporations switched from a monolithic design to a modular design using microservices. Figure 1.1 shows the market growth of microservices the last few years and the forecast for the future which looks very positive.

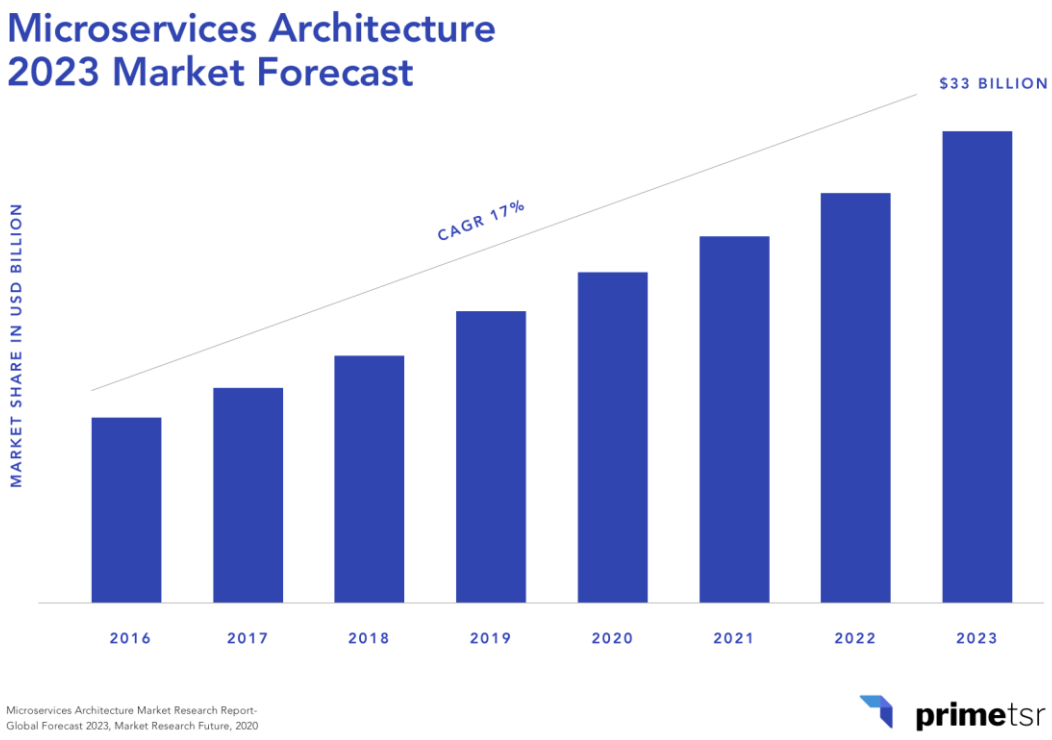

Figure 1.1: Microservices Architecture market Forecast by primetsr[12], an IT consultant.

Along with the microservices a lot of new services and products have appeared, that complement the microservices architecture. A few of those include Docker, Kubernetes, rkt and many more. Programs like Docker[7] allow you to set up containers where you launch your services. It makes the development, launch, and update of a microservice a lot easier. One of the biggest benefits of using one of the above programs is that through the containers, it allows you to build your website/application/program in minutes at any server you want, since the containers have everything your service requires. They also allow you to scale certain services in your build according to your traffic, which is one of the most important benefits of using microservices in containers.

Therefore, a way to monitor your containers was also needed. Building your programs on the cloud infrastructure without a way to find the problems that occur would make things harder instead of fixing them. For that reason, services like cadvisor[9], Prometheus[8], and Jaeger[6] were created.

In conclusion, microservices architecture is a big part of the present and a bigger part of the future. Through understanding the theoretical background of microservices' design patterns, this study will showcase monitoring, testing, and evaluating the microservices performance and try to reveal any bottlenecks the microservice applications may have.

1.2 Challenges

The main goal of the research is to be able to analyze the results and identify bottlenecks in microservice applications, through the implementation, the monitoring, and testing of microservice applications in cloud computing. First and foremost, the understanding of the theoretical background of microservice's design patterns was very important. The knowledge of how the microservices work and communicate between them is a very essential step in our research, since it allows us to create a real example, find the proper application to monitor it, and stress test it.

Consequently, finding or creating a proper microservice application that is complicated enough to resemble a real-life workload was one of the hardest obstacles. For the results to be accurate an application was needed where a lot of services where communicating between them and proper measurements could be taken during testing and benchmarking. Additionally, being able to monitor, stress test, and run the application on one machine was a difficult thing so a server where the application was running along with the monitoring programs was needed. After having a proper structure where a device was acting like a server and another creating the requests, different scenarios needed to be created. Using the scenarios, our goal was to test realistic situations that might occur and through the testing, understand the microservice we are benchmarking and its weaknesses.

Lastly, the internet connection is a very important part of the stress testing since it limits the amount of the requests we can create to the server, causing the request to fail sometime. The testing happened in a closed network helping to limit the fails but there were also limitations to the max capacity of requests you can make.

1.3 Contribution

The research's aim is through the testing of a microservice application to be able to analyze the results and reveal possible bottlenecks in the microservice applications that are running in the cloud. Through the results of the research, we hope that people interested in this architecture will be able to understand better what it has to offer and what are its drawbacks.

At the same time, this research will showcase superficially how to have a fast and complete microservice application with a lot of the important monitor tools. Our hope is that through the research, the importance of monitoring your applications, the knowledge of its drawbacks and possible bottlenecks, will help with avoiding equivalent problems.

1.4 Outlined Content

Chapter 1: Introduction

The first chapter introduces the reasoning behind the need of the microservice architectures and the positive impact it had on the field of Computer Science. It also showcases the fast growth of the architecture in the market until today, and the predictions that it will continue to grow through the following years. We briefly go through the challenges we encounter while contacting the research and we explain their importance. Lastly, we mention the area we want to contribute to and how we hope to help.

Chapter 2: Literature and related work

Chapter two is about the literature on microservice architecture and on similar work. It explains summarily a few of the research on microservices and a few of the other papers on the topic of benchmarking microservices.

Chapter 3: Methodology

The third chapter analytically explains the applications and services used to contact the experiments. It explains how we use Docker[7], and what applications are used for monitoring the microservice application. Finally, we describe in detail the way the data is extracted and analyzed for the experiments.

Chapter 4: Experiments

The fourth chapter is about the experiments we conducted to benchmark the microservice application. Through analyzing the results of the experiments, we are able to understand some of the problems and bottlenecks of the microservice, like the absence of a load balancer.

Chapter 5: Conclusion

In the last chapter, is the conclusion about the research and some examples on how it can be extended in the future.

Chapter 2

Literature and related work

2.1 Literature

In the recent years with the transition to cloud services, microservice architecture has become a lot more popular due to its scalability, modularity, ease of maintenance and fast deployment. For this reason, a lot of researchers are investing their time trying to analyze the microservice architecture and determining through their research some of the benefits and negatives of the aforementioned architecture. Another massive target of a lot of the literature, is to help the reader to become accustomed to the newer architecture.

A research was published in 2018 [2] by Soldani J., Tamburri D. A. and Van Den Heuvel W.-J that focused on the benefits and drawbacks of microservices. They aimed to analyze the microservice architecture and through the research explained in detail the pains and the gains as they named them of a microservice architecture. In their research they analyze in detail the positives and the negatives in the different stages that take place during the creation of a microservice application. More specifically they used 3 stages, the design stage, the development stage, and the operation stage. In the operation stage, they illustrated the weights of the resources needed, and in their illustration, you can see that the network is a big drawback in cloud microservices, something that I noticed and mentioned in the challenges.

Another research was published in 2017 [4] in which the authors, Vural H., Koyuncu M., and Guney S., wanted to help people understand the new architecture that is microservices. It is a more general research focused on being a steppingstone into the

modular creative world that is being created through the microservices. The questions that they answer through their research is the type of research that is currently being contacted on the microservices architecture, what are the reasons behind the microservice's research and what are the standards and existing tools on the forenamed architecture.

The above are some examples of the research on the microservices architecture. The amount of research material is increasing by the years due to the popularity and importance of the architecture. The knowledge on microservices and their possibilities are increasing by the years, with more and more researchers discovering new benefits and negatives of the architecture, helping it progress and fix the drawbacks. This will hopefully lead us to a modular design that is close to perfection.

2.2 Related work

A lot of research on benchmarking microservices are helping you understand the new architecture that has surfaced in the last few years. Different researchers, professors in universities, and some developers, are trying through their experiments to understand and find ways to make the architecture better. There is a lot of research about benchmarking already and there is more coming which makes it easier for the developers to study more specific scenarios according to their needs. Some of those are the "Benchmarking the Performance of Microservice Applications" [1] by Grambow M., Wittern E., and Bermbach D. where they test their approach on benchmarking microservices and they evaluated it with their prototype. Another research is the "Benchmarking Microservice Systems for Software Engineering Research" [5] from Zhou X., Peng X., Xie T., Sun J., Xu C., Ji C., and Zhao, W. where they conduct research on an open-source system, and review literature to help them determine the chasm between the current benchmark system and the microservice systems. This is just a small percentage on the research that exists on benchmarking microservices. The architecture is one of the most popular ones, so new articles and new research are being released every year about it.

Chapter 3

Methodology

3.1 Methodology Overview

The research methodology can be divided into four categories, Building phase, Testing, Data Collection, and the Data processing. We can see the overview of the methodology in the Figure 3.1.

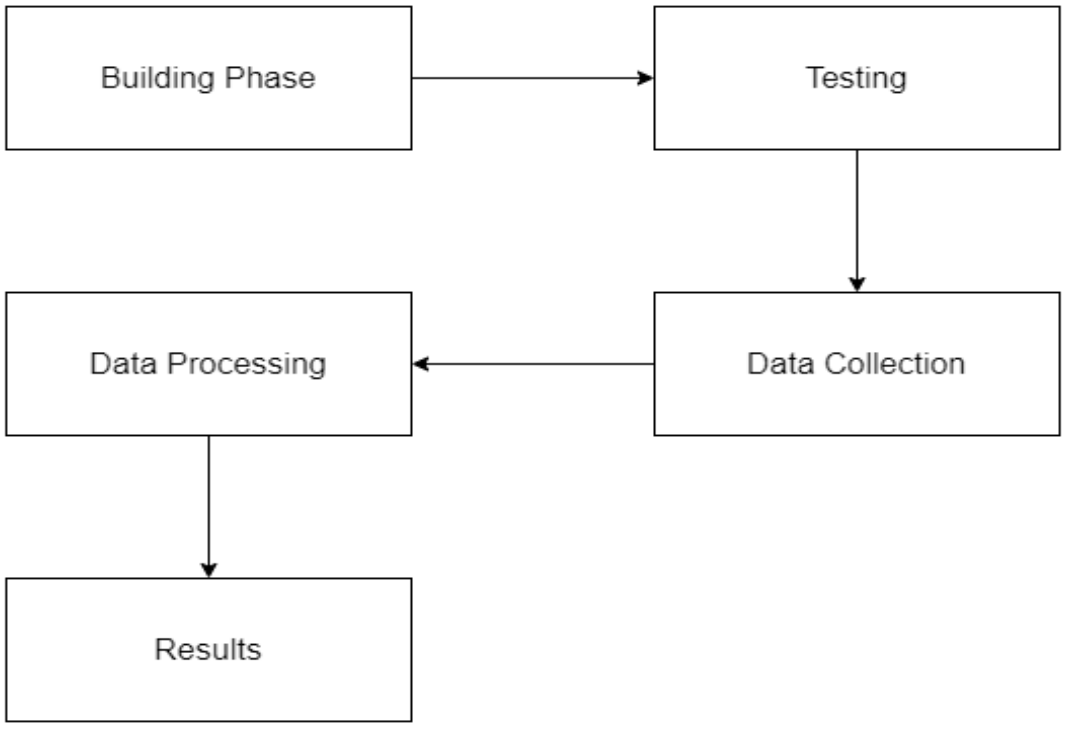

Figure 3.1: Overview of the methodology categories

At first, we set up the monitoring programs and the microservice application that we want to stress test. After we make sure that everything works correctly, we begin to stress test the application depending on the scenario we want to emulate. Immediately upon the end of the stress test we collect the data and then we process it. Lastly, when we have the processed data, we can see the results of the scenario that we emulated.

3.2 Building Phase

We chose to use Docker[7] as the environment where we were going to build the microservice application and perform our experiments, due to the plethora of features it has and the support it has from the community. It is also one of the most popular environments for microservices.

Before stress testing our microservice application we need to have a good monitoring system that we will let us extract all the data we need from the tests. There are a few applications that you can use to monitor your microservices and for each scenario some of them may work better than others. For our use case we decided to use Prometheus[8] with cadvisor[9]. Prometheus is a monitor application that you can run as a container and you can get metrics in detail for your server. It allows you to analyze and create detailed graphs with the metrics since it permits calculations. We used cadvisor because if you connect it with Prometheus, it allows you to monitor each container specifically, allowing us to analyze and understand the microservice application we chose better. Lastly, we used Jaeger[6], a monitoring software that allows you to track the requests that target the server it is installed on. It is a very useful monitoring program since it allows you to understand which services communicate between them.

Figure 3.2: An example of a Prometheus graph

After setting up the monitoring software we needed to find or create a microservice that was complicated enough to represent a real-world example. We discovered a demo that a senior developer at Google created [10]. In the Figure 3.3 which was provided by the creator of the application, we can see the services that form the microservice application.

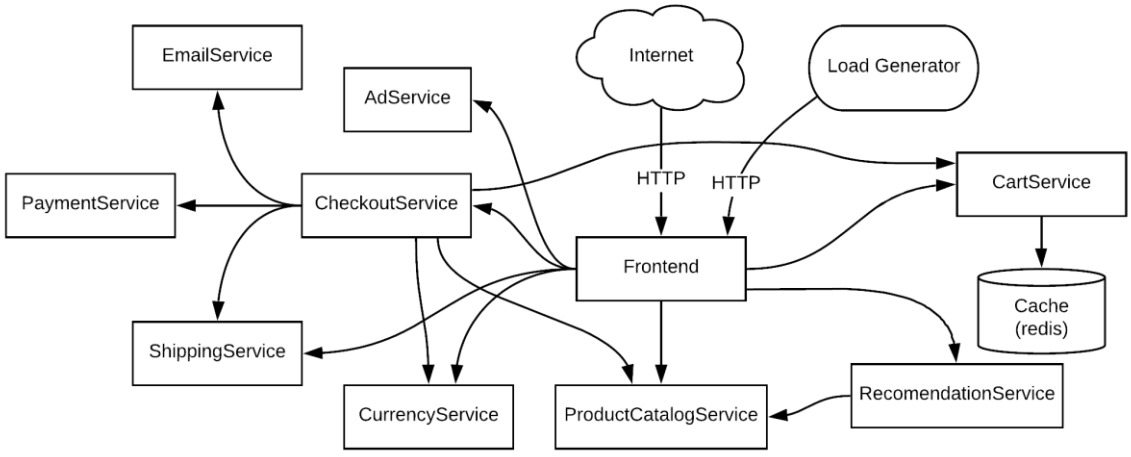

Figure 3.3: A design to showcase the microservice.[10]

In the aforementioned Figure, we can see all the microservices and how they communicate between them. It is a very helpful layout, since it allows us to understand and confirm a few of our experiments later. The application was built for Kubernetes, but they shared all the files along with the dockerfiles making it easier to create a docker compose to start the application on Docker[7]. It represents a shop with nine products, a

cart, advertisements, recommendation list, and a few other features helping us create a realistic environment for the experiments.

3.3 Testing

At the testing phase we wanted to stress test the microservice application using scenarios that are common in real-life. For example, a scenario where a lot of users visit a product that is popular, and then move to the checkout page after deciding that they want to purchase it. Creating realistic scenarios is important, as it will show the weaknesses of the application and the problems that may occur if it was going to launch for the public. There are a lot of ways to stress test your application and for our case we chose Jmeter. It offers a variety of ways to create a load and for different use cases.

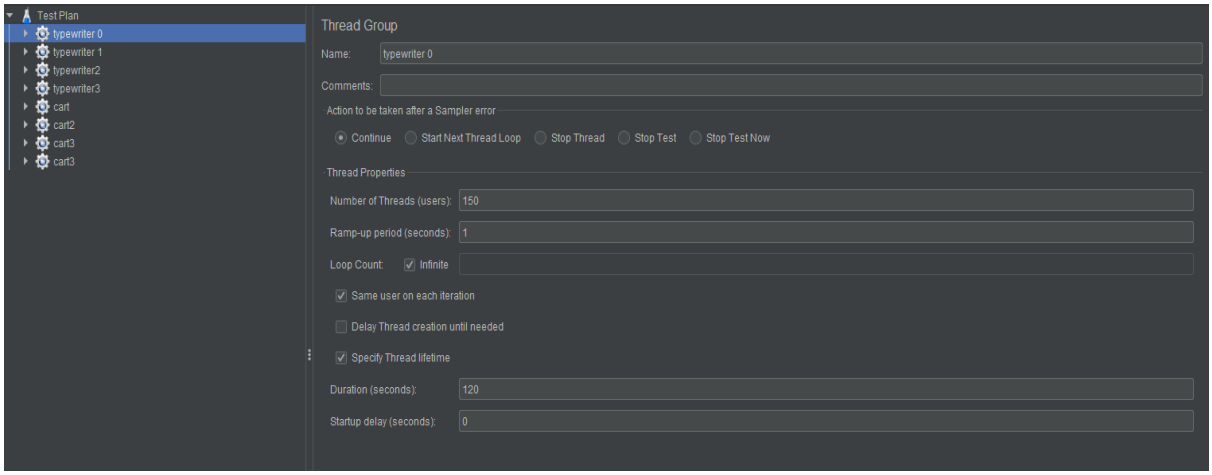

Figure 3.4: Example of a Jmeter Test

For our case, the choice of load for the application that we were testing was http requests, where we simulated a number of users that were requesting some pages depending on the scenario we were simulating. While the simulation was running, Jmeter was displaying data like the success of the request, the latency and was extracting them in the end.

3.4 Data Processing and Data Collection

Even though Prometheus[8] can display detailed graphs as the Figure 3.2 shows, we needed a way to extract the data to be able to create our graphs and analyze them. There are a lot of ways to extract the data from Prometheus and in different formats. The preferred choice was csv because it is a very easy format to work with using Excel. For the extraction we used an open-source code from GitHub [11] which allows you to extract all the data for a certain container. We use it to extract the data every 10 seconds for the duration of each experiment.

After the experiments were finished, we had a csv for every container from the Prometheus data and a csv that was extracted from Jmeter. From the data that was extracted we took the time that each container was using the CPU and the memory usage for the containers that were related to each experiment, and we used them to create graphs. The latency from the Jmeter was also used to create a graph and find the average latency of each experiment.

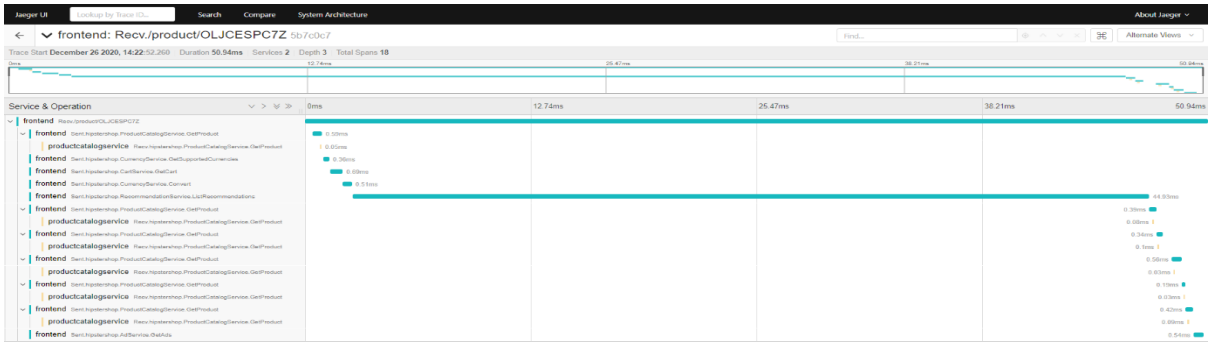

Figure 3.5: Example of a request displayed by Jaeger

In our experiments Jaeger[6] was one of the most important tools since it allows us to search the requests that happened for each service of the application. The Figure 3.5 is an example of a request taken from Jaeger. The main use of Jaeger in our experiments was to find out which containers were used during each request, which ones were the slowest and how they were affected by each experiment. It also let us analyze each request independently and think about new experiments or different things to try to help the performance of our microservice application.

Chapter 4

Experiments

4.1 Description of experiments

We contacted eight experiments which are simulating real life scenarios or testing different settings to see how they affect the performance of the microservice application. For each scenario, we gathered the data as mentioned in Chapter 3 and created the graphs. Even though we only used one microservice the scenarios an easily be replicated for every microservice application.

Before testing the applications, we needed some base statistics to be able to compare the results of the experiments, so for that reason we let the microservice run without creating any load. Through this we want to observe the CPU time and the memory usage when the application has no requests or any kind of load. The graphs when the system is idle will give us something to compare the results from the experiments and see how they affected the application.

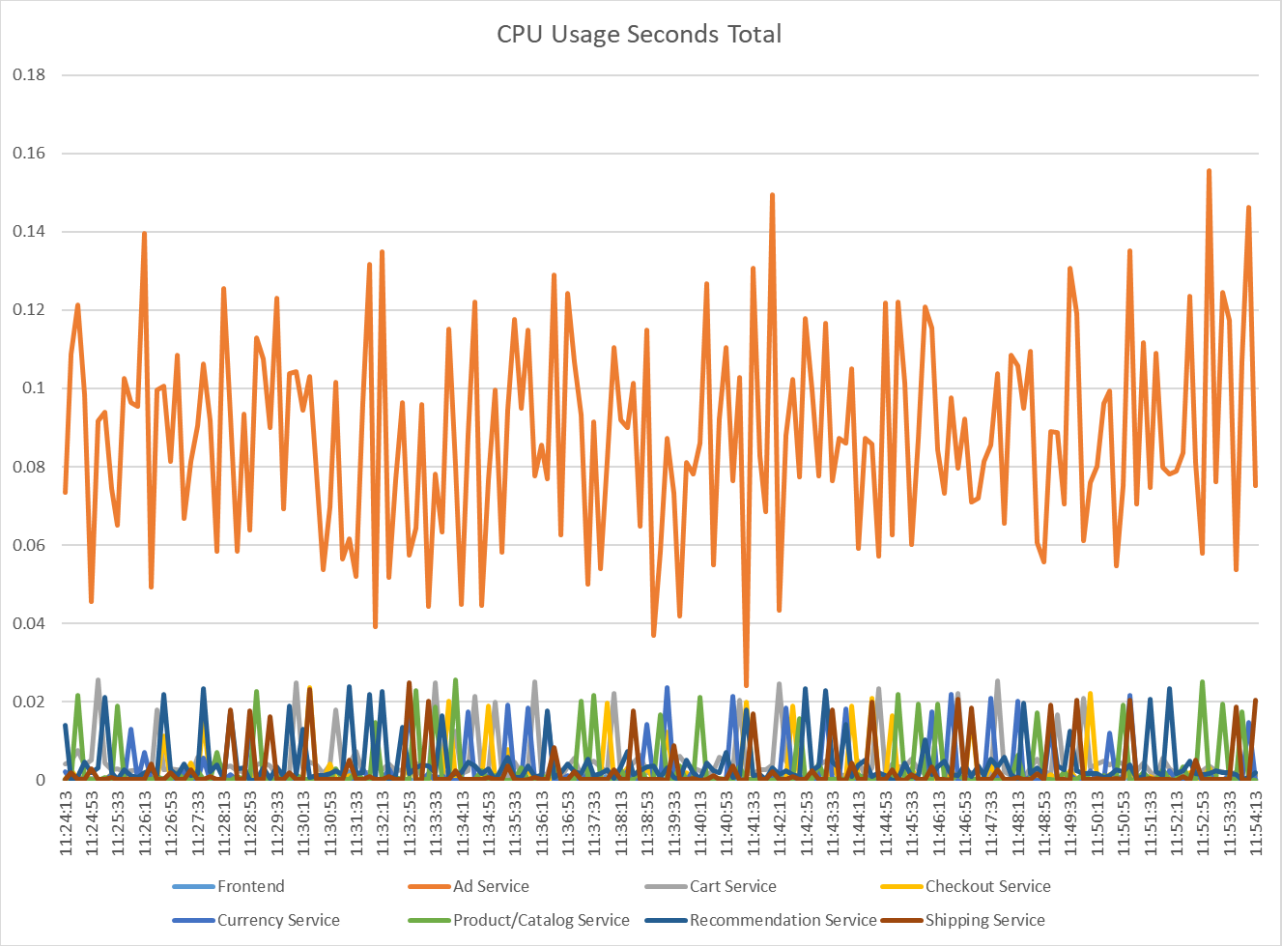

Figure 4.1: CPU usage when the microservice application is idle

In the Figure 4.1 we can see the CPU usage in seconds when the application is idle. As we can see from the graph the application does not require a lot of CPU time since it is in an idle state. The container with the highest usage is the Ad Service container with only 0.14 to 0.15 seconds and the other containers have the same CPU usage from 0 to 0.02. All the containers have moments that have a higher usage, but it is not stable, we can notice spikes every few seconds and then the usage drops.

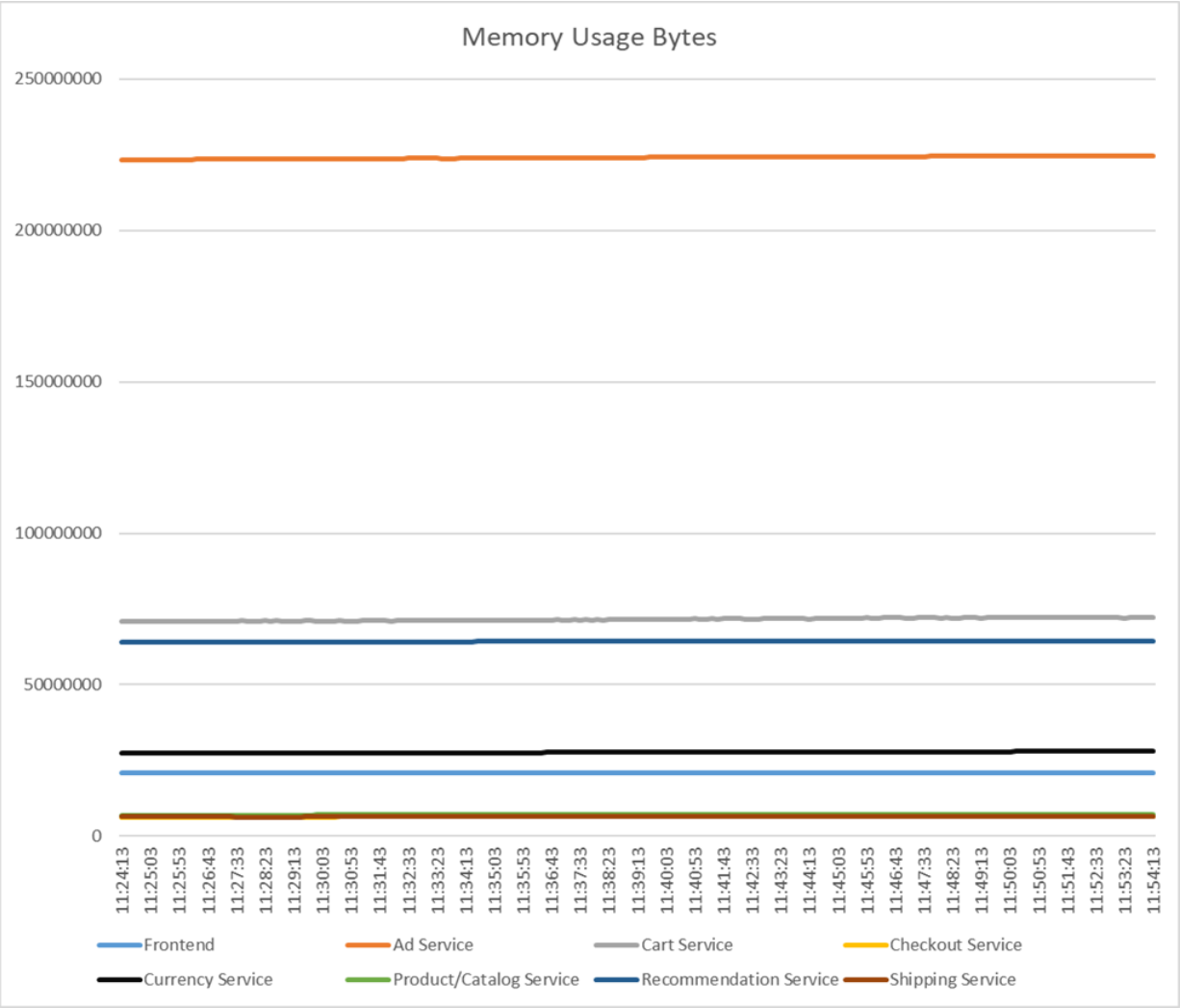

Figure 4.2: Memory Usage when the application is idle

The Figure 4.2 shows the memory usage of the system when it was idle. All the containers seem to have stable memory usage that is consistent. The service with the highest memory usage is the Ad Service with 223MB of usage, with a big difference from the other services. The rest of the services have less than 100MB of memory.

From the aforementioned figures, we can observe that the microservice application does not need a lot of resources when it is idle and one of the containers that will need a lot of resources will be the Ad Service. From the Figure 3.3 and from the knowledge about the microservice applications we can expect the Frontend container to need a lot more resources when the application is stressed due to the fact that it is the main service, that connects and helps the others communicate. The Frontend is also the one that handles the requests from the users.

4.2 Experiment I

In the first experiment, we emulated a simple scenario where the users that create requests to the application were increasing by one hundred every four minutes. This experiment is meant to emulate a real-life scenario, where a website receives a different number of users each hour and to see how our application handles it. At the beginning, one hundred users were requesting the main website (localhost:8080/) and at the sixteenth minute it reached the five hundred users that were constantly requesting the site. The experiment ended in fourteen minutes after the last hundred users started the requests.

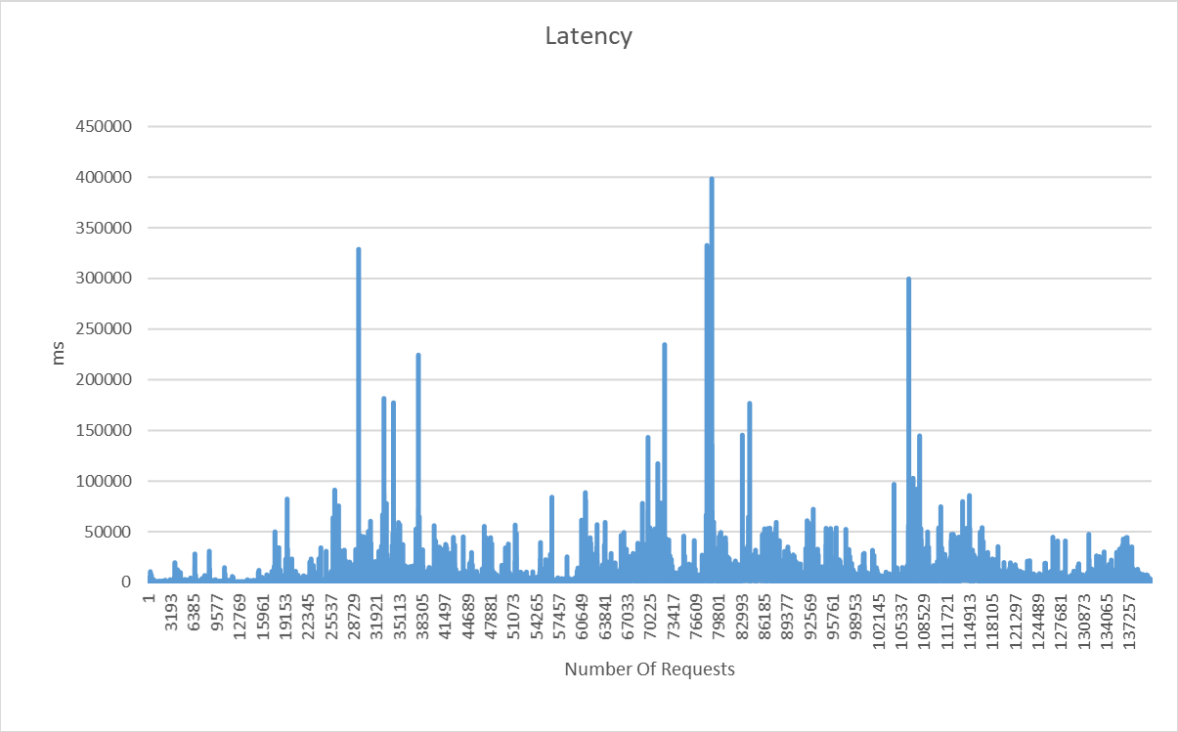

Figure 4.3: Latency for Experiment I

The average latency for the experiment was 2.3 seconds and as we can see from the Figure 4.3 there are a few requests that have a massive latency which was probably caused due to a network problem. Although, the latency of those requests is bigger, it did not change the average latency by much due to the number of requests that happened. From the figure we can also notice the latency started increasing after a certain amount requests because the number of users had increased.

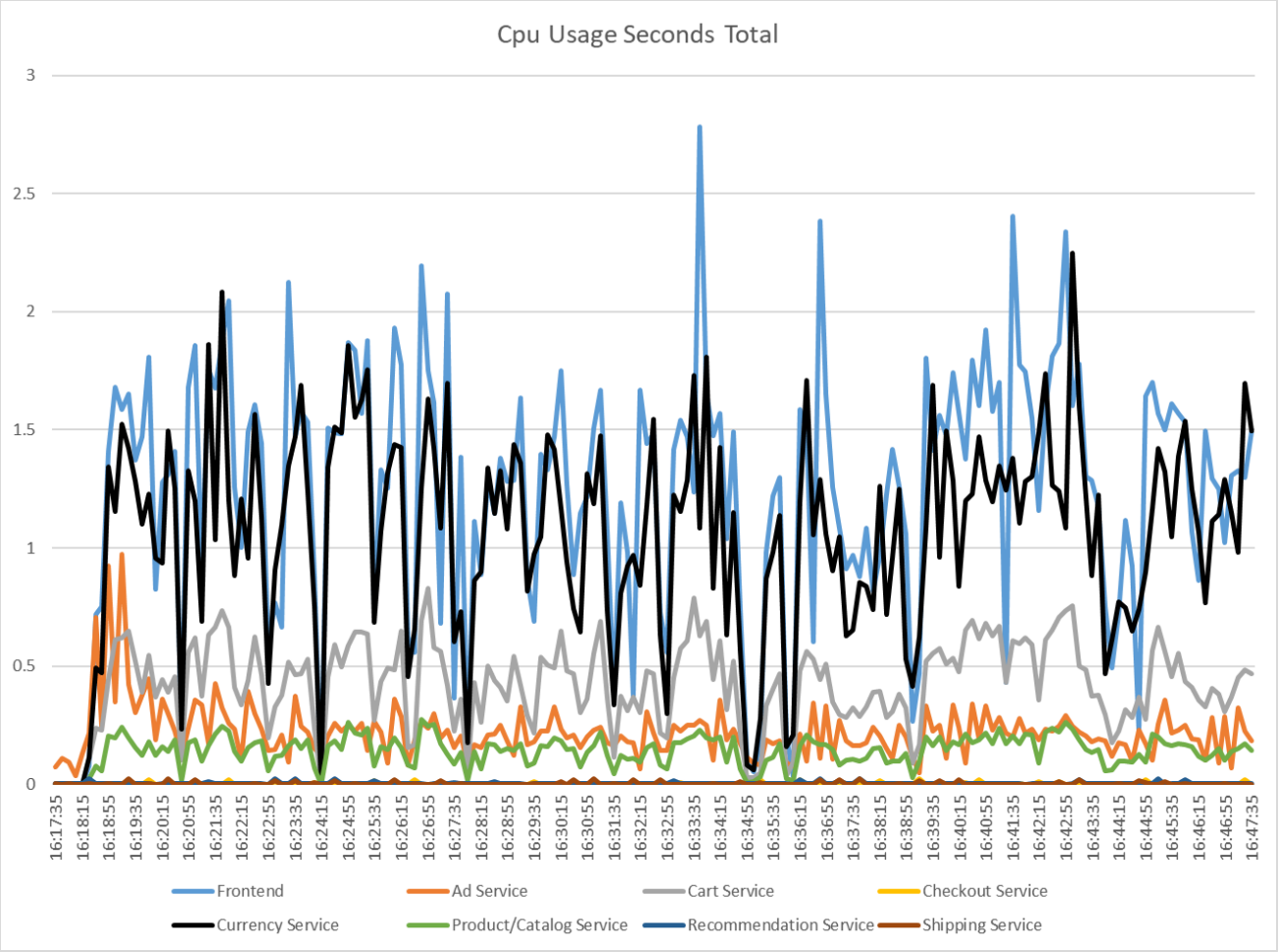

Figure 4.4: CPU usage for Experiment I

The Figure 4.4 helps us observe that the CPU usage for Ad and Product/Catalog services has slightly increased and for the Frontend, Currency, and Cart Service has increased dramatically from the idle numbers. The Checkout, Shipping and Recommendation services' usage seems to have stayed the same. From the Figure 4.4 we can also see that the CPU usage has increased in some services when the experiment started, it did not increase further when the new users started their requests. The usage of those services kept fluctuating the same way as the beginning. At 16:33:35 which is around the time where the last users join the network, we can see that the Frontend reached its highest CPU usage at almost 2.8 seconds. Apart from that spike it is very hard to understand from the Figure where the rest of users started their requests.

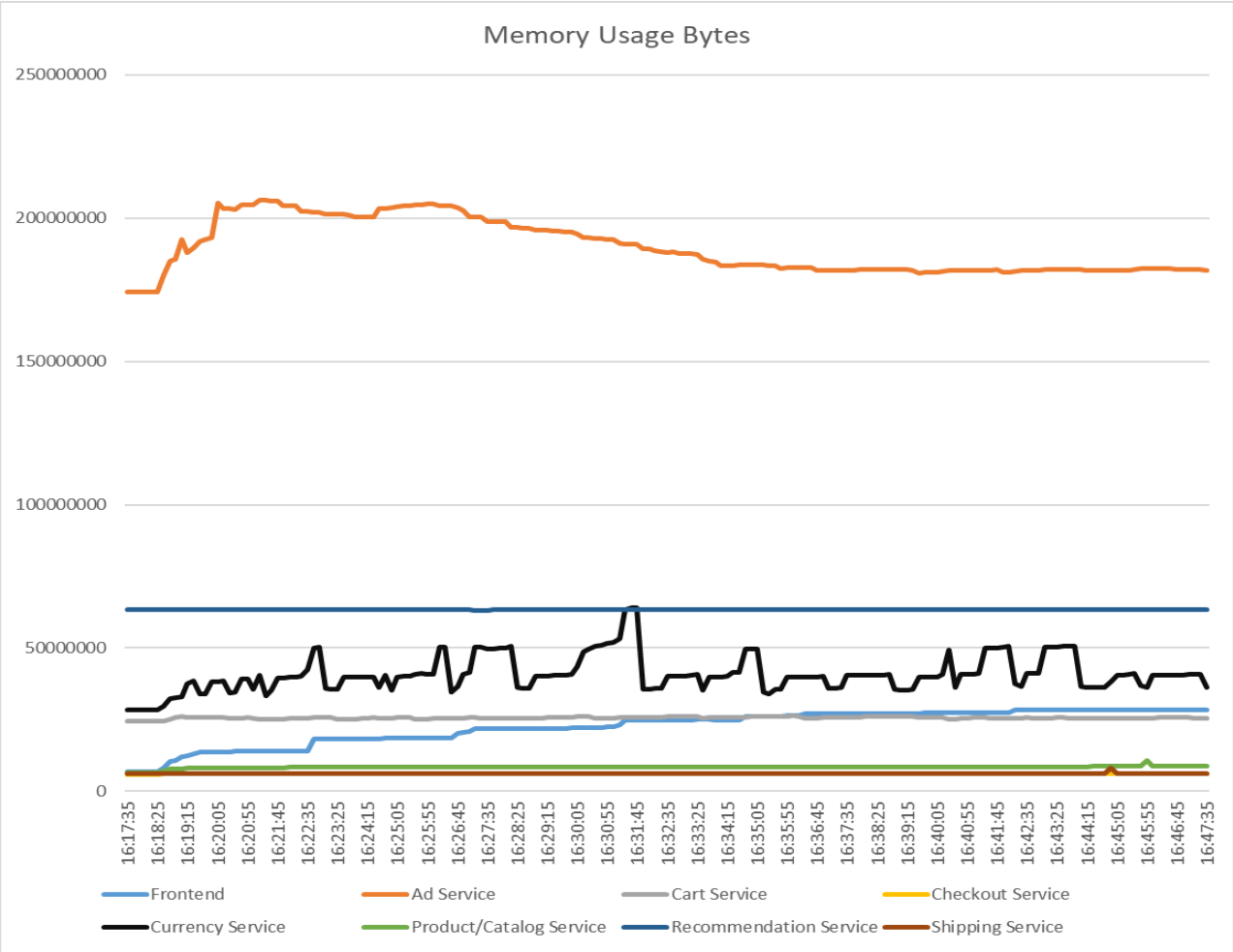

Figure 4.5: Memory Usage for Experiment I

The memory usage of the Frontend is increasing every few minutes, at the times where the new users were starting their requests, but as we can see in Figure 4.5, it slightly increased even after we had reached 500 users in Jmeter. The memory usage by the Ad service slightly increased at the beginning but then it returned to the previous value slowly. Also, the memory usage for the Currency service was fluctuating and its values were marginally higher than its idle state.

At the Figure 4.6 we can see the request that happened at the Frontend and the services that needed to be used for each request. All the services that had a noticeable change in the CPU and memory usage area were all part of the main website and were all used in each request on the Frontend. The biggest changes were in the Frontend and Currency service which makes sense since every request goes through the Frontend and was using the currency service multiple times.

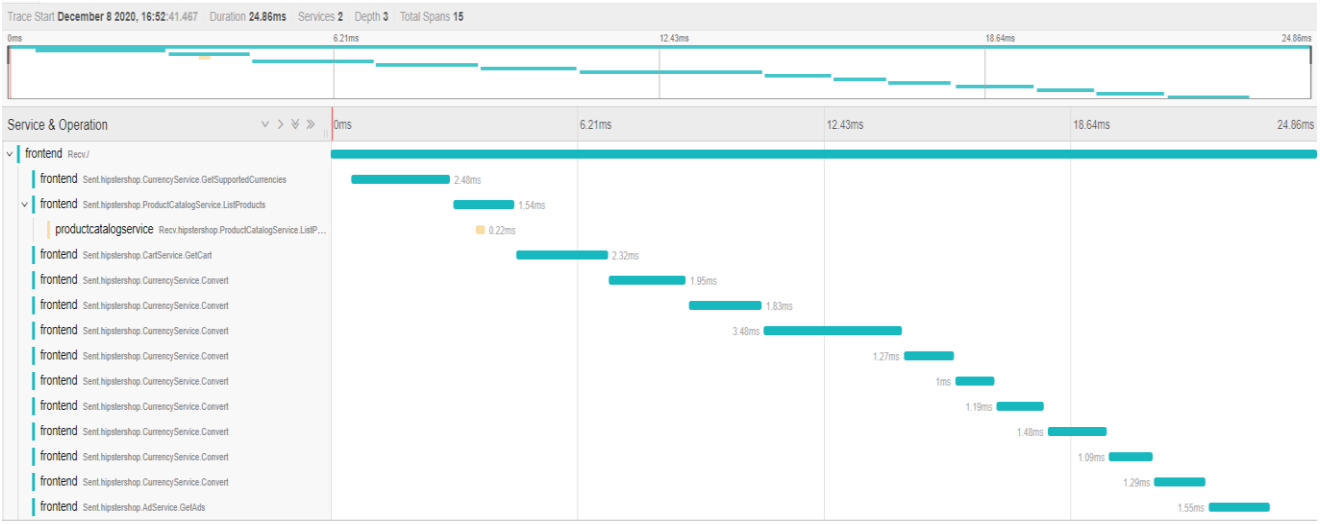

Figure 4.6: Screenshot from Jaeger showing a request at the Frontend

4.3 Experiment II

The next experiment was also about the Frontend service. Since through it pass all the requests even if someone is looking at a product or the cart, we scaled it and split the load of 3000 users into 100% at the Frontend 1 and 0% at Frontend 2, 25% at the Frontend 1 and 75% at Frontend 2 and 50% at the Frontend 1 and 50% at Frontend 2. Through this we wanted to see if the application will benefit by having the Frontend scaled and for this reason, we used 3000 users to make the results more noticeable.

The average latency of the request for the 100-0 test was 4.9 seconds from the request at Frontend 1, for the 50-50 test was 4 seconds at Frontend 1 and 3.9 at the Frontend 2 and for the 25-75 test 4.2 seconds and 5.5 seconds, respectively. The difference in latency between each test is small but it seems the 50-50 scenario was the best as expected.

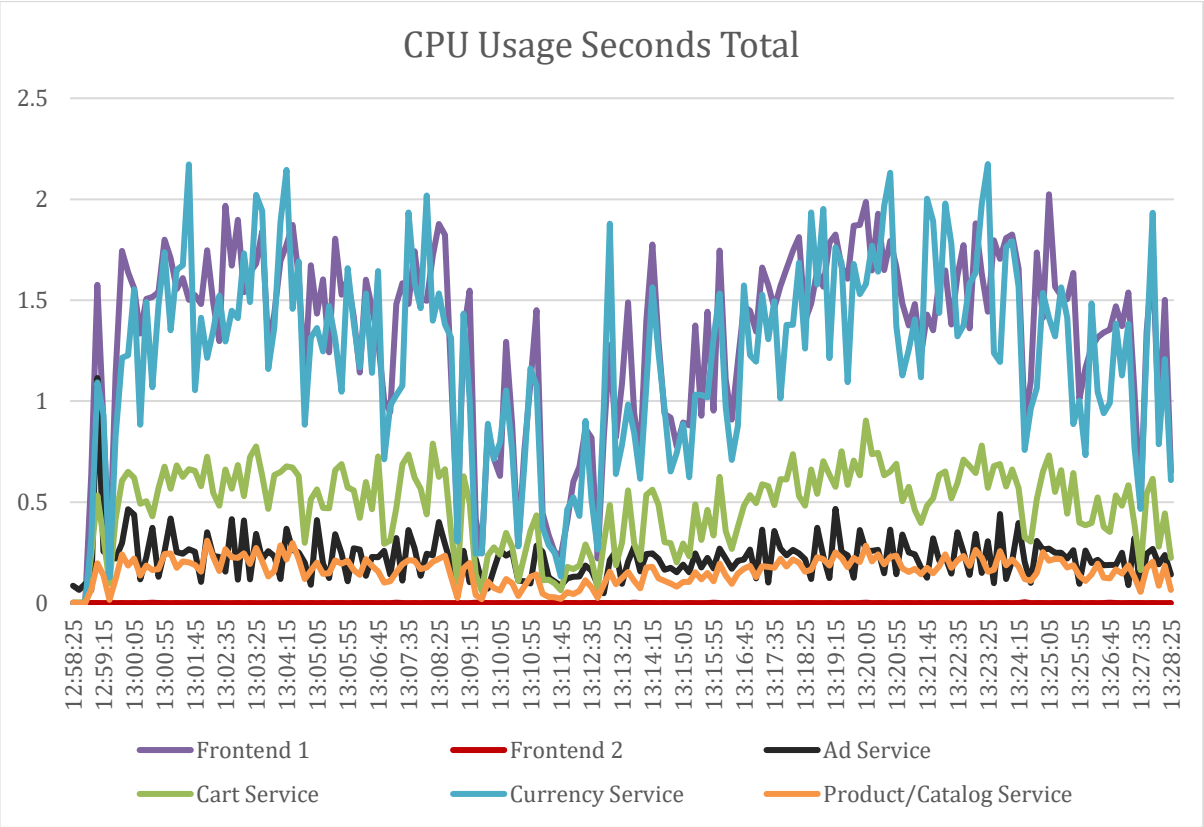

Figure 4.7: CPU usage for the 100%-0% scenario for Experiment II

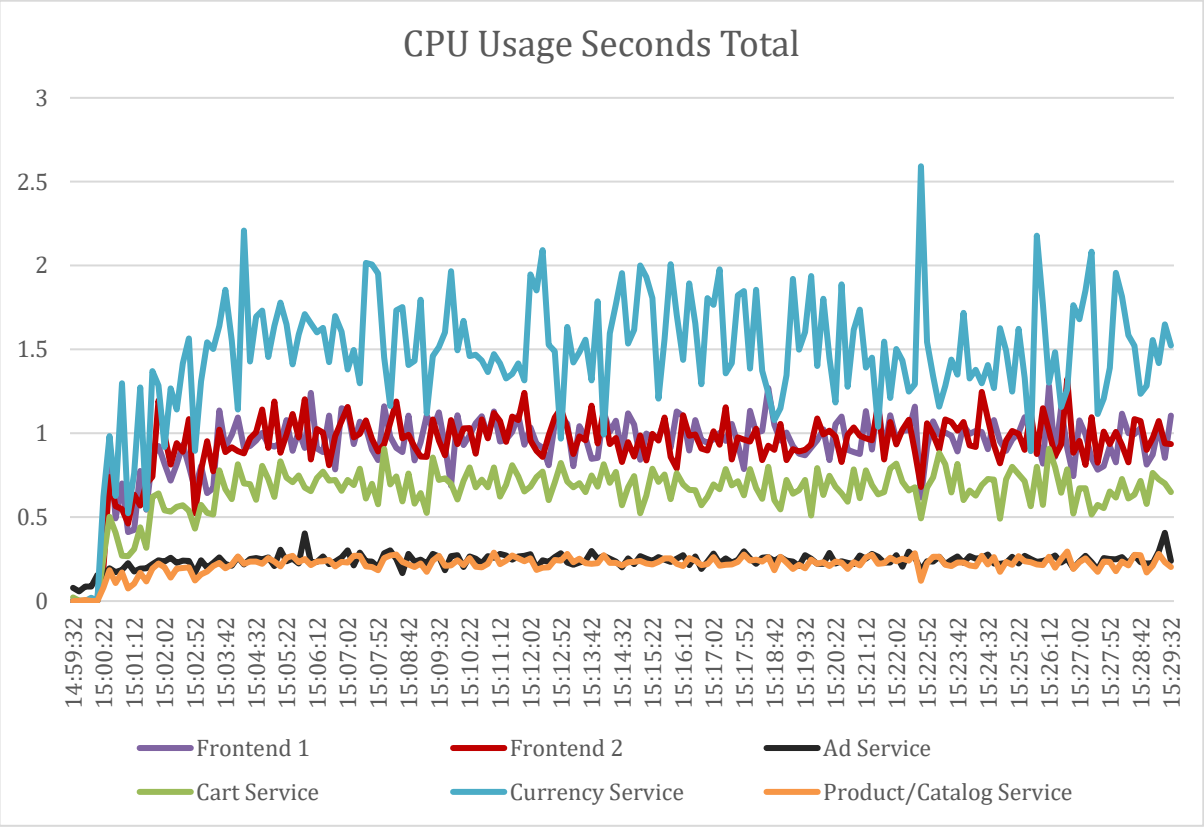

Figure 4.8: CPU usage for the 50%-50% scenario for Experiment II

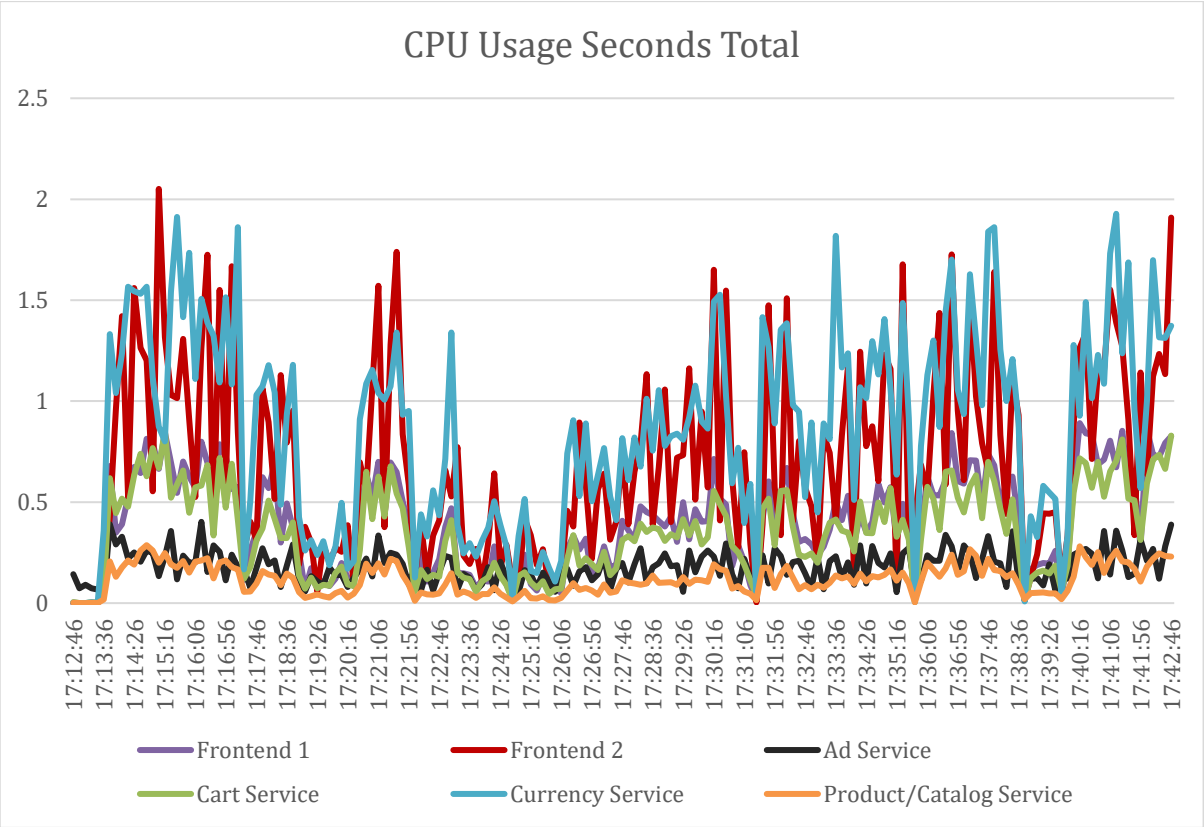

Figure 4.9: CPU usage for the 25%-75% scenario for Experiment II

In the above Figures we can see the CPU usage of each scenario. In all the scenarios the Frontend fluctuates between different values, but the rest of the services are fluctuating between similar values. In both 100%-0% and 25%-75% scenarios the CPU usage is less stable than 50%-50% where the difference in values that the services are fluctuating is smaller. Also, in all the scenarios where the load between the two Frontend services is not equal, we can see that there is a big difference between the usage of the two services.

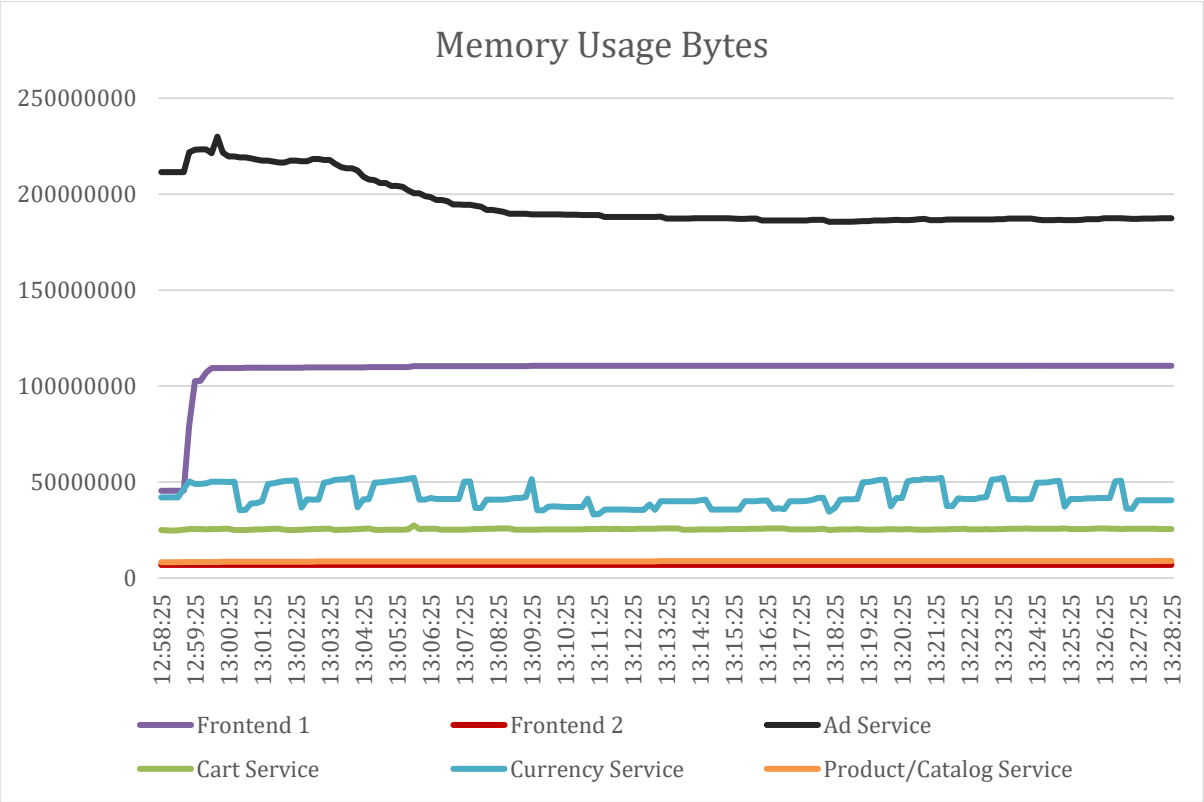

Figure 4.10: Memory usage for the 100%-0% scenario for Experiment II

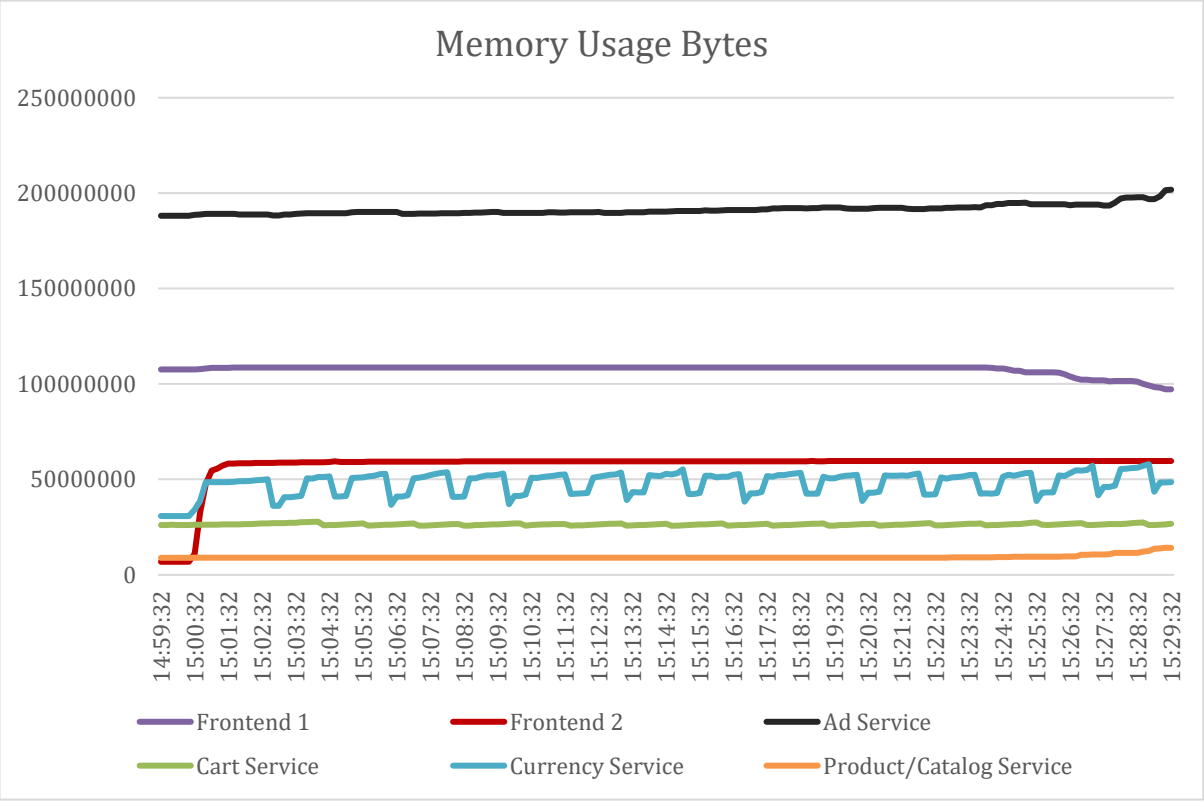

Figure 4.11: Memory usage for the 50%-50% scenario for Experiment II

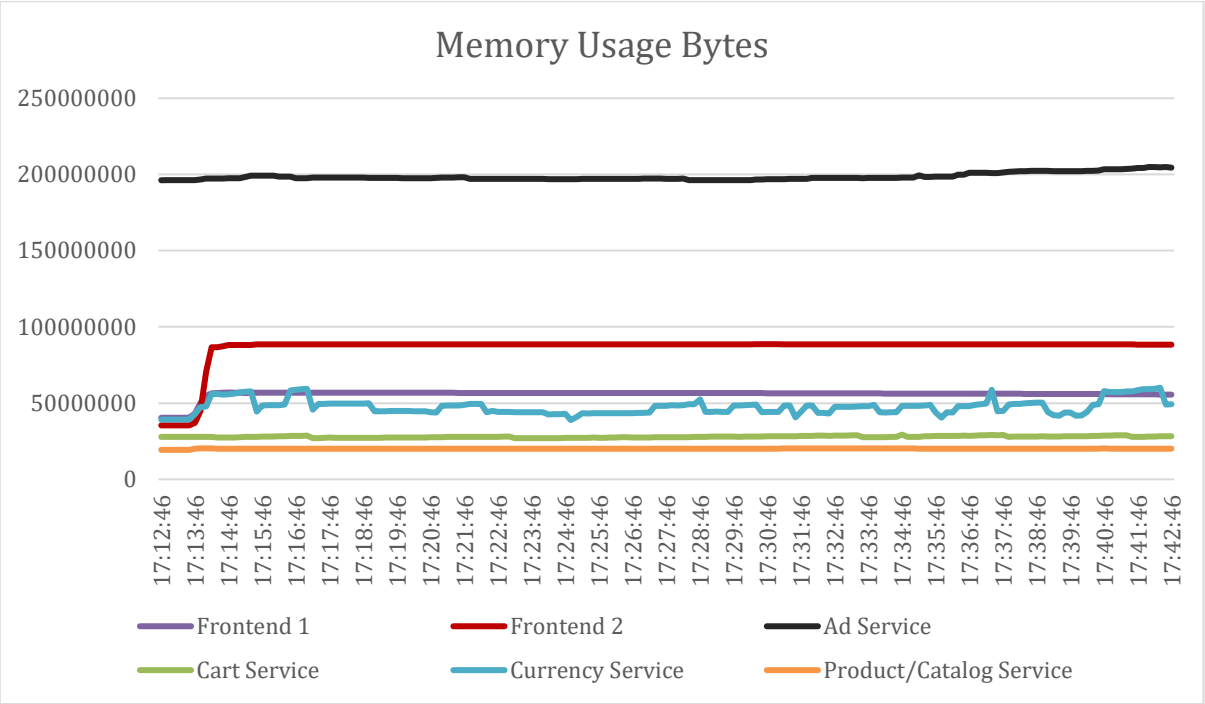

Figure 4.12: Memory usage for the 25%-75% scenario for Experiment II

In the Figure 4.10, Figure 4.11, and Figure 4.12 we can observe the memory usage of each scenario. All the services except the two Frontends are almost equivalent in all three graphs. In the Figure 4.10 the Frontend 1 uses more memory than Frontend 2 and at the Figure 4.12 the Frontend 2 has the higher usage, which is logical since in each scenario those are the ones that receive the higher number of requests. However, in the Figure 4.11 Frontend 1 uses almost double the amount of ram even though they receive the same number of requests.

From the aforementioned figures, we can see a benefit of scaling the Frontend service that receives the request if the load is distributed equally since it used less CPU resources, with the time of each service in the CPU being more stable with small fluctuations. Also, the latency had a small decrease compare to the two scenarios where the load was not equally distributed. Finally, as we can see in the following Figures in the 50%-50% scenario in Jaeger[6] showed that the maximum time a request needed in the system from the last 1000 requests was lower than the other two scenarios.

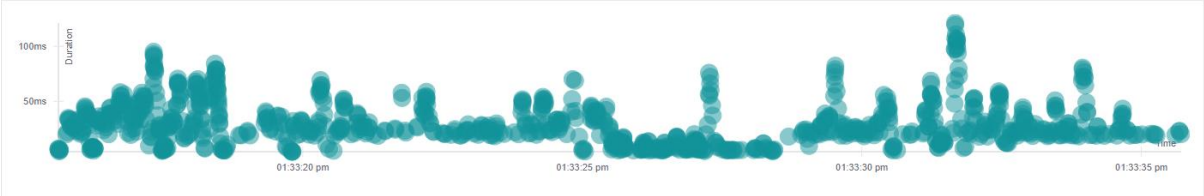

Figure 4.13: Jaeger Graph for the 100%-0% scenario

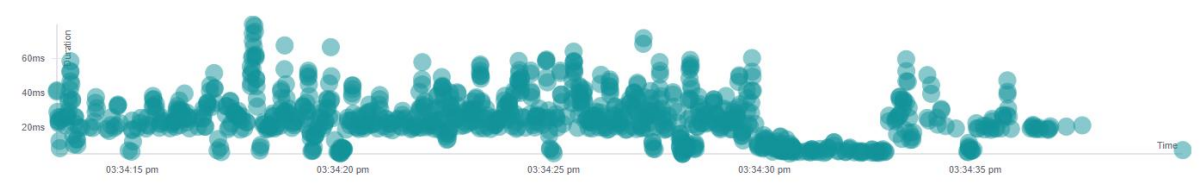

Figure 4.14: Jaeger Graph for the 50%-50% scenario

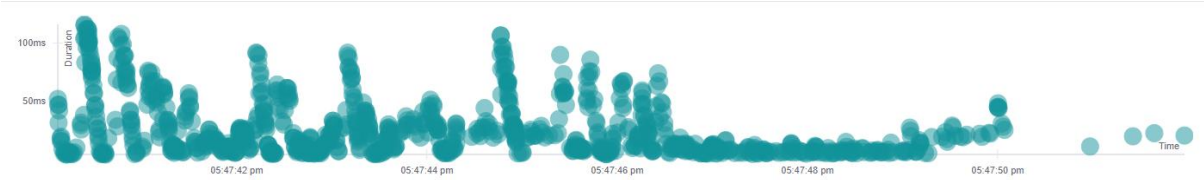

Figure 4.15: Jaeger Graph for the 25%-75% scenario

4.4 Experiment III

In this experiment we tried to determine if having a certain number of users requesting the page of one product or multiple ones affects the microservice in a different way. In scenarios like ours where the microservice applications are shops or websites, usually the users will be spread in different pages but in case a product or a page is trending, it might see a large number of users requesting it.

The average latency between the 2 scenarios was significantly different. At the first scenario, where the users were requesting only a product, the average latency was 3 seconds and on the second scenario where the users were split in different products it was only 1.7 seconds.

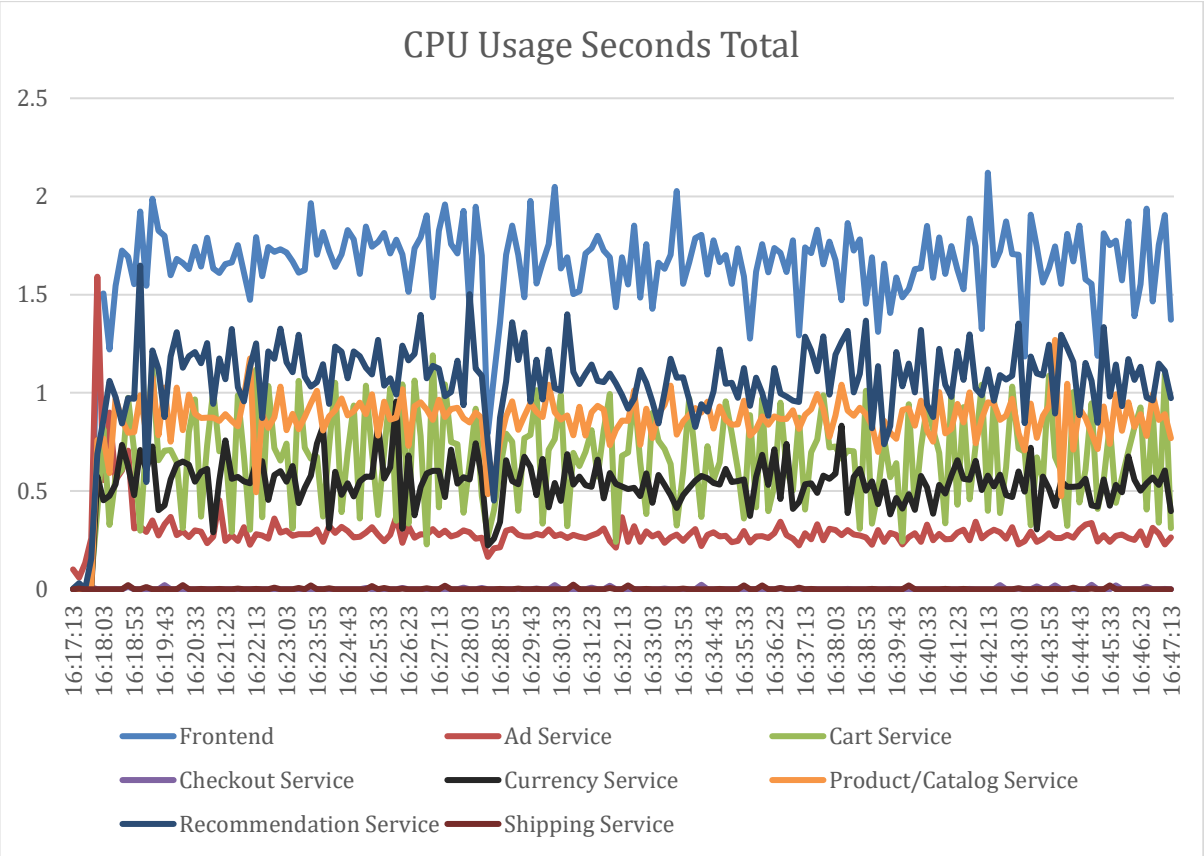

Figure 4.16: CPU usage for the requests at one product

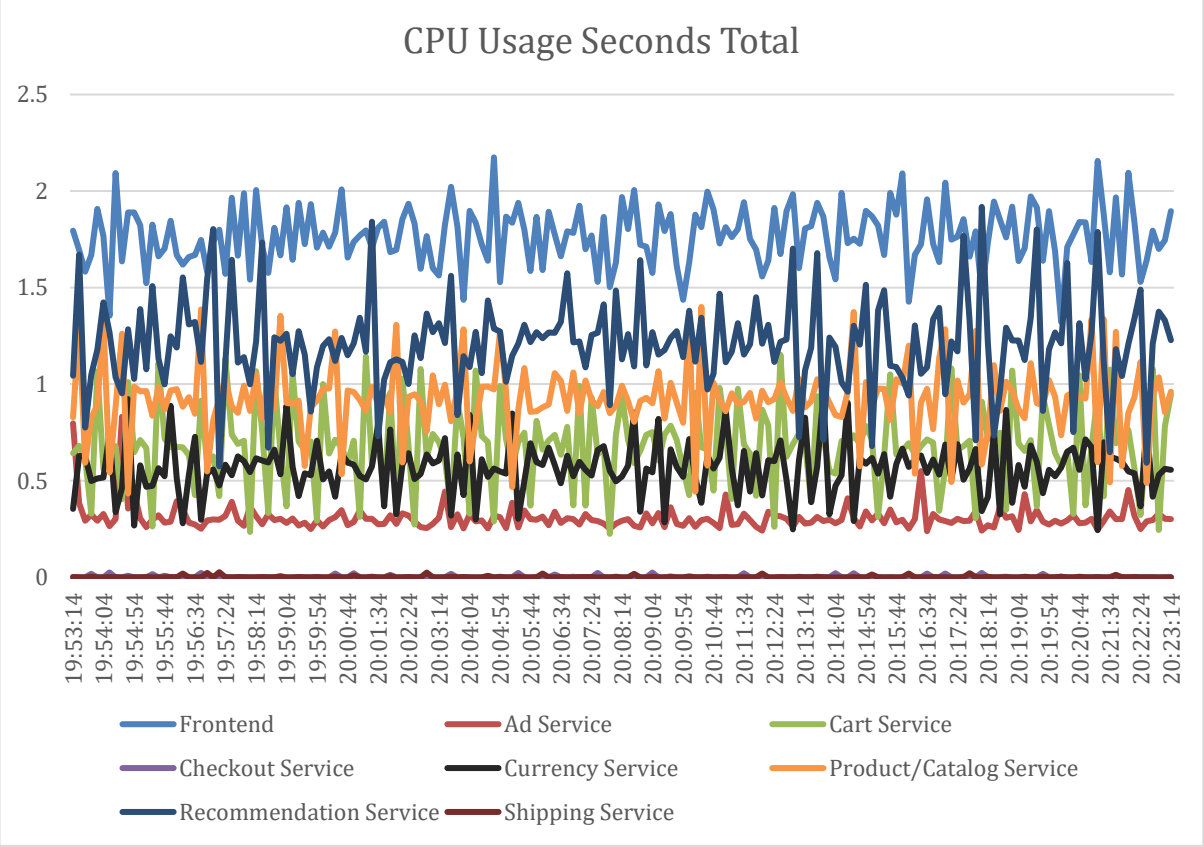

Figure 4.17: CPU usage for the requests at multiple product

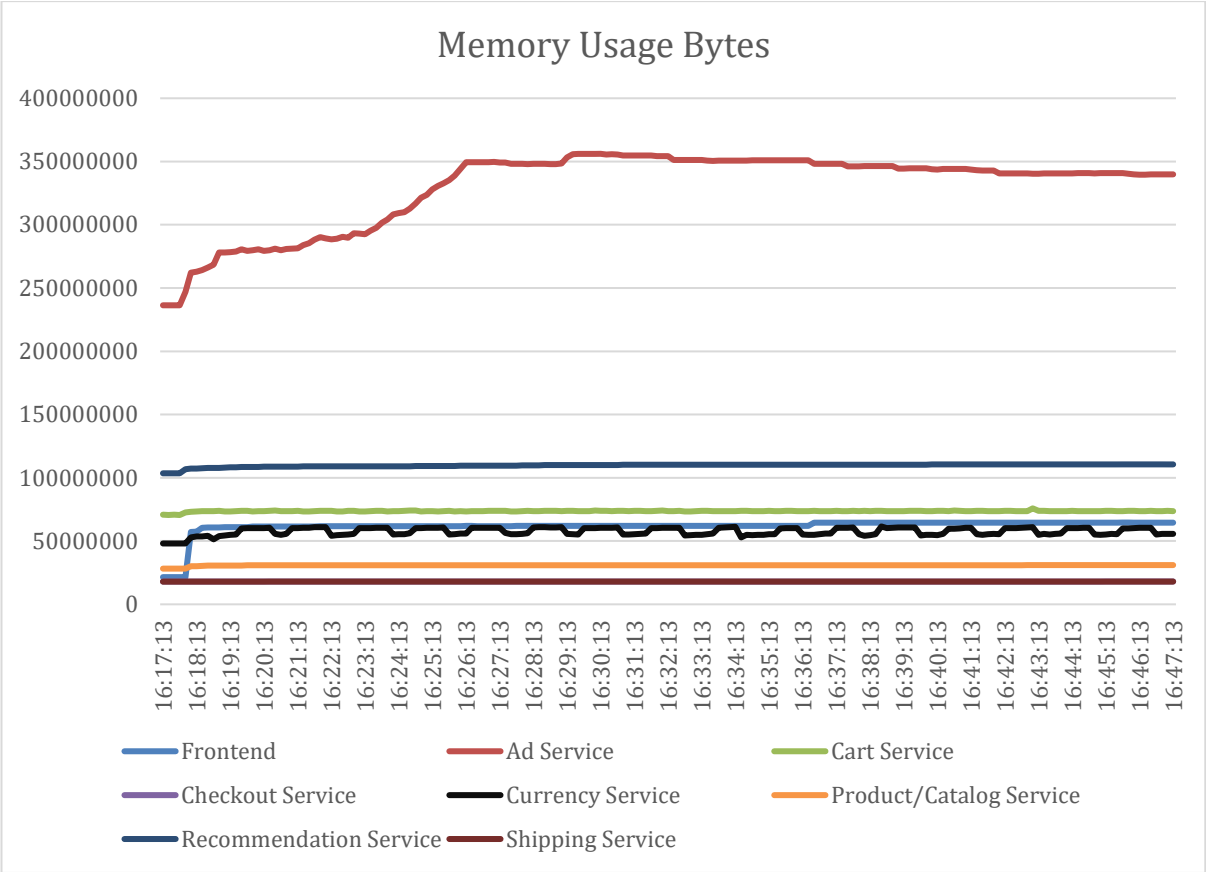

Figure 4.18: Memory usage for the requests at one product

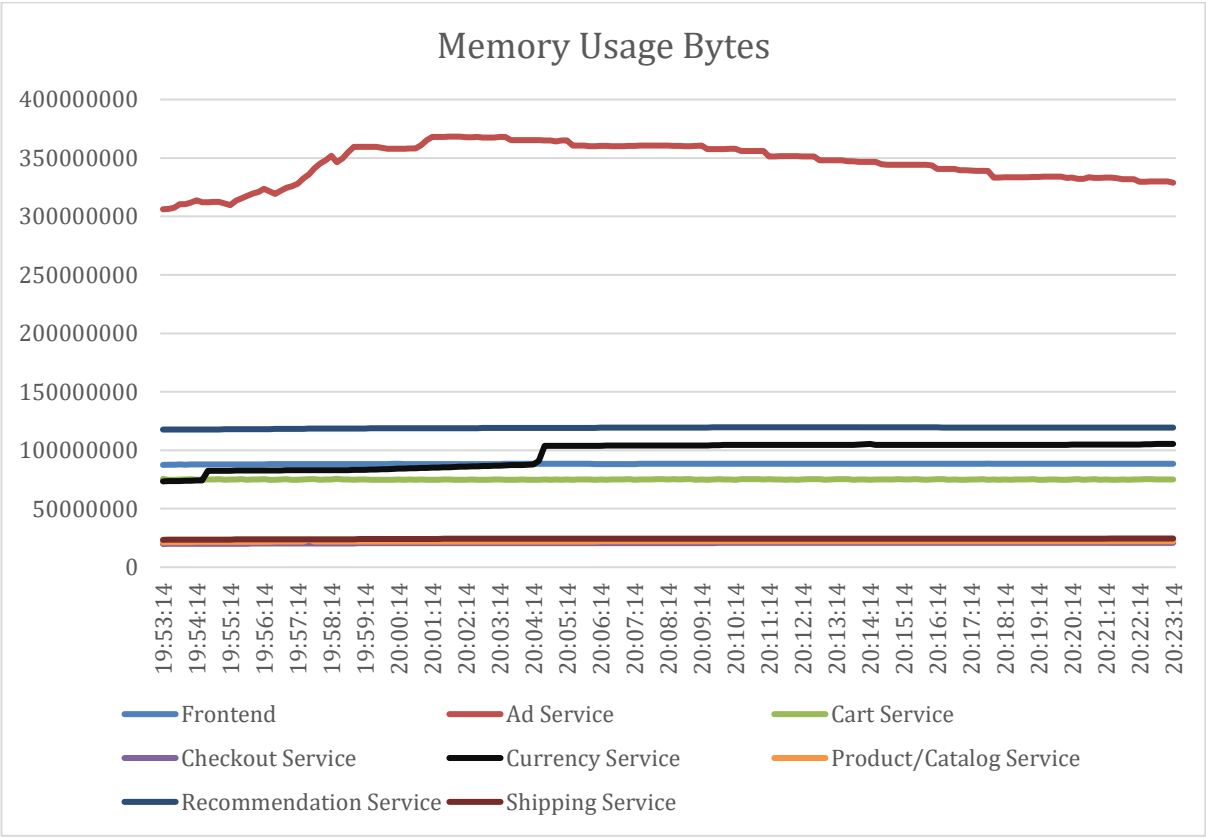

Figure 4.19: Memory usage for the requests at multiple product

From the Figure 4.16 and 4.17 where we can observe the graphs for the CPU usage, we can see that both graphs are almost identical and there are not big differences to notice a change of behavior in the microservice application. In the Figures 4.18 and 4.19 all the services have identical memory usage except Currency service. When the requests were only for one product, the memory usage for the Currency service was fluctuating between the same values throughout the duration of the experiment, but on the scenario where users were requesting multiple products it increased two times and then became stable.

The results show that it does not affect the microservice application what product the users request, since the CPU usage graphs, and the memory usage graphs are almost similar. The only big difference that could be noted was the way the Currency service behaved in the memory usage graphs and the better latency in the scenario where the users were split in multiple products.

4.5 Experiment IV

During the previous experiment as we can see in Figure 4.20, we noticed through Jaeger that one of the services was taking a lot of time in the completion of the request, it was the Recommendation service. For testing purposes, we limited the CPU usage of that service at 30% of the available processing time to see how it affects the processing times and the rest of the services. After that we repeated the previous experiment.

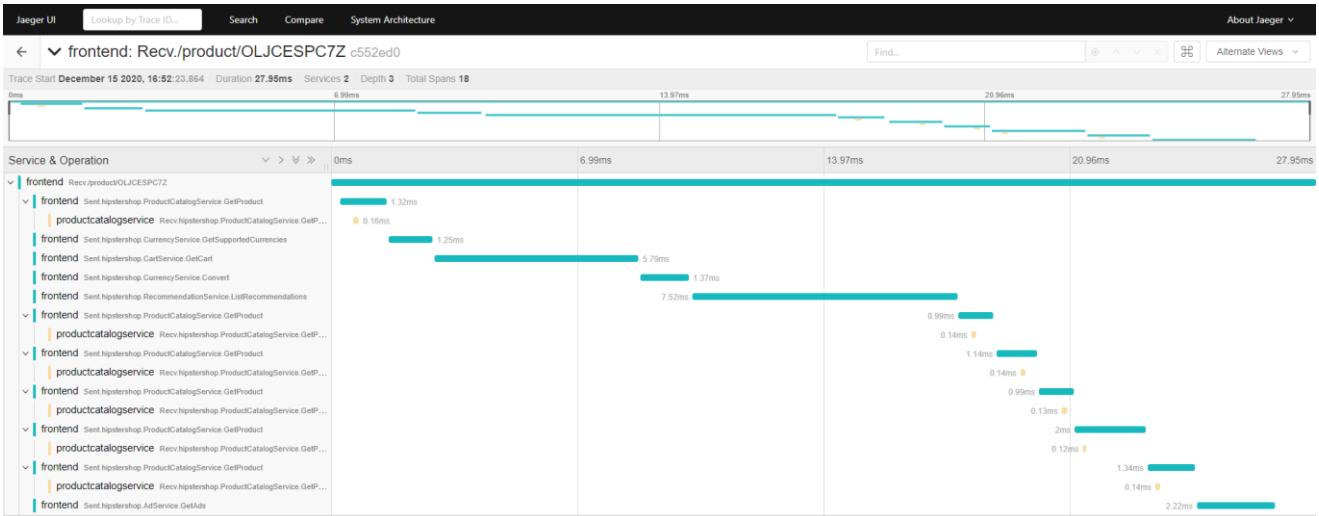

Figure 4.20: Request on a product in the Jaeger

The latency for the scenario where the users are requesting one product was 2.3 seconds and for the multiple products 1.2 seconds. The big difference between the two scenarios remains even with a CPU limit in one of the services.

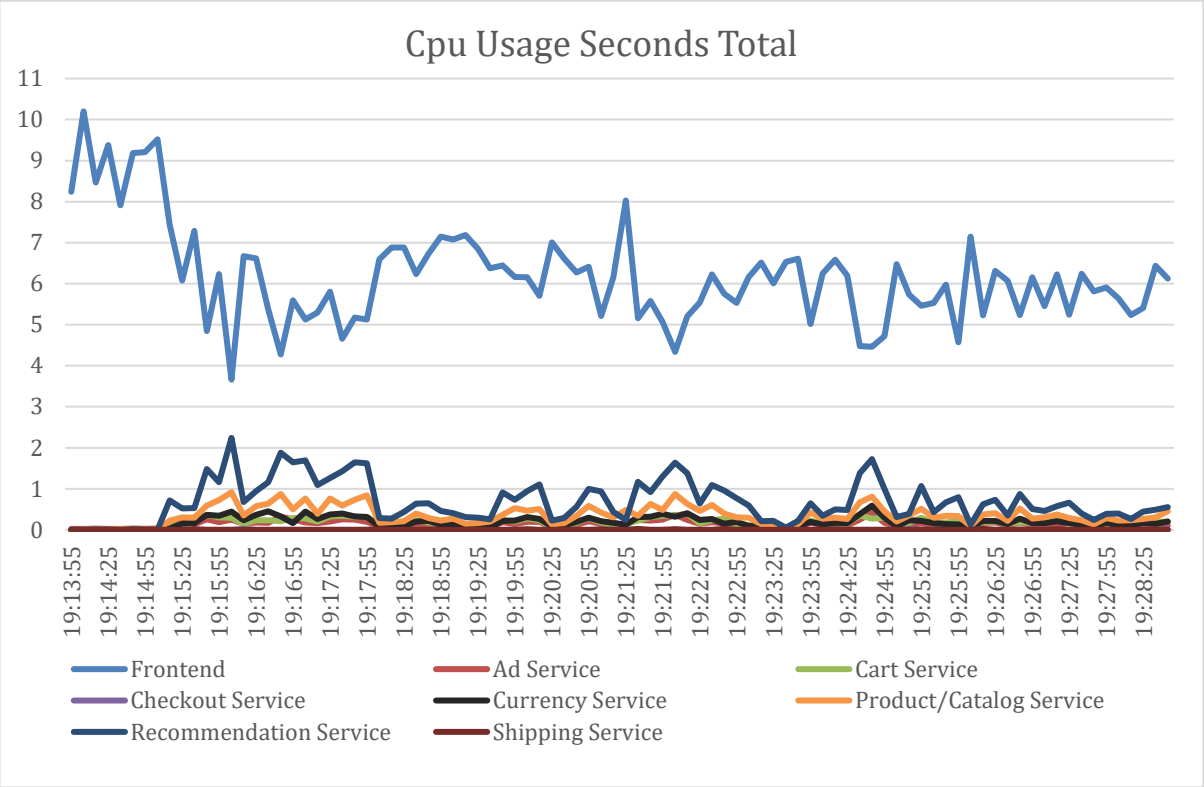

Figure 4.21: CPU usage for the requests at one product (IV)

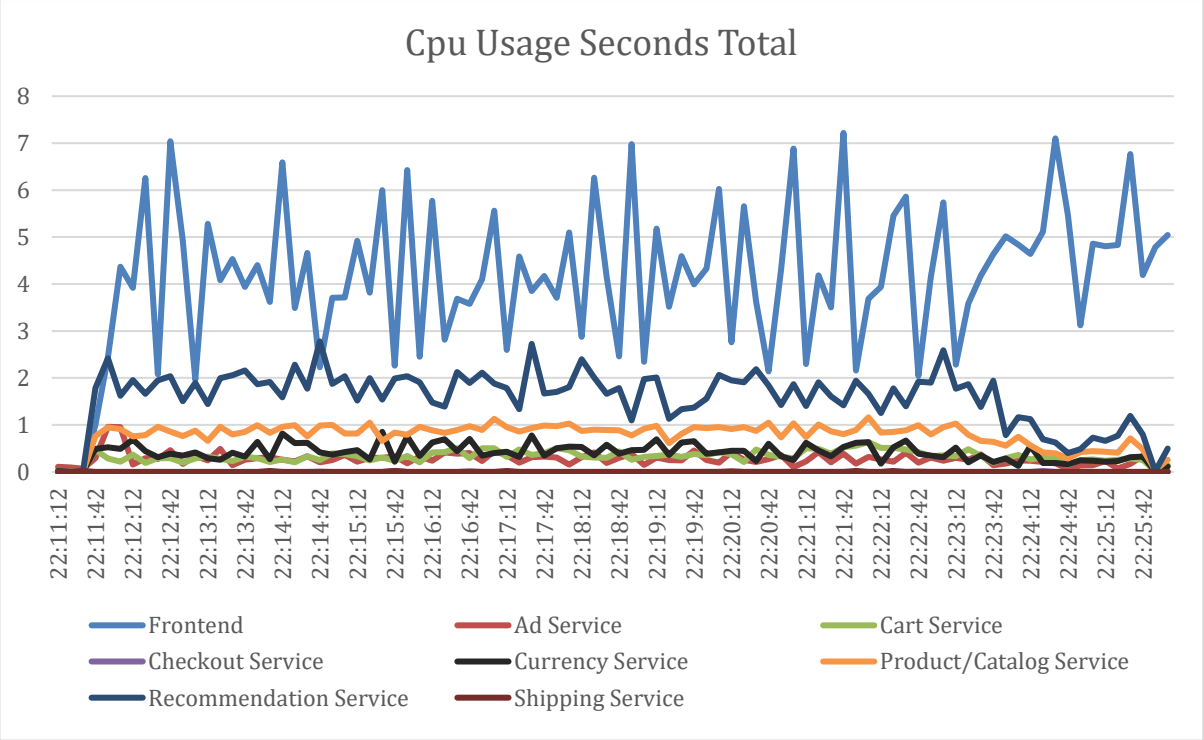

Figure 4.22: CPU usage for the requests at multiple product (IV)

As we can see from the above Figures the seconds that the Frontend is using the CPU have increased drastically. In the previous experiment, the CPU usage from the Frontend was around 2 seconds and now is well above that, reaching the 10 seconds in the scenario with the one product. Additionally, we can observe that in the scenario with the one product all the other services, except Frontend, start from almost 0 seconds and only the Recommendation service has more usage than 1 second. The same things happen in the scenario with multiple products with the main differences being that the fluctuations are smaller and the highest number the Frontend reaches are the 7 seconds.

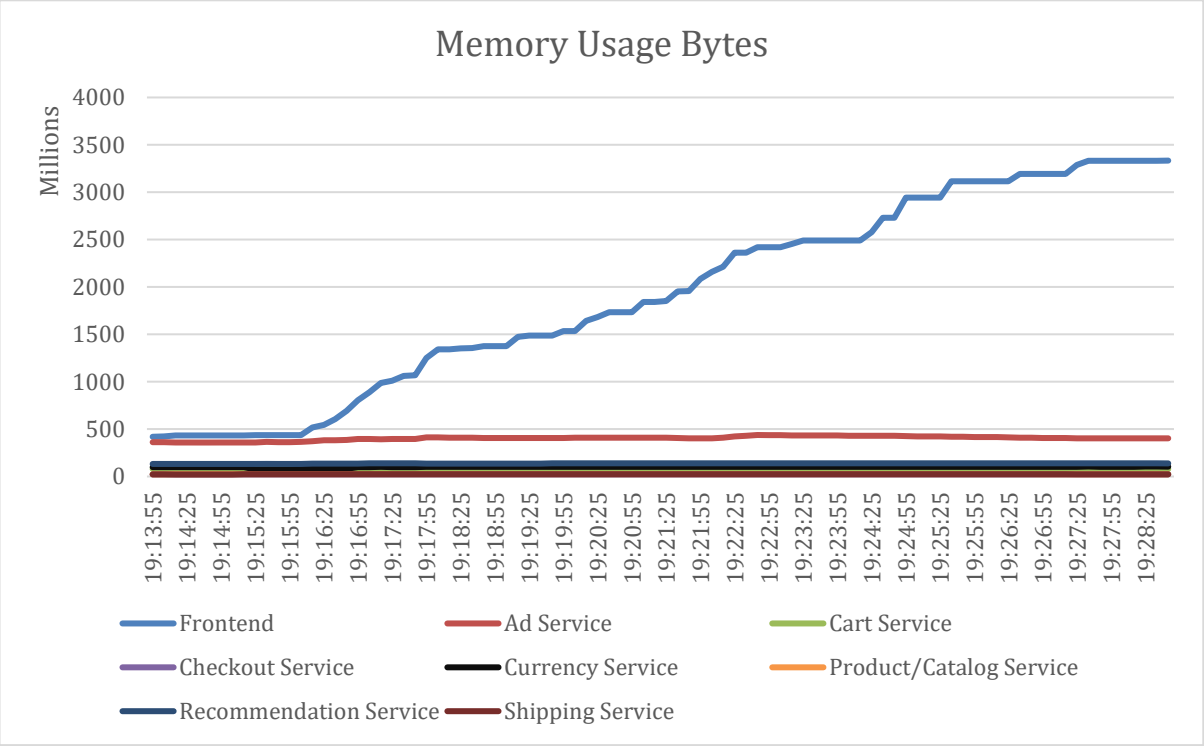

Figure 4.23: Memory usage for the requests at one product (IV)

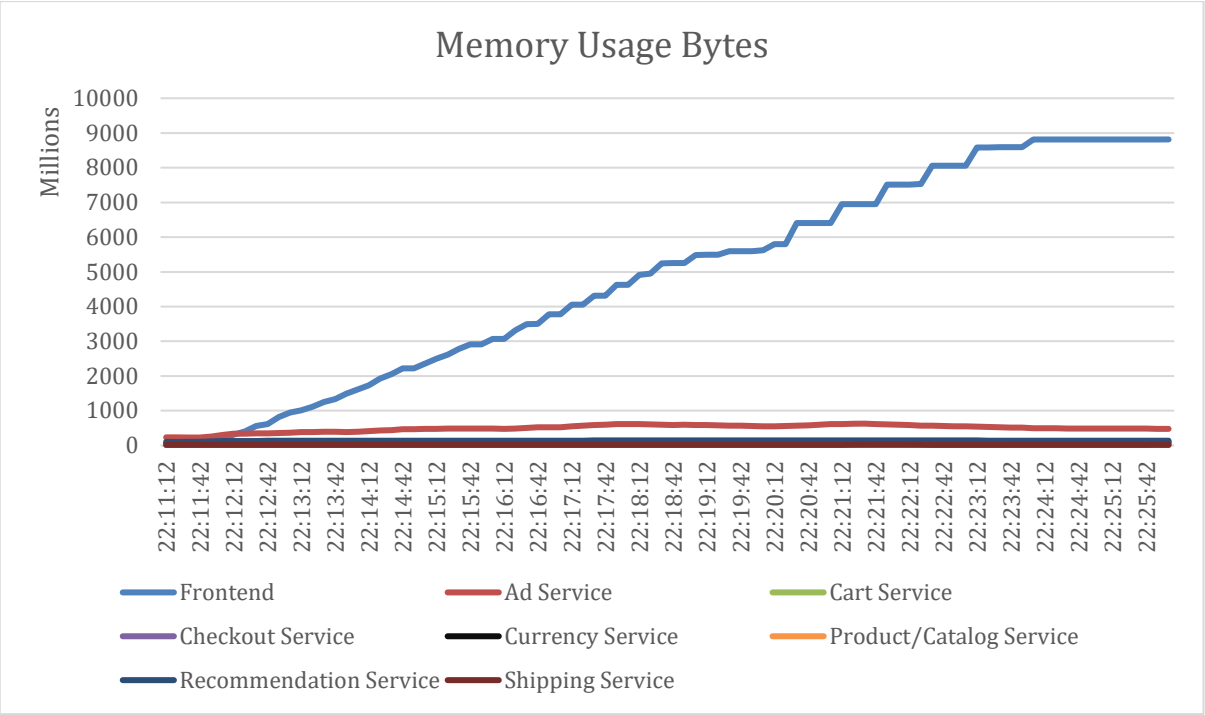

Figure 4.24: Memory usage for the requests at multiple product (IV)

In the Figures 4.23 and 4.24 we can see that the memory usage of the Frontend has reached the thousands of millions of bytes and at the scenario with the multiple products it scales up to 9GB. The numbers that the Frontend service reaches, make the memory usage of the other services seem very small. In both scenarios, the memory usage by the Frontend grows fast and it does not fluctuate, as we have seen happen to other services in other experiments.

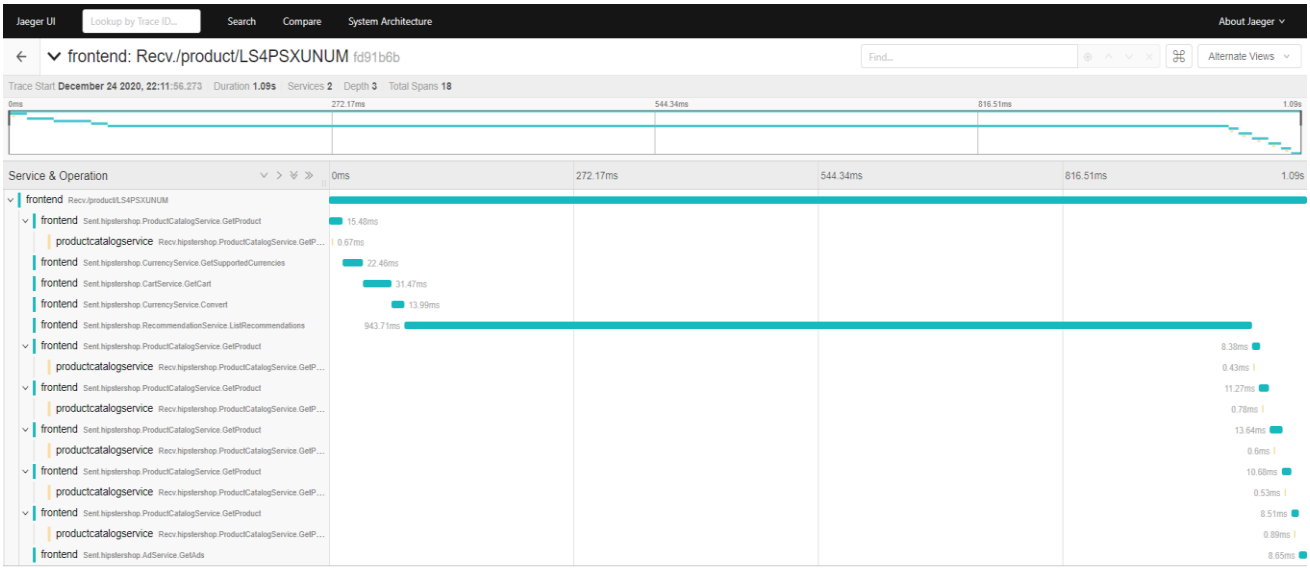

Figure 4.25: Request after the CPU limit

As we can observe from the CPU and memory graphs limiting the CPU time of a service affects all the other services but mainly the Frontend. From Figure 4.25 we can see that the requests took more than 1 second when it usually takes less than 100 milliseconds. By limiting the resources of a service that is used in the requests we caused the application to need even more resources.

4.6 Experiment V

In the next experiment we added limits to the resources of the Frontend to see how it affects the application. We limited the CPU usage to 100% of one single core and 7GB maximum memory usage since in the previous experiment it almost reached 9GB. We created a small number of users that were requesting the cart page, a product page, and the main page for 5 minutes each.

The average latency of all the requests was 0.64 seconds which is probably due to the lower number of users since only 300 were active.

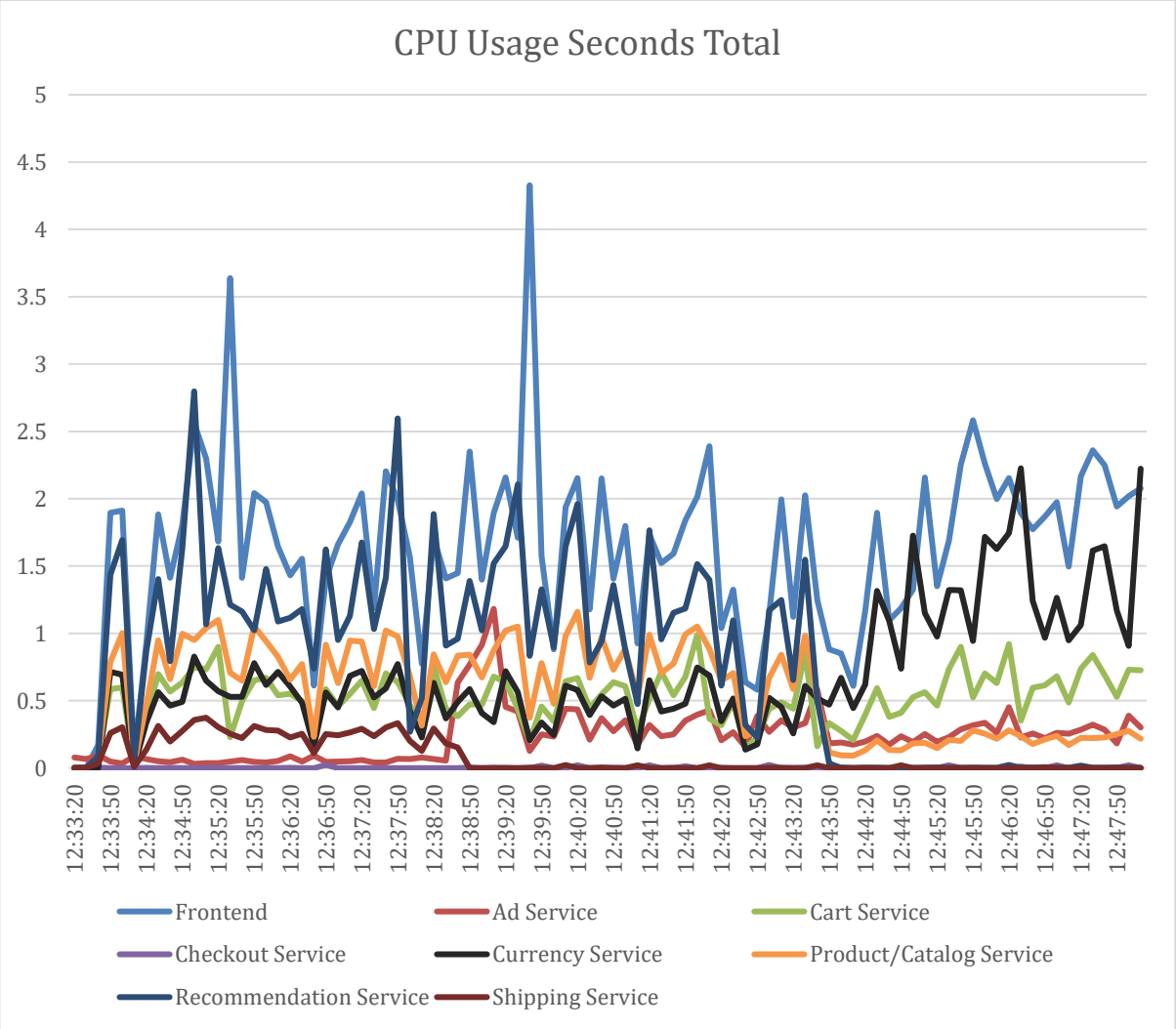

Figure 4.26: CPU usage for Experiment V

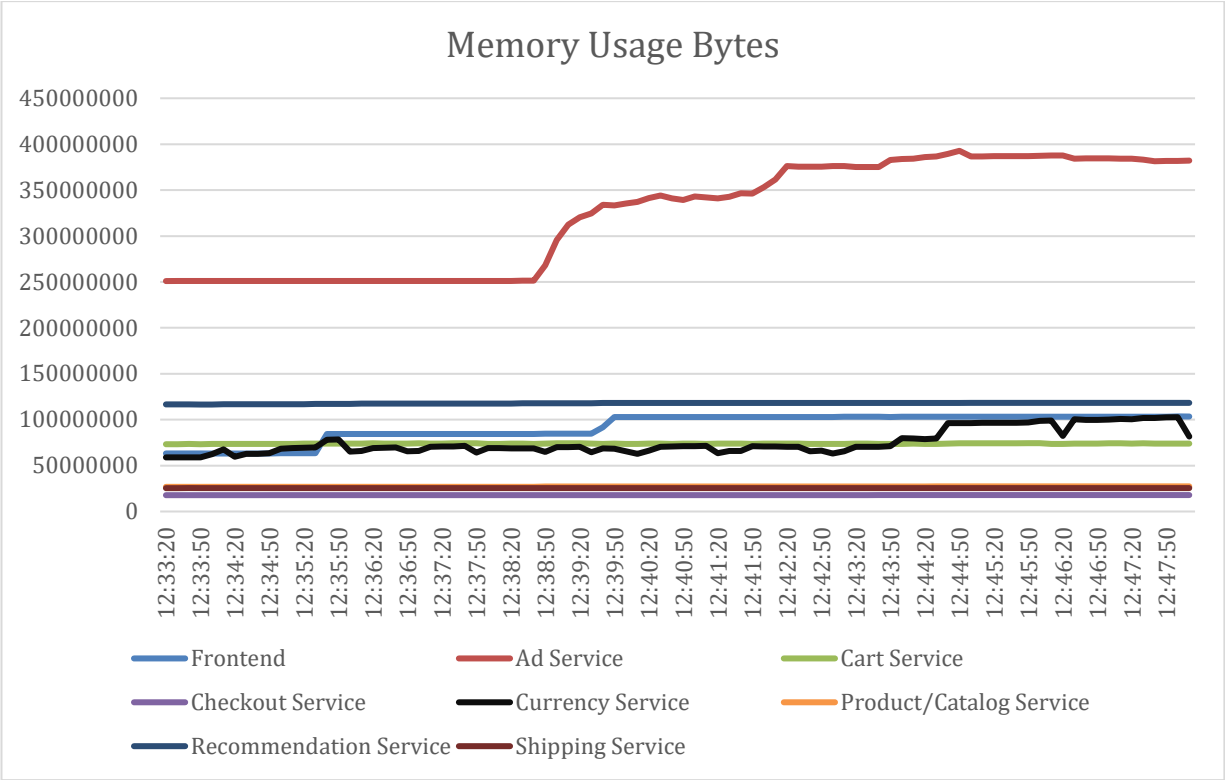

Figure 4.27: Memory usage for Experiment V

In the first 5 minutes of the graphs at Figures 4.26 and 4.27 we can observe the CPU and memory usage when the users were requesting the cart page. The following 5 minutes the requests were targeting a product and the last 5 the main page. From the aforementioned graphs, we can understand that the Ad service is not needed for the cart page and when it is part of the request the CPU and memory usage increase. Every service has increased CPU usage compared to the Figure 4.1, which shows the idle usage. The memory usage of most services is close to the idle except the Ad service that has a massive increase, the Frontend that increased slightly and the Currency service that fluctuates.

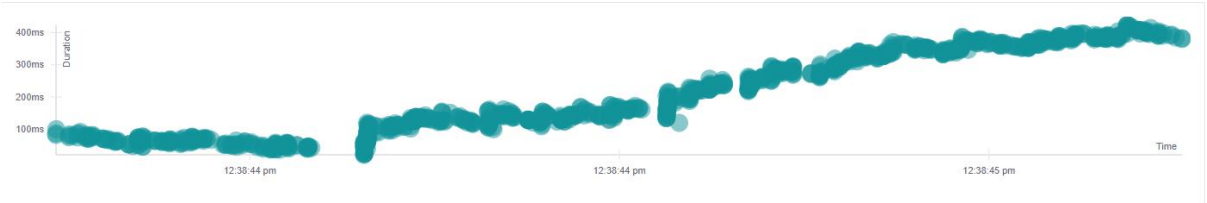

Figure 4.28: Jaeger Graph for the cart requests

Figure 4.29: Jaeger Graph for the product requests

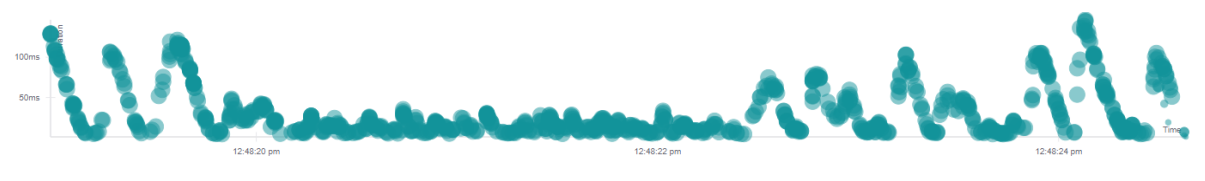

Figure 4.30: Jaeger Graph for the main page requests

From the above Figures we can see the time needed for a request to be completed, is a lot higher for the Cart requests since it reaches 400ms and it seems to be the one affected the most from the limits set to the Frontend. The requests on the product and the main page go above 100ms but they are mostly under 100ms. The affect from the limits on the Frontend was not as bad as the Experiment where the limit was on the Recommendation service.

4.7 Experiment VI

In the following Experiment we wanted to see how the application responds when the users increase and decrease after a certain period. We created a constant load of 100 users requesting the main page for all the duration of the experiment and every two minutes until the sixth minute, we were adding 100 users who were requesting a product. After six minutes from the time the 100 users began their requests for the product, we stopped them. We created the stress test this way, so we can have a smooth increase and decrease in usage.

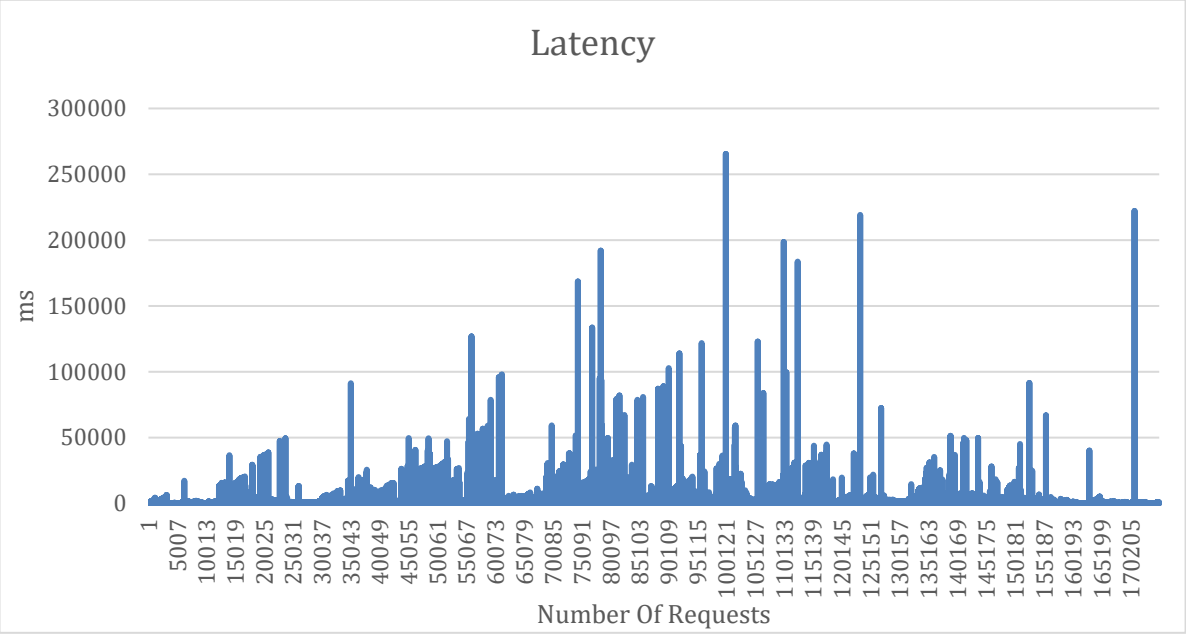

Figure 4.31: Latency Graph for Experiment VI

The average latency of the requests was 0.6 seconds and as we can see from Figure 4.31, after a certain number of requests, we can see an increase in the latency which is due to the new users starting to request the product. We can observe that the same thing happens at the end of the graph, where only the 100 users that request the main page are still

creating requests. There are a few requests that have a very high latency which can be due to the network.

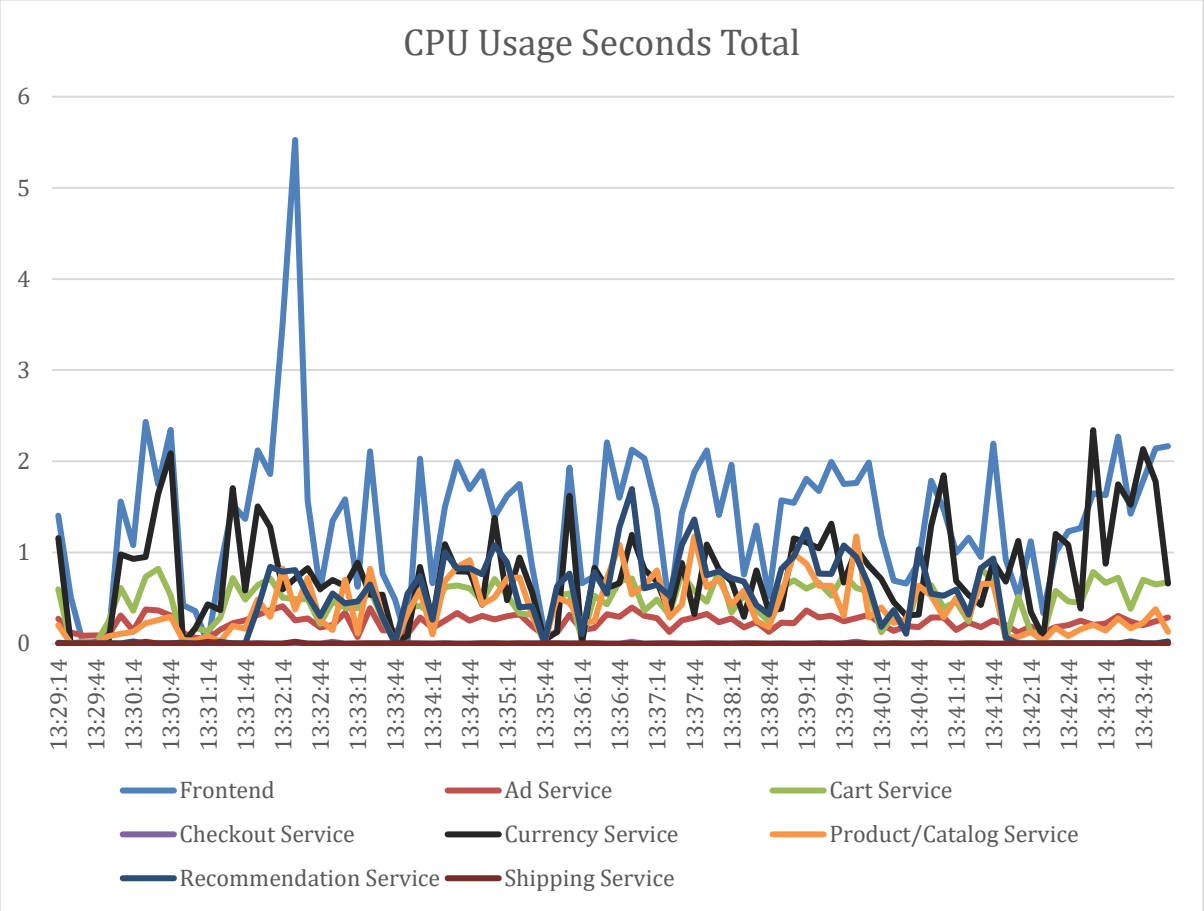

Figure 4.32: CPU usage for Experiment VI

In the Figure 4.32 we can see the CPU usage during this experiment. The most noticeable thing in the Figure is the spike the Frontend has, which is at the time the first 100 users started requesting the product. Through the CPU usage of the Recommendation service and Product/Catalog service, we can observe the time the users started the requests on the product and the time they stopped. Even though we can get the time that the requests for the product start and stop, we cannot find out when the number of users is increased or decreased since the usage of the services keeps fluctuating between the same values. The rest of the services keep fluctuating between the same values from the beginning of the experiment and the increase and decrease of users does not seem to affect them a lot.

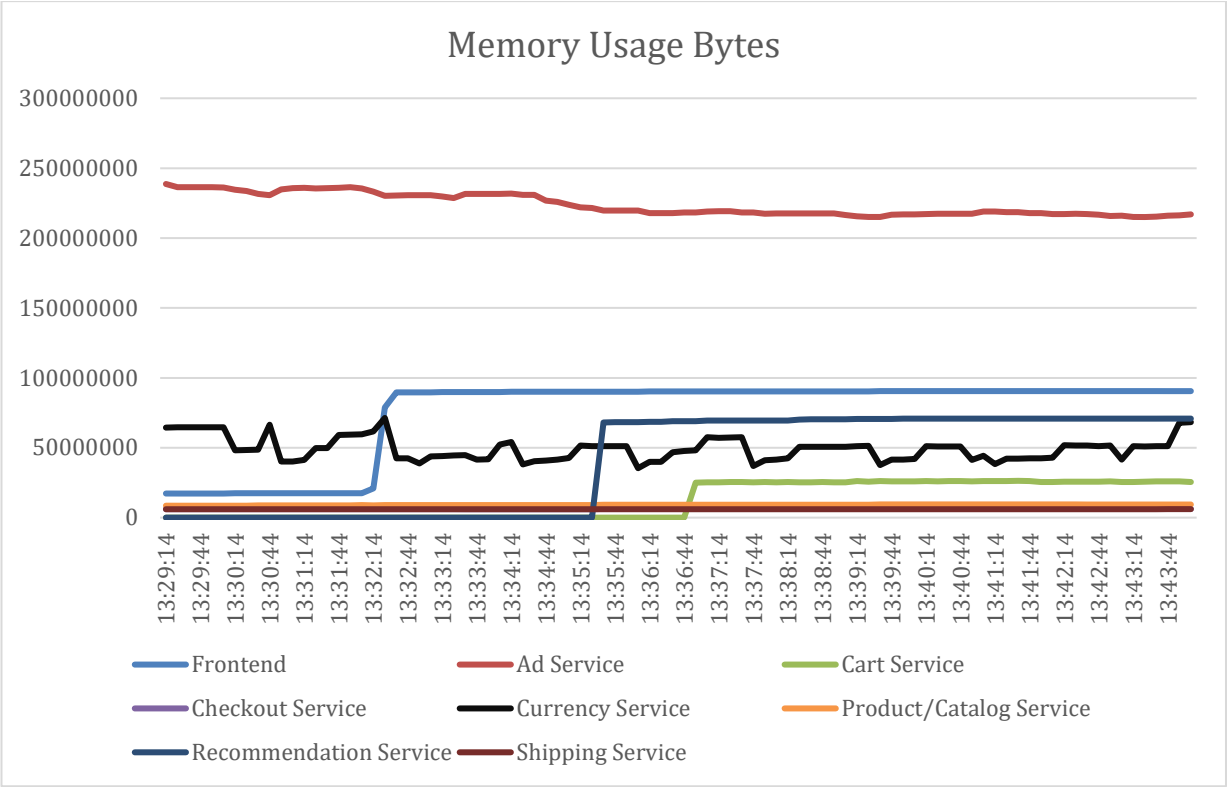

Figure 4.33: Memory usage for Experiment VI

From Figure 4.33 we can observe the memory usage of the services. The memory usage by the Currency service is fluctuating for all the duration of the experiment and the memory usage for the Ad service seems to be slowly decreasing. The behavior of the Frontend and Recommendation service was a bit unexpected. The memory usage of the Frontend was once increased at around 2 minutes after the experiment starts, which is the time the first 100 users start their requests for the product, and after that, it is stable for the rest of the duration of the experiment. The Recommendation service's memory usage is increased a while after the first 100 users when the number of users was increased by 100 more. There was a small increase in the Cart service's memory usage after all 400 users were sending requests. None of the aforementioned services that had an increase in memory usage, had a decrease, after the 300 users that were requesting the product stopped.

4.8 Experiment VII

In the previous Experiment as well as in the Experiment III we observed that the Recommendation service was one of the most time-consuming services. In some cases, as we can see in the Figure 4.34, which shows a request at a product from Experiment VI, it needs as much time as all the other services combined. For this reason, in this experiment, we decided to try and scale the Recommendation in 2 containers and see if that affects the performance of the application. After scaling the service, we used the same stress test as the Experiment VI.

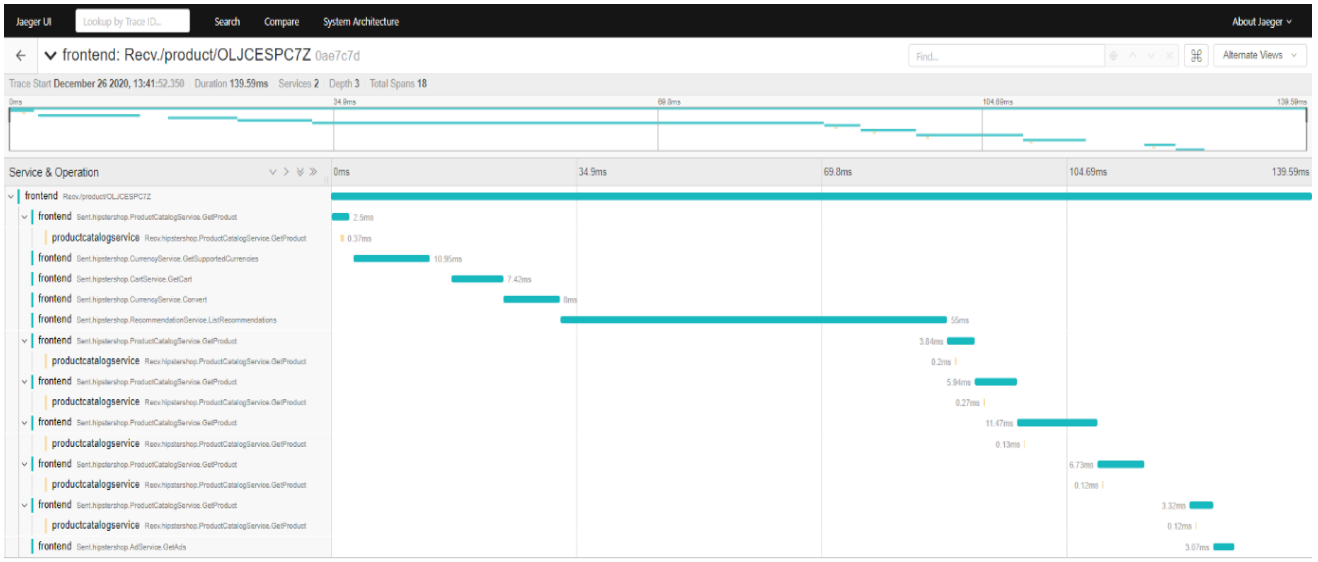

Figure 4.34: Jaeger Graph for the product request from Experiment VI

The average latency was 0.4 seconds and as we can see from the Figure 4.35 the latency of the requests does not increase a lot after a certain number of requests. We can observe small spikes and a very small increase in the latency in the requests 93556 to 280666. The big difference in the latency from this experiment to Experiment VI could be due to a more stable network, since there were less massive spikes in the latency.

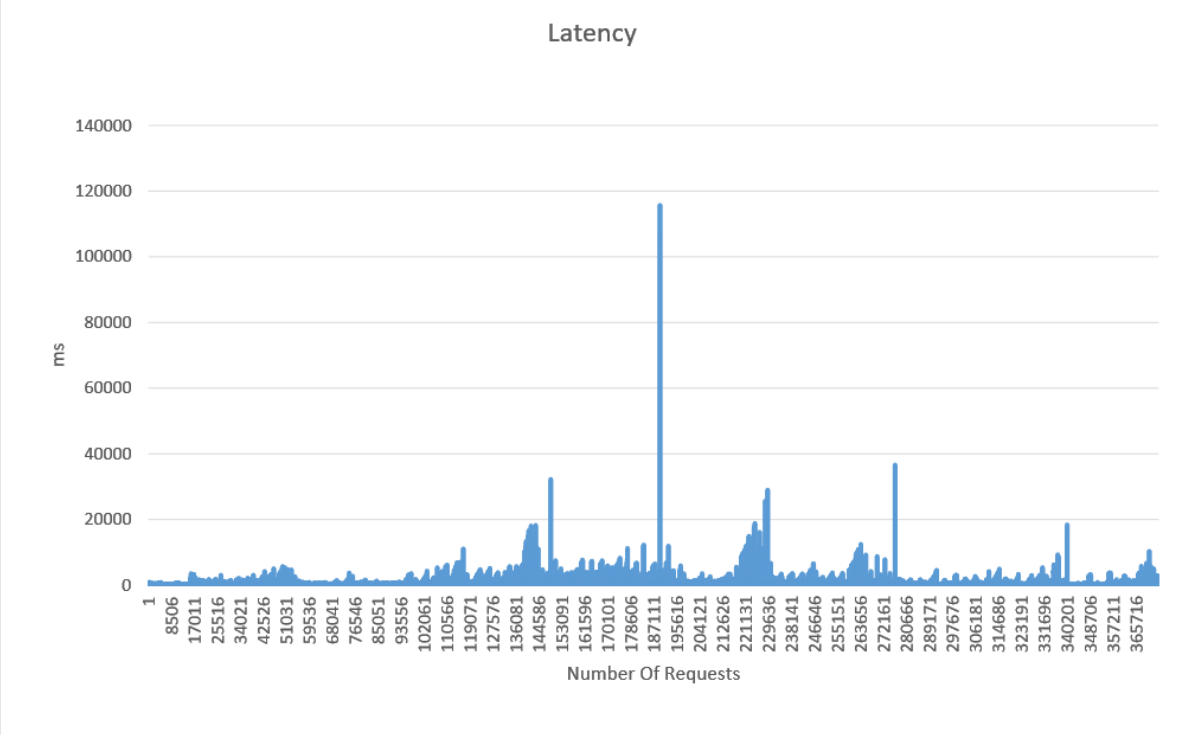

Figure 4.35: Jaeger Graph for the product requests (VII)

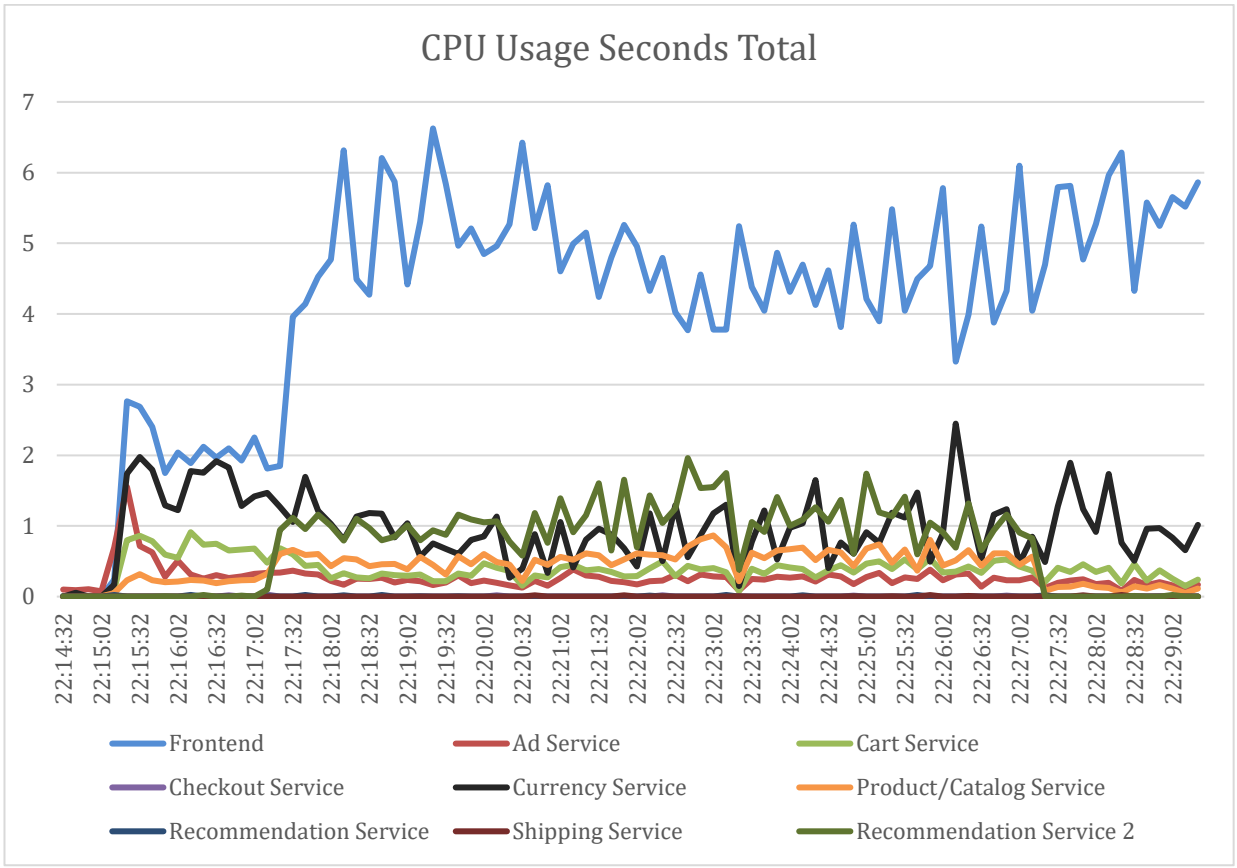

Figure 4.36: CPU usage for Experiment VII

In the Figure 4.36 we can observe the CPU usage of the services during the experiment. The Recommendation service and the Product/Catalog service have the exact same behavior as the preceding experiment, the memory usage was increased when the users that were requesting the product started and decreased when they stopped. Additionally, all the other services except Frontend are fluctuating at around the same values and under the 2 seconds. On the other hand, the Frontend at the beginning when the requests target only the main page has a similar behavior but when the requests for the product started it increased drastically. The Frontend's memory usage is fluctuating between 4 and 6 seconds when in Experiment VI was under 2 seconds. Lastly, the application only utilizes 1 of the 2 Recommendation containers.

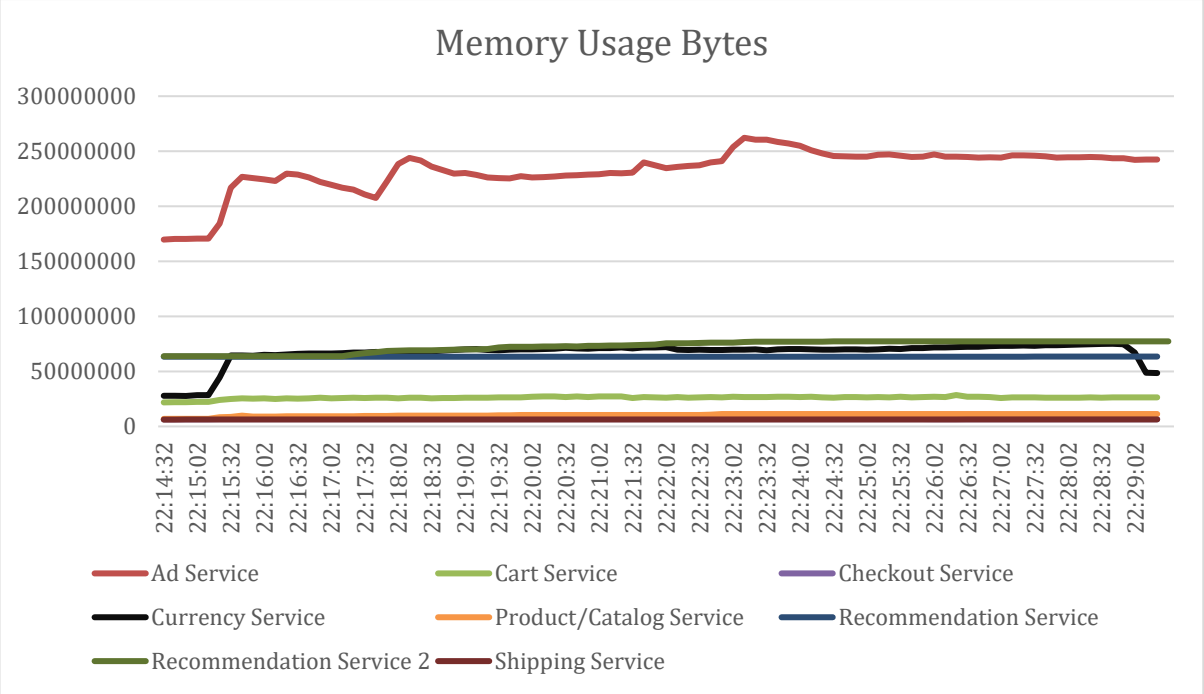

Figure 4.37: Memory usage without Frontend for Experiment VII

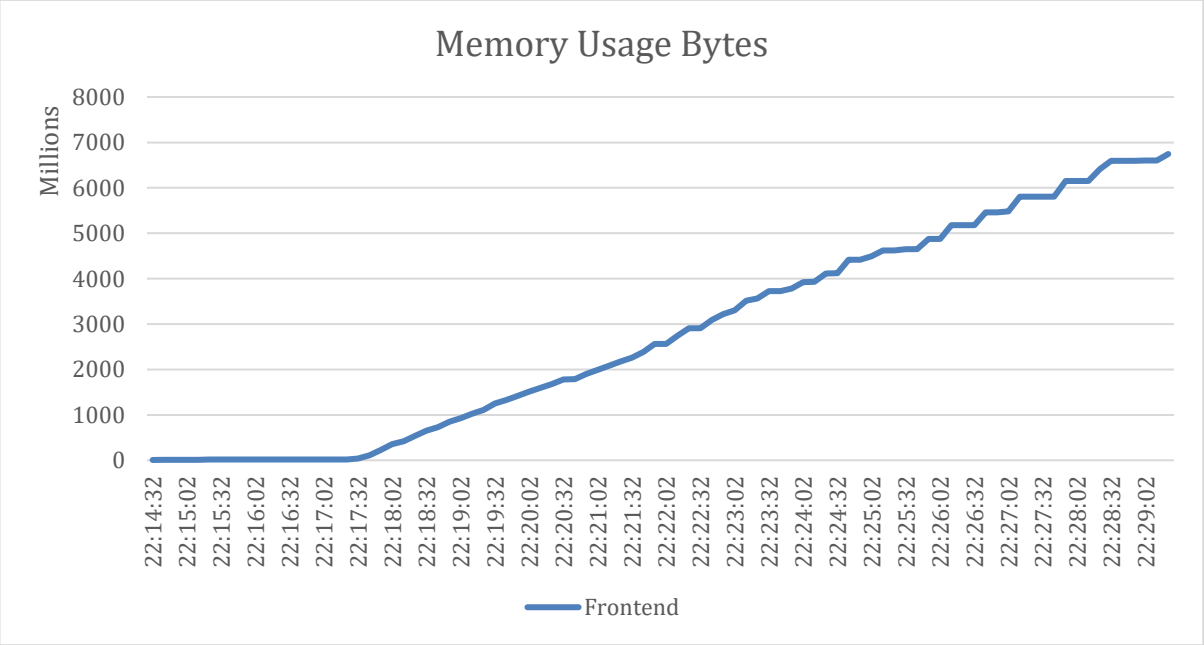

Figure 4.38: Memory usage of Frontend for Experiment VII

In the Figure 4.37 we can see the memory usage of all the services except the Frontend. As we can see from Figure 4.38 Frontend's memory usage kept getting bigger and in the end of the experiment, it almost reached 7GB. Other than the Frontend, the other services behaved normally. From the Figure 4.37 we can see that the memory usage by the Currency service was increased once at the beginning and the Ad service's usage increased by almost 100MB. Both Recommendation services start from the same memory usage but the Recommendation service 2 is increasing slightly until the end. The other services have a stable memory usage from the beginning of the experiment to the end.

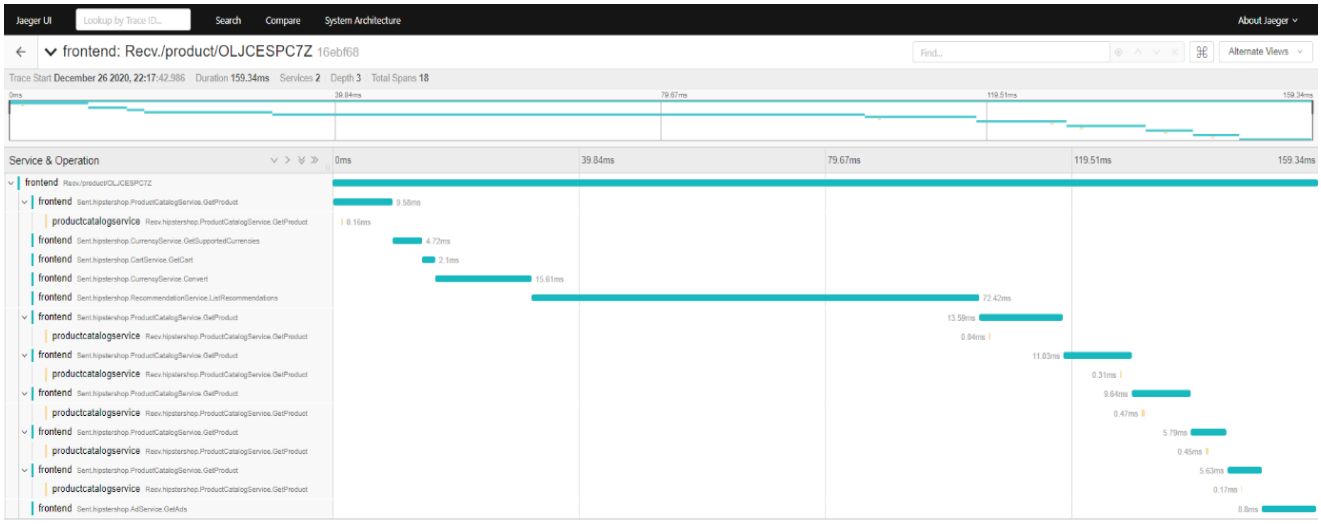

Figure 4.39: Jaeger Graph for the product request from Experiment VII

The results from the CPU and memory usage graphs as well as the request in the Figure 4.39, show that the scaling of the Recommendation service did not benefit the performance of the application. The performance of the application became worse since Frontend requires more time in CPU and higher memory usage. The reason that this is happening is due to the absence of a load balancer from the application.

4.9 Experiment VIII

In this experiment we wanted to simulate a real-life scenario, where a large number of users are requesting a product and after their request they move to the cart. We started with 600 users that were requesting a product and every two minutes we decreased them by 150 and increased the users that were requesting the cart page by 150. Through this experiment we want to see how the application will handle the users moving from one service to another.

The average latency was 1 second and as we can see in Figure 4.40 it had a lot of spikes at the beginning and at the end the last thousands of requests had less and very small spikes.

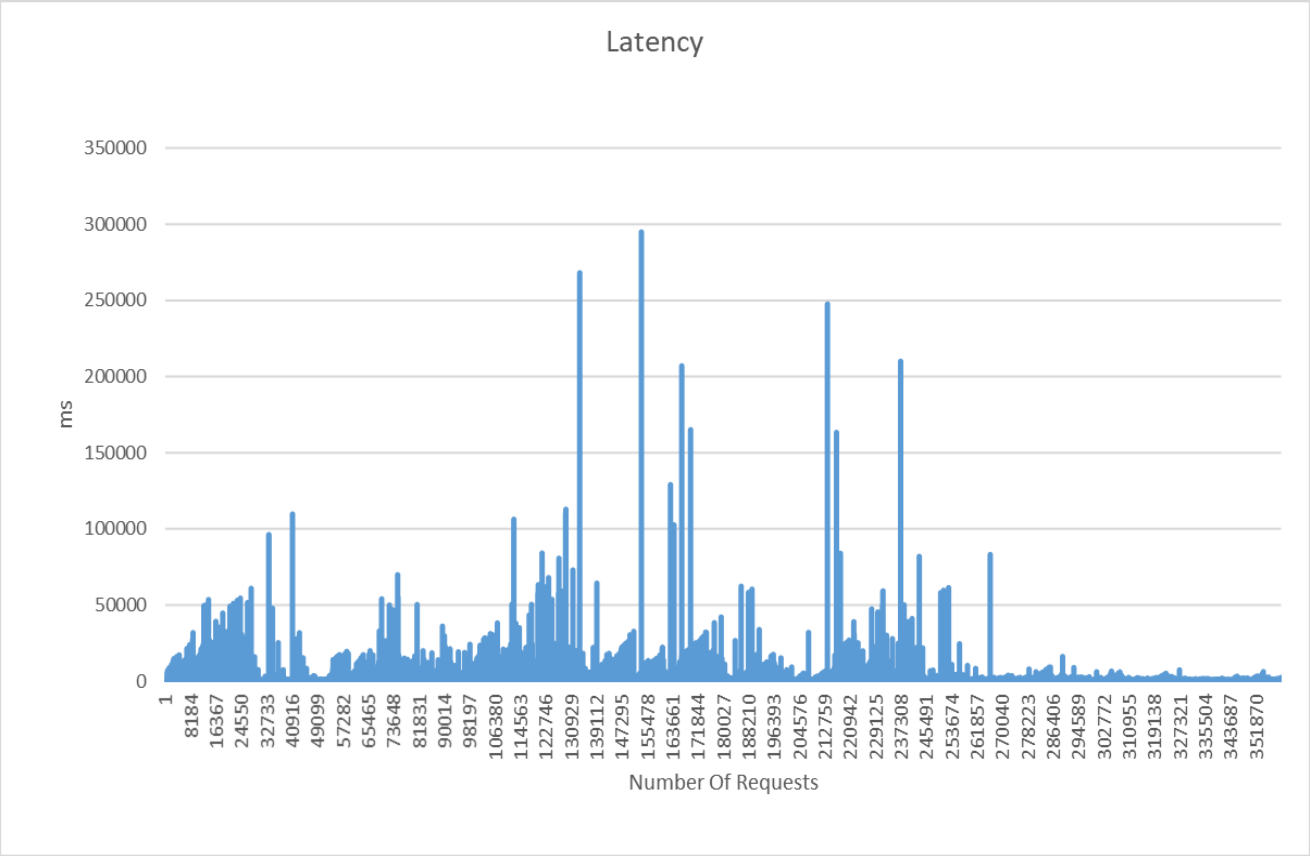

Figure 4.40: Latency Graph for Experiment VIII

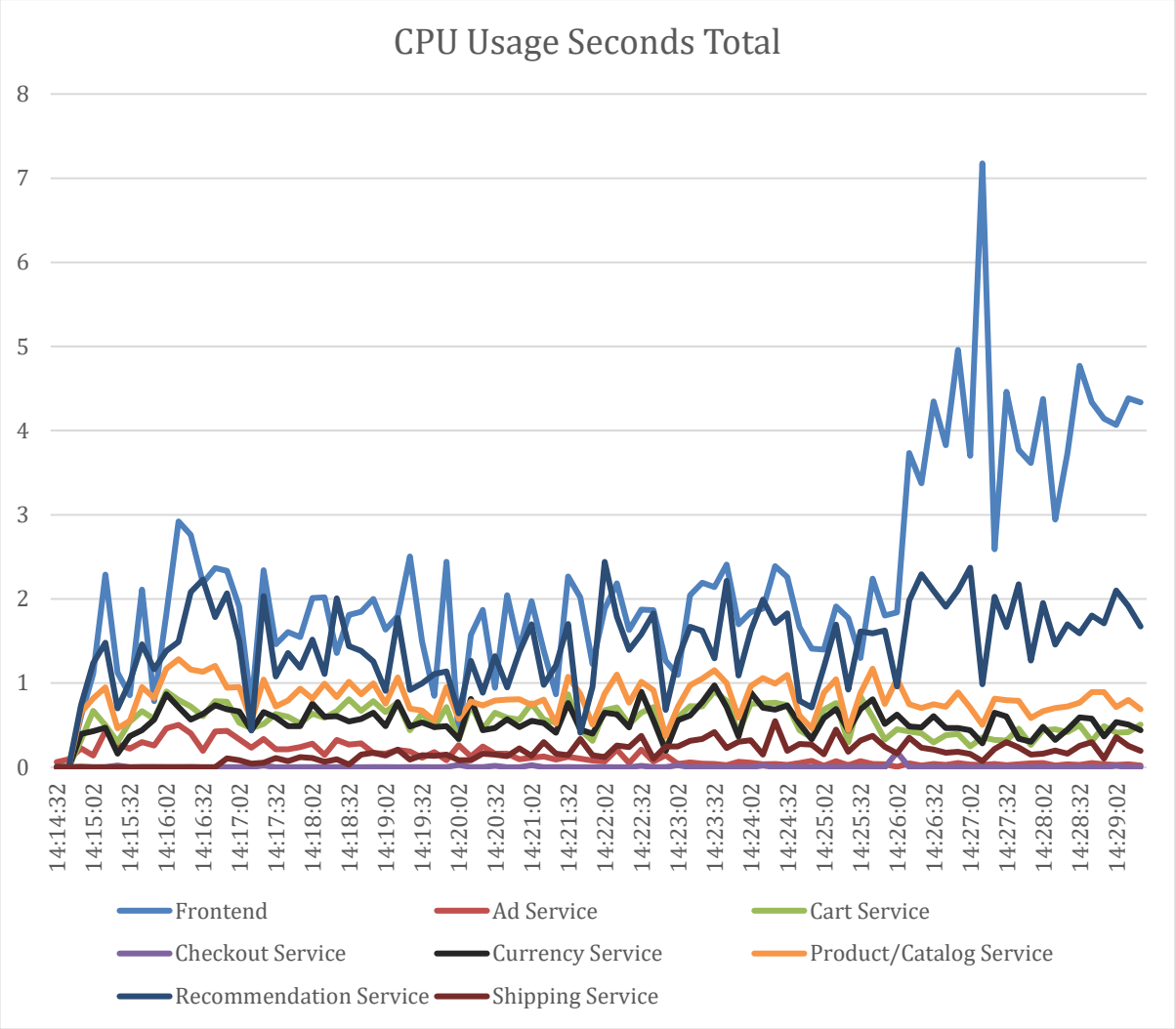

Figure 4.41: CPU usage for Experiment VIII

As we can see from the above Figure most of the services are not affected by the users moving from the product page to the cart page. We can notice small changes that show us the decrease of the users who were requesting a product and the increase of the users requesting the page. The CPU usage of the Shipping service sees a small increase after 2 minutes and the CPU usage of the Ad service is decreasing until it becomes almost 0 seconds. There is a small increase in Frontend's time in the CPU by more than 2 seconds the last few minutes but there is not a clear indication on what affected the change, since the users requesting the product were reduced to 0 at 14:22:32 and there were not any changes in the requests for a few minutes.

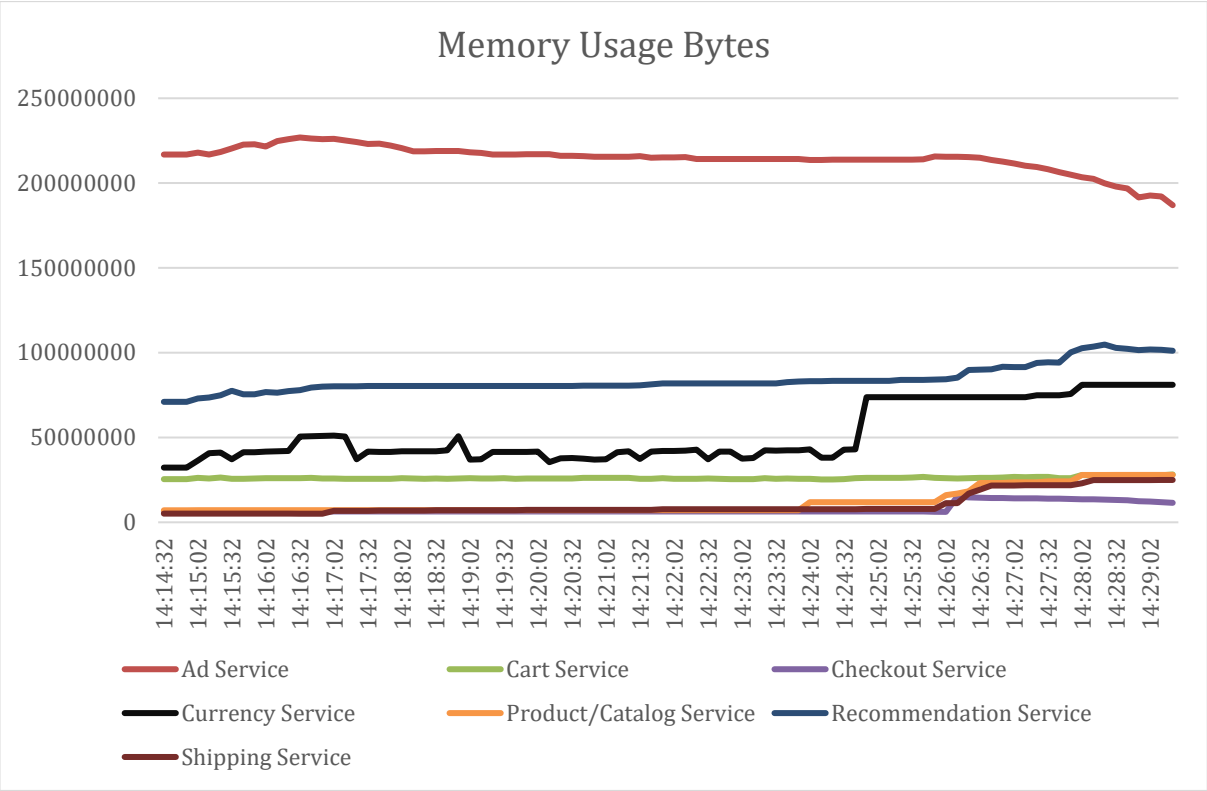

Figure 4.42: Memory usage without the Frontend for Experiment VIII

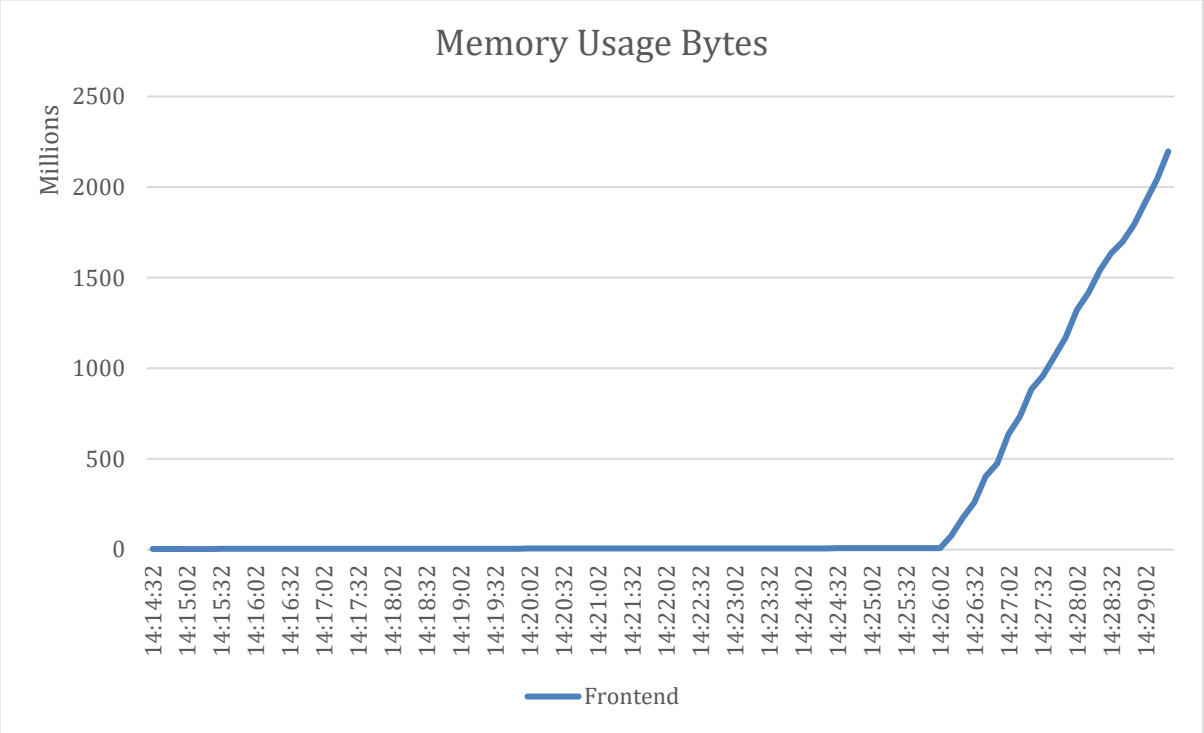

Figure 4.43: Memory usage of the Frontend for Experiment VIII

From the above Figures we can see that the memory usage from the Frontend had an abrupt change at the same minute as the CPU usage. The Currency service's memory usage fluctuates and at the end it sees a small increase and becomes stable. The Shipping, Recommendation, Product/Catalog and Checkout service's memory usage is slowly increasing, and the memory of the Ad service is increasing. Most of the services that are seeing an increase of memory usage during the experiment are part of the cart page as the Figure 4.44 shows.

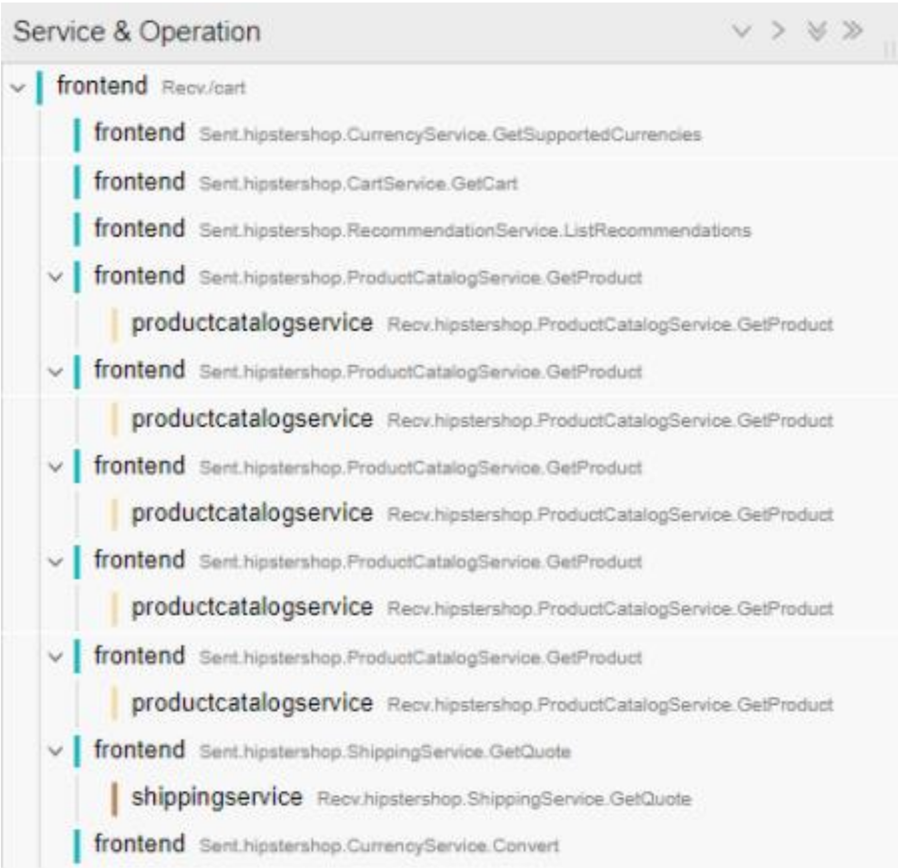

Figure 4.44: Services that the Cart page uses taken from Jaeger

4.10 Conclusions from Experiments

There are a lot of things to take away from the experiments. Through Experiment II and Experiment VII, we can see that a load balancer in a microservice application can benefit the performance and that through scaling a service in an application, without a load balancer, yields no benefit and it can require more system resources.

Creating realistic scenarios helps us understand the behavior of the application and find problems that may occur. Knowing which service needs the most resources in different scenarios will help with balancing the system and setting the right limits on the services. As we can see from the Experiments IV and V limiting the CPU and memory usage on some services can cause opposite results than expected. In Experiment IV the limits in the Recommendation service affected mostly the Frontend, as the application needed more resources in the end and increased the time needed for a request to be completed. Through the realistic scenarios, we can gain the knowledge needed to avoid a lot of problems and fix a lot of issues that would have occurred after the application was launched.

Lastly, a big part of the Experiments are the latency tests. Launching the microservice in an environment with stable and fast internet is important. Even if you have a perfectly balanced application, having a bad network connection can affect the performance and the user experience depending on the purpose of the microservice application.

Chapter 5

Conclusion

5.1 Conclusion

Currently, there are not any standard ways and guidelines to benchmark the microservice applications. There are a lot of articles and research papers that suggest different ways to benchmark them, but there is not one of them that stands out the best. One of the simplest ways to benchmark your application is through stress testing it, with different scenarios that might occur. The purpose of this research was to benchmark a microservice application by analyzing and monitoring it, during different experiments and understand the way it works and discover possible problems and bottlenecks.

The way mentioned above of benchmarking a microservice application can be easily recreated for any microservice application and by anyone. Benchmarking a microservice application is important and we hope through this research that it will be more understandable and easier for someone to understand the behavior of this microservice and fix any problems.

5.2 Future Work

Benchmarking the microservice applications is a topic with a lot of possibilities. There are a lot of different ways to extend this research. One of the ways to extend the research, is to create an environment with higher tier hardware and a network connection between

the server and the machine creating the requests that will be fast, direct and without another network load. This will help create an experiment where the latency cannot be affected by external factors and having higher tier hardware will create a lot more possibilities for different and more complicated experiments. Another way to extend the research is to recreate the experiments in two different environments, one in a server in a home, and another in a server facility creating the possibility to see if the microservice application will be massively affected and find the limit in which someone can have a microservice application running at home.

Lastly, one of the most important ways to extend the experiment is to recreate it in a fog environment instead of a cloud one. There are a lot of different ways to do this and one of those is with Fogify[3], a framework that lets you emulate a fog computing environment. Comparing the results of the cloud and the fog can lead to unexpected outcomes.

Bibliography

[1] M. Grambow, E. Wittern, and D. Bermbach, “Benchmarking the performance of microservice applications,” *ACM SIGAPP Appl. Comput. Rev.*, vol. 20, no. 3, pp. 20–34, Sep. 2020, doi: 10.1145/3429204.3429206.

[2] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, “The pains and gains of microservices: A Systematic grey literature review,” *J. Syst. Softw.*, vol. 146, pp. 215–232, Dec. 2018, doi: 10.1016/j.jss.2018.09.082.

[3] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, “Fogify : A Fog Computing Emulation Framework,” pp. 1–13, 2020. https://ucy-linc-lab.github.io/fogify/

[4] H. Vural, M. Koyuncu, and S. Guney, “A systematic literature review on microservices,” Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 10409 LNCS, no. February 2020, pp. 203–217, 2017, doi: 10.1007/978-3-319-62407-5_14.

[5] X. Zhou et al., “Benchmarking microservice systems for software engineering research,” in Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, May 2018, pp. 323–324, doi: 10.1145/3183440.3194991.

[6] “Jaeger: open source, end-to-end distributed tracing.” https://www.jaegertracing.io/ (accessed Jan. 02, 2021).

[7] “Empowering App Development for Developers | Docker.” https://www.docker.com/ (accessed Jan. 02, 2021).

[8] “Prometheus - Monitoring system & time series database.” https://prometheus.io/ (accessed Jan. 02, 2021).

[9] “google/cadvisor: Analyzes resource usage and performance characteristics of running containers.” https://github.com/google/cadvisor (accessed Jan. 02, 2021).

[10] “GoogleCloudPlatform/microservices-demo: Sample cloud-native application with 10 microservices showcasing Kubernetes, Istio, gRPC and OpenCensus.” https://github.com/GoogleCloudPlatform/microservices-demo (accessed Jan. 02, 2021).

[11] “gluckzhang/prometheus2csv: Tool to query multiple metrics from a prometheus database through the REST API, and save them into a csv file.” https://github.com/gluckzhang/prometheus2csv (accessed Jan. 02, 2021).

[12] “Advantages (and Disadvantages) of Microservices – Prime TSR.” https://primetsr.com/insights/advantages-of-microservices/ (accessed Jan. 02, 2021).