Final Report - Performance and Elastic Scalability in SockShop

Georgi Ivanov
Technische Universität Berlin
Berlin, Germany
georgi.ivanov@campus.tu-berlin.de

Georgi Kotsev
Technische Universität Berlin
Berlin, Germany
g.kotsev@campus.tu-berlin.de

Kalin Iliev
Technische Universität Berlin
Berlin, Germany
kalin.iliev@campus.tu-berlin.de

Momchil Petrov
Technische Universität Berlin
Berlin, Germany
m.petrov@campus.tu-berlin.de

Krutarth Parwal
Technische Universität Berlin
Berlin, Germany
k.parwal@campus.tu-berlin.de

I. Introduction

In a world defined by constant evolution within the software development landscape, enterprises are strategically shaping their IT systems to be able to adequately handle the rapidly increasing workload. Over the years, different system architectures have proven to be effective - the most prominent examples being the monolithic and the microservices architectures. Each of those brings its own set of advantages and factors to consider. While monolithic architectures simplify development and deployment, microservices often offer more granularity and scalability.

This work examines one of the fundamental principles of the microservice architecture - the ability to adapt to workload. Such flexibility is important to businesses due to the fast-paced flow of data and user interaction nowadays. It involves both scaling up, when there is an increased traffic and scaling down, when there is lower utilization. This way, organizations can ensure optimal performance of their applications, which leads to better user experience, but at the same time they ensure proper resource utilization. Unnecessary running instances of a microservice during low-traffic time ultimately increase the cost of the business, so minimizing these and only adding more instances, when they are actually needed, is an important aspect for IT solution providers to consider.

Specifically, we aim to enhance the elastic scalability of the SockShop microservice system, allowing it to adjust well to varying levels of workload. Our goals include examining the system and the interaction between the components, performing load testing in order to pinpoint the component with the worst performance compared to others (i.e., the bottleneck) and ultimately adjusting the architecture and deployment in a way that it reacts better to fluctuating workload.

II. Measurement data

Following the refactoring approach defined in our Design Document, we started by measuring the system load under our test scenario, which aims to reproduce real user interactions:

1) Loading the initial page
2) User registration - since our SockShop is relatively new to the market and we are performing heavy marketing, we expect the majority of our users to be first time users and thus require a registration.
3) User login
4) Loading the catalogue of socks
5) Selecting one random product and viewing its details
6) Selecting another random product and viewing its details
7) Adding the second product to the cart
8) Loading the catalogue of socks again
9) Selecting another random product and viewing its details
10) Adding it to the cart
11) Going to the cart
12) Performing checkout

Each of our simulated users executes the above actions in this exact order. The distribution of active users over time is displayed in Fig.1. We experience two peaks of 80 and 100 simultaneous active users - one at 10:00 (when clients are waking up and stumbling upon our ads while scrolling) and another one at 20:00 (when users prepare for the next day and realise they ran out of socks) respectively. In the other time slots, we experience moderate traffic of around 20-60 simultaneous active users.

The simulation is executed on a Kubernetes cluster, deployed on the following hardware:

- CPU: 12th Gen Intel i7-12800H, 14 Physical Cores, 20 Logical Processors
- RAM: 32GB DDR5 4800 MHz

As a baseline measurement, the simulation was executed on the above-mentioned hardware. Fig.2 displays the median and average latency the clients are experiencing according to the time schedule.

Furthermore, Fig.3 shows the distribution of the measured latency between the five externally available microservices.

III. Refactoring steps

For the purpose of achieving our objective, which is improving the performance of the SockShop microservice-based

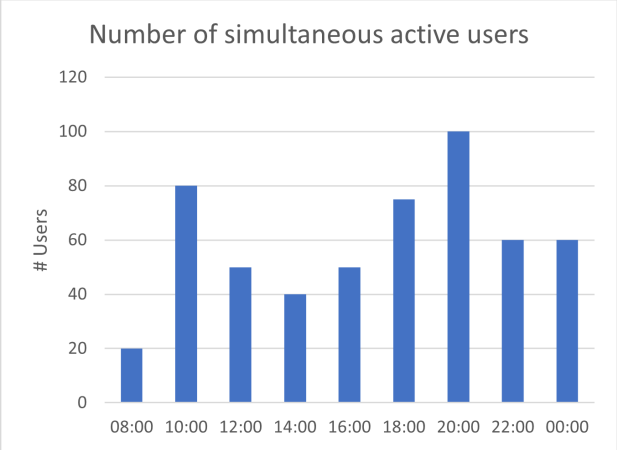

Fig. 1. Number of simultaneous active users

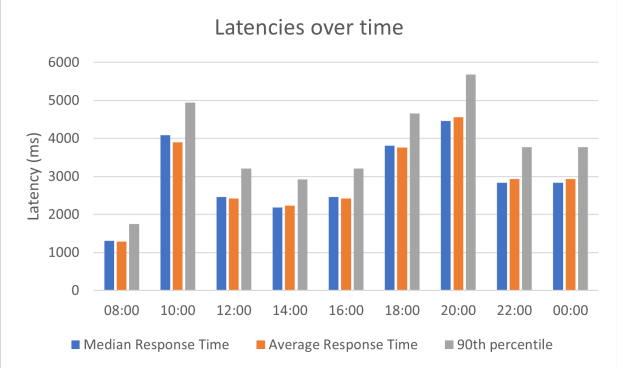

Fig. 2. Base Latency

application and scaling it elastically depending on the amount of network traffic, we rely on two optimization steps. As we have already mentioned in the Design document, our team utilizes the Kubernetes' built-in functionality called HorizontalPodAutoscaler (HPA)[1], which is responsible for adjusting the number of pod replicas, which is constrained by a previously defined range (normally 1-5). Being part of Kubernetes' native autoscaling pipeline, the HPA has to have access to the running pods' resources.

In order to enable this, we introduced a *metrics-server*[2], which is a Kubernetes component responsible for gathering metrics from other components running in the cluster. These metrics are important for monitoring and managing the cluster's performance. CPU usage and memory consumption are prominent example for collected metrics, since they are used by the HPA, when scaling up and down. Although the *metrics-server* is not a core component of Kubernetes itself, but rather an optional extension, it is deployed in the control plane, within the *kube-system* namespace. Each pod in Kubernetes has a *kubelet*, which exposes a */metrics* endpoint on its built-in HTTP server. This way, the *metrics-server* can access the

[1]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[2]https://github.com/kubernetes-sigs/metrics-server

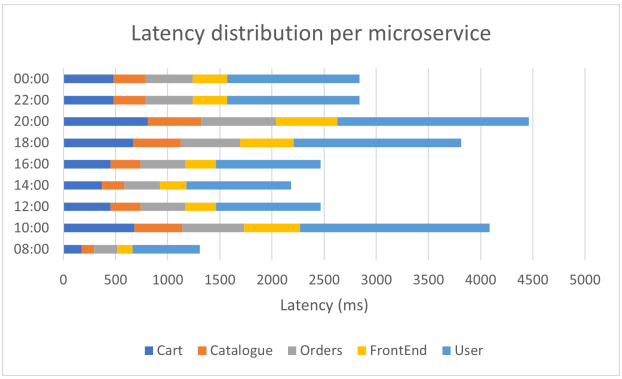

Fig. 3. Base latency distribution

pods' resources. Furthermore, the *metrics-server* implements the Kubernetes Metrics API[3], which makes it possible for the HPA to periodically fetch the collected data. Fig.4 displays the way the *metrics-server* operates and how it communicates with the other components to enable autoscaling.

Based on the data retrieved from the Metrics API and on the resource settings defined in the dedicated YAML file for each microservice, the HPA decides whether to add or remove instances of a specific microservice, thus enabling automatic autoscaling. For example, when there are 500 or 50 concurrent users we need a different number of pods in order to match the workload. If the observed pod, to which autoscaling is applied, exceeds the given CPU percentage, then the HPA increases the number of replicas, which guarantees that the load is handled and the response time does not increase drastically (for example by 5–10 seconds). Another benefit of the horizontal autoscaler is the prevention of bottlenecks and system failures, if an overload occurs.

An efficient usage of the HPA needs an analysis of the results of the load testing with respect to some microservices (Users and Orders) that seem to have a greater increase of latency in comparison to the rest of the system. The mentioned slow-performing microservices are further analysed and their HPA configurations are tuned by repeatedly executing the load tests under various conditions (50, 100 and 500 users working simultaneously).

Apart from horizontal autoscaling, in the Kubernetes deployment YAML file we additionally define how available resources should be divided between each microservice. In other words, resource limits and requests are set beforehand as part of the container configuration. These settings specify the requested computing resources, such as CPU and memory, and the maximum allowed consumption for each container in a pod. Furthermore, we provide here a brief explanation of this approach:

- After measuring the initial state of the system in Fig. 3, we take these microservices that perform slower than the others and analyse them individually.

[3]https://github.com/kubernetes/metrics

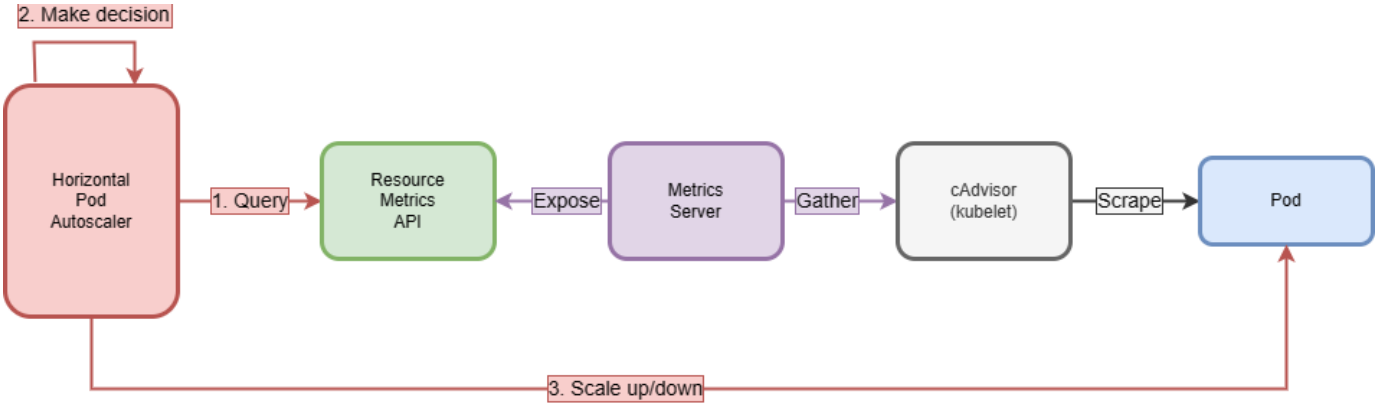

Fig. 4. Horizontal Pod Autoscaler

- The respective resource requests and limits are adjusted and then tested under different amount of load.
- Having the results of the experiments, we investigate if the current behavior has improved with respect to the initial one and if it ensures steady scalability, meaning no bottlenecks or outages.

Our motivation behind this approach is to tune the configuration for the slowest containers in order to ensure that the HPA will work smoothly, since the scale-up is often bound by the available system resources.

Improve the next paragraph if necessary

Our third optimization approach is observing the implementation of the microservices and finding places in code, which cause slow performance and should be refactored. One example would be for a service to have too many functionalities that can be divided between two microservices. Thus, we can reduce the number of requests that need to be handled and scale the new microservices separately.

IV. Architectural Changes

As we have described in the previous section, one of our aims was to refactor and optimize the architecture of the SockShop application. Therefore, we have investigated the microservices Users, Carts, and Orders and have discovered some findings throughout our work.

To start with, the implementation of the Users microservice is based on the programming language Golang, which compiles and runs faster than other languages like Java, Python or C++[4]. Moreover, it consists of uncomplicated functions and does not contain redundant functionalities, which could decrease the performance. In addition, while testing manually and executing the Locust load test, we have not experienced any errors caused on an implementation level.

The Carts microservice is implemented in Java and contains the items that a user wants to purchase. A cart has an assigned ID and related user, whereby the standard CRUD functionalities for maintenance are available.

The Orders microservice, just like the Cart, is also implemented in Java and it gets triggered once an order is placed in the front-end service. The Orders microservice creates a new entity, whereby the customer ID and all other relevant data such as items, address, payment details, and summary are assigned to the entity. Once the order is submitted and the payment gets authorised, the corresponding flag for shipping is set and the order is pushed to a queue, which gets further maintained from the Shipping microservice. The status of the order can be checked with the response API-call once the order is pushed to the queue.

One of the main reasons behind our decision to make no modifications to the current architecture is the outdated release in GitHub, whose latest changes are from 2017. There is no support provided by the creators and we struggled with dependencies as well as scripts, which are now deprecated. Therefore, it requires more time investment to rewrite these files from scratch than the code refactoring itself. What is more, the release is no longer stable and new images of microservices cannot be built.

Taking everything into consideration, we have done no architectural changes and focus on the utilization of HPA and tuning of CPU and RAM parameters, the outcome of which is presented in the following section.

V. Results

Investigating our baseline measurement, which is displayed in Fig. 2 and Fig, 3, we were able to discover which of the microservices under-perform. It can be clearly seen that the latency increases proportionally to the number of users that are performing actions in the system concurrently. This result gives us a warning that somewhere there is a bottleneck, which decreases the response time of the application. For this reason, we take a look at the latency distribution per microservice, where the Users service delivers latency between 1500 and 4000 ms. This behavior is due to the many REST calls transferred between the front-end and the Users service in order to assign a value to each attribute of a user and save it in the database. The second-slowest is the frontend, which serves as a single point of communication with all

[4]https://www.tftus.com/blog/golang-vs-other-languages-a-comparative-analysis

microservices through a proxy, which causes a bottleneck for coming requests. The third place is taken by the Orders service, which has a response time between 500 in best conditions and 2000 ms when high-load traffic occurs.

A. Front end

After a thorough investigation, we discovered that the front-end microservice is actually a proxy to the other microservices. It also includes additional validations on the input user data before the actual requests are sent. Being solely responsible for distributing the requests to the other microservices, its load was constantly peaking. After performing the load test with the minimum defined number of users, we found out that the front end microservice was constantly using 100% of its allocated CPU and 10% of its RAM. After increasing the CPU limits from 300m to 1000m (equal to 1 CPU core) and executing the same scenario, the CPU load dropped significantly to between 40% and 50%, while the RAM consumption remained constant.

Additionally, in order to ensure smooth user experience, we deployed the HPA for the front end microservice. According to our test scenario (varying between 20 and 100 users), we figured the optimal configuration to be minimum 2 and maximum 5 pods with a CPU threshold of 80% for up-scaling. Since the front end is relatively lightweight, it requires significantly lower startup time compared to the other microservices and reacts adequately to traffic peaks. Even at higher loads (250 active users), the front end microservice with the deployed HPA still remains under the average latency and provides responses to user request in less than 500ms on average.

B. Carts microservice

The Carts microservice is the slowest performing one in terms of latency per request. Executing the test scenario with 20 users results in 75-90% CPU load and constant 80% RAM usage. In order to optimize it, the following steps were performed:

- Increased the CPU limit from 300m to 500m (and minimum required CPU from 100m to 300m). This resulted in 60-80% CPU utilization and constant 80% RAM usage (for the same scenario with 20 users). The average latency dropped from 175ms to 100ms.
- Further increased the CPU limit from 500m to 750m, which resulted in a slight CPU utilization improvement (60-70%) and latency drop from 100ms to 90ms. Interestingly, the load on the front end started to increase from 50% to 70-80% CPU load, which indicates that the front end starts to become the bottleneck in this experiment.
- As expected, increasing the CPU to 1000m reduced the average CPU load to 50%, however there were no significant latency changes (latency reduced to 85ms). This change was not incorporated.
- Executing the test scenario with 100 users (instead of 20 up to now) shows the following results: 60-70% CPU load, constant 80% RAM usage and average latency of 220ms per request (compared to the 400 base latency per request).
- Executing the test scenario with 500 users increases the CPU load to 70-80%, but preserves the response latency of 220ms on average.
- Increase the Java Virtual Machine heap size from 64m to 128m (and the maximum from 128m to 256m), which are controlled by the execution parameters '-Xms128m' and '-Xmx256m' and increasing the RAM limit of the pod from 500MB to 700MB. After applying it and executing the test scenario with 100 users, the CPU load slightly reduced to 50%, the RAM usage increased from 400/500MB used to 570/700MB, according to the increased heap size. The measured average latency per request dropped from 220ms to 80ms.
- Executing the test scenario with 500 users increases the CPU load to 60% and the RAM usage to 90% and results in 155ms latency per request (improved from 220ms).
- The last step was deploying the HPA. From all the data collected, we came up with the following configuration: minimum 1, maximum 3 pods with target CPU load of 60% (because of the slow start-up time of the Java Spring Boot microservice of 1-2 minutes).

C. Orders microservice

As said, the orders microservice was also a big candidate for improvement. In particular, we investigated the impact of the three tuning variables of the autoscaler on the Orders POST-API call, which is responsible for the actual order placement. Following the load test scenario that we created, various combinations of the three parameters were executed. In order to validate each combination of the three variables, we performed each test five times and always took the median result regarding the average response time of the test case. Still, for the comparison of the different configuration setups, we also compared the minimum and maximum response times, because the average results were relatively close. We incremented the CPU utilization in steps of 10 percents and the maximum number of pods up to 15. Another important aspect that we considered and used as a filter for further investigation of particular combinations was the parameters' effect on the number of failures. The main observations that we had during the testing are listed below:

- Holding the target CPU utilization below 50 percent for all different combinations of the minimum and maximum number of replicas resulted in worse performance for both successfully completed requests and average response time. For this reason, more dedicated research on the performance with higher CPU utilization was performed.
- By increasing the maximum number of pod replicas we achieved results with a lower maximum response time, but a similar average response time to setups with a lower max number. Also with a higher number of replicas allowed, we experienced high initialization times, so we tried to keep the max number as low as possible.

- By reducing the scope between minimum and maximum allowed replicas we did not receive any significant improvement, so the configurations with a gap of 2 replicas between the thresholds were further investigated in detail.
- We also investigated the impact of CPU, heap and RAM parameters related to the pods, whereby an increase in the values resulted in better test performance.

Following the observations we collected locally the best setup with a minimum number of 3 replicas, a maximum number of 8 replicas, and a target utilization of 60 percent as tuning HPA parameters. The average response time for this setup was 1100ms, whereby no failures in the orders POST request were experienced.

D. Users microservice

In the auto-scaling configuration file of the Users service the allowed pod replicas were limited to the range of 1-10, which was a wider range than it is actually needed. With respect to the CPU utilization it was set at 50%, which was insufficient and in cases of more than 100 users it was exceeded, which caused new replicas to be created. Furthermore, the time needed for an initialization of a new replica is around 30 seconds, which leads to a degradation of the system performance. In order to find an optimal number for the pod replicas, we executed tests with 50, 100 and 500 concurrent users. As it can be expected, for a low level of traffic it was enough to have between 1 and 3 replicas, but tests with more clients showed that 3 can be insufficient and most of the time more than one replicas were necessary. The maximum level of replicas which were created never exceeded 5, therefore we decided to set the range from 2 to 5 replicas.

When the CPU utilization is taken into consideration, although the initially set 50% was not a poor value, because it caused response time of no more than 5 - 6 seconds in an overloaded system, it was still not an optimal solution. Having executed the above-mentioned experiments with different number of users, we were able to see that lower CPU percentage decrease the performance. For this reason, we stopped on 70% as a final decision, because higher values did not seem to deliver better results, and thus we were able to reduce response time to only 1 - 2 seconds in worst conditions.

E. Gluing the whole system together

1) First execution with the improvements: After combining all individual service improvements and running the test scenario on our system, we figures that the orders microservice almost instantly spawns all 8 allowed pods, each of them requiring 3+ minutes of initialization time. This causes the system to be overwhelmed by the initialization and causes a lot of timeouts to the respective requests.

2) Orders service: We reduced the maximum pod instances from 8 to 4. At the same time, we increased their CPU limit from 500m to 1000m to ensure they provide similar latency and performed the same RAM optimization as done in the carts microservice (increasing the heap memory to 128MB and 256MB for initial and maximum respectively and increased the RAM limit for the pods from 500m to 800m in order to reflect the larger heap size).

3) Final evaluation: After deploying the optimized system, we ran the same load test scenario (according to the user distribution described in Fig.1. The results for the median, average and 90th percentile latency for the clients are presented in Fig.5. As visible from the graph, all three indicators are significantly improved over the base implementation (on average 2-3 times lower latency). Moreover, as shown in Fig.6, the latency distribution amongst the five microservices is improved in case of many users (30+). Except the users microservice, all other microservices account for almost equal part of the latency, meaning that there is no bottleneck in the system. The users microservice is an exception, partly because it accounts for almost one third of the requests in our use case and is dependent on the execution of the other microservices. These characteristics prevented us from providing an equal distribution for its latency compared to the other microservices.

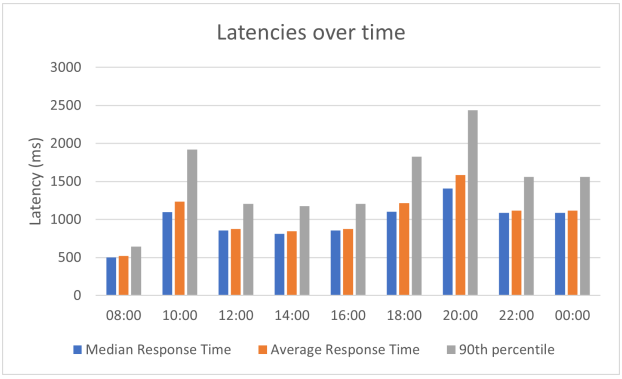

Fig. 5. Improved Latency

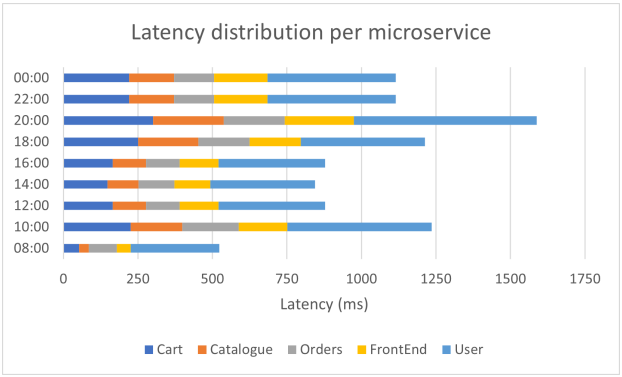

Fig. 6. Improved Latency Distribution

As an overall comparison, Fig.7 displays a visualization of the median, average and 90th percentile latency for the base and improved system. It is clearly visible that the improved system not only improves the latency by a factor of 2-3, but also provides better scaling of the system with respect to the increasing number of users. Fig.8 represents the latency as a

function of the number of simultaneous users. The average latency of the improved system tends to scale better and is expected to continue in similar fashion with increased number of users (above 100).

4) CPU and RAM utilization: As a result from our optimization, the overall system requires at least 7.3 CPUs and 6.8GB RAM and can take at most 17.2 CPUs and 15.2GB RAM, which falls safely into the limits of the tested hardware. The varying limits also indicate that under low load, more than half of the resources will remain unused. A future work might look into the possibility to deploy the system serverless or utilize the unused resources for caching, pre-fetching or cold-storing some of the data in order to optimize the resource utilization.

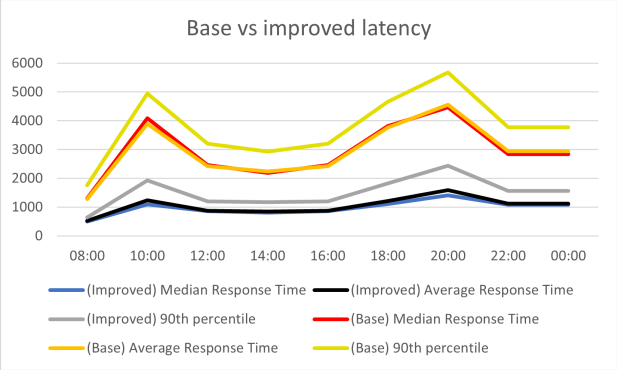

Fig. 7. Base vs Improved Latency

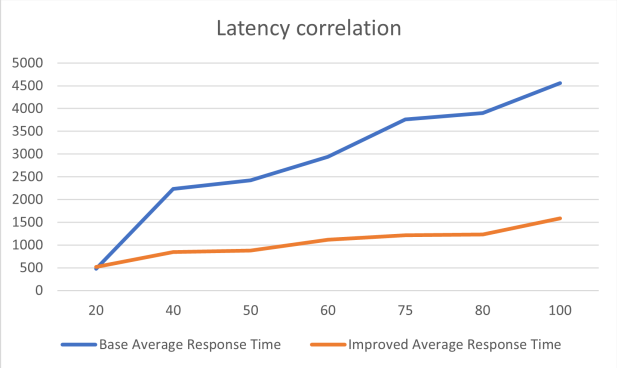

Fig. 8. Latency correlation

VI. Cloud Deployment

Traditionally, as a developer used to create an application the most important decision was to choose the framework, programming language, tool stack for development. But nowadays, choosing a cloud service provider is as important as choosing development tools stack. Moreover, choosing a cloud provider plays a crucial role in deciding the application development stack, such as programming languages, databases etc. There are a lot of available cloud services available in the market such as Google cloud platform, Amazon Web Services, Microsoft Azure, IBM cloud etc. Selecting a cloud service provider can be a difficult task as factors such as scalability, reliability, cost, flexibility, customer support etc. plays an important role in the making a final decision[5].

As of Q2 2023 AWS (Amazon Web Services) is the leading cloud service provider, covering 32% market followed by Microsoft Azure and google cloud at 22 % and 11% respectively, as shown in Fig. 9. For our project one of the crucial factor for choosing cloud service was free tier, as in cost. Each cloud service provides its own version of free tier. For instance, Google cloud provides with 300 dollars worth of credit for new users, whereas Microsoft Azure provides with 100 dollars. Similarly, AWS offers a range of services under an unlimited free tier, and this usage remains valid for a period of up to 12 months from the time of registration. Moreover, AWS offers Amazon EKS (Amazon Elastic Kubernetes Service) which helps to run Kubernetes on AWS cloud and Amazon itself manages the availability and scalability of the Kubernetes deployed application. Unfortunately, the EKS service is not included in the free tier [6] and therefore will not be used for the project deployment, but is worth mentioning. The figure shows

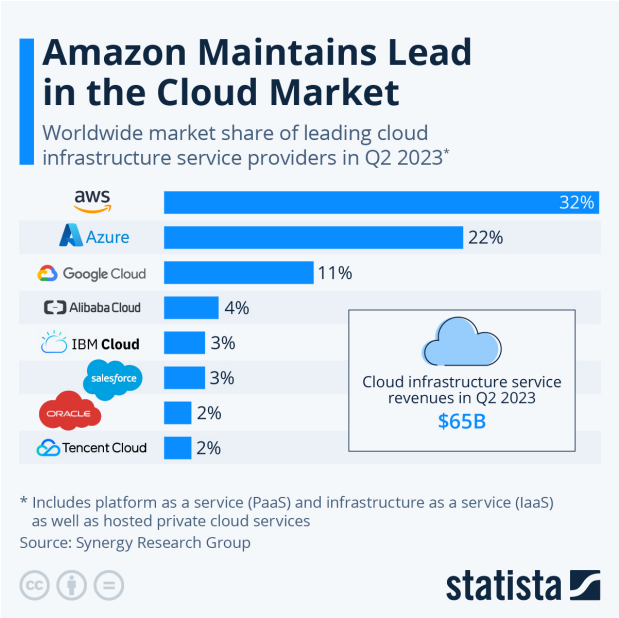

Fig. 9. Leading cloud market share for Q2 2023

The solution in this section will mainly be focused on the cloud level rather than on microservices level of Kubernetes. In order to showcase the impact, a cloud provider can make using different services. The pre-worked sock shop application deployment was done with the help of Amazon Elastic compute cloud (EC2) instances. EC2 allows the user to create, launch and terminate server instances as needed, thus the user has more control over the scalability and availability of the servers running the application. Moreover, there is only one

[5]https://www.researchgate.net/publication/301687424_Selecting_the_Right_Cloud_Services

[6]https://aws.amazon.com/free/

free tier general purpose instance available, namely t2.micro with the specifications as follows:

- CPU: 1vCPU up to 3.3 GHz Intel Xeon Scalable processor(Haswell E5-2676 v3 or Broadwell E5-2686 v4)
- RAM: 1 GiB per vCPU

With such less resources, it was impossible to deploy the whole sock-shop application on one instance. Hence, it was decided to create a cluster of multiple EC2 t2.micro instances manually. The cluster consisted of 6 Instances of which 1 acts as master node where the core-dns, kube-controller, api-server etc. will work and 5 other acts as worker nodes. It was tried to create with 3 worker nodes, but the worker node started crashing due to not enough resources and reached up to 99.9 % of CPU utilization. Therefore, fixating the default minimum worker nodes to 5 and bringing the cluster total specification to :

- CPU: 1vCPU X 6 ≈ 6vCPU up to 3.3 GHz Intel Xeon Scalable processor(Haswell E5-2676 v3 or Broadwell E5-2686 v4)
- RAM: 1 GiB X 6 ≈ 6 GiB

For deployment further additional packages were installed such as kubelet, kubeadm, kuebctl, containerd, weavenet. Initially the servers were pre-installed with docker which used a significant number of CPU-resources, up to 56 % of our low resource instance. Hence, Docker was replaced with another similar tool called containerd, which is a lower-level component focusing primarily on container runtime functionality and helped to bring the node memory usage to 27 % and RAM usage to 31.1 %.

The figure 10 shows as during EC2 instance creation installing and running Kubernetes on master node reached up to 69 % of CPU utilization, prior to running sock shop.

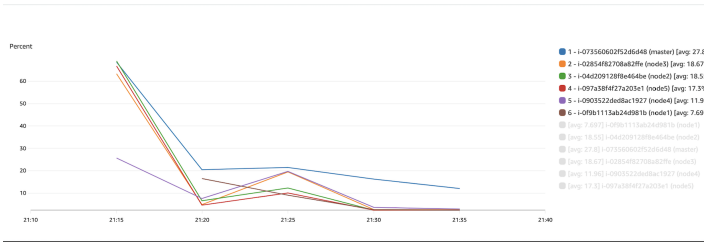

Fig. 10. CPU utilization for different nodes including master

Another challenge faced was Kubernetes itself as Kubernetes minimum requirements are

- CPU: at least 2 CPU of which only had 1 for an instance
- RAM: at least 1700 of which only 965MB was available

This was solved by ignoring the preflight errors which is not a viable option but is a workaround for t2.micro instance, which converts the error to warnings as shown in Fig. 11. Hence, if possible, it is advisable to use at least t2.medium EC2 type instance from AWS for master node.

In order to test out the baseline infrastructure, pre-worked version of sock-shop was deployed and post-worked version of sock-shop is deployed in improved label. Using locust

```
[WARNING NumCPU]: the number of available CPUs 1 is less than the required 2
[WARNING Mem]: the system RAM (965 MB) is less than the minimum 1700 MB
```

Fig. 11. warning message for Kubernetes initialization

tool, the following data was generated as shown in figure 12. The data generated is only for the user at 20:00 which is 100 users as shown in figure 1. As comparing figure 2 and figure 12 90th percentile latency has drastically decreased by 90 % which further decreases by another 45 % once elastic load balancer is deployed with the deployed sock shop application, as seen in figure 13. Similarly, average response time decreased by around 95 % by deploying on the cluster of nodes, as can be again seen from figure2 and figure 12. Furthermore, deploying the application with ELB (elastic load balancer) which is another free tier service offered by AWS. As observed from figure 13 average response time decreased by 65 ms just by using ELB. Furthermore, with our improved solution, deployed with ELB, decreases the response time by another 18 ms. The figure 13 and 12 shows how the handles request increased drastically from 41 request/second to 55 request/second just by using ELB. Another point to note is that the improved deployed sock shop in figure12 has almost the same throughput as the baseline deployed sock-shop with ELB in figure 13. Hence, showing the impact of using cloud service on the created cluster and also proving the point that the changes made in microservices have scaled up the performance of the application.

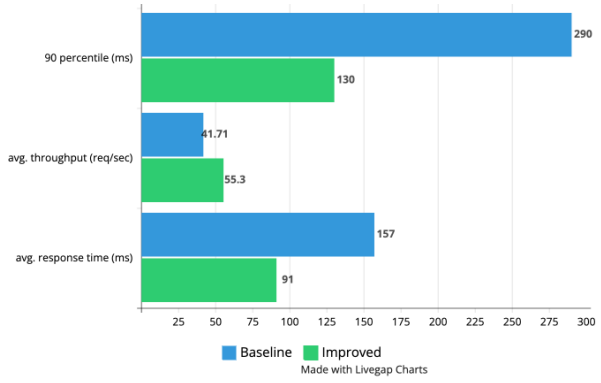

Fig. 12. AWS deployed baseline and improved solution comparision

Moreover, another AWS cloud feature was used, which helps to deploy the same instance in different regions. For instance for the region Frankfurt known as eu-cental-1 in AWS has sub region eu-central-1a, eu-central-1b, eu-central-1c. This allows to deploy any instance in multiple region, so if any instance is down another instance in another subregion is up and running. A test was done using the same tool where again 100 users were taken as before and AWS automatically forwarded them to different cluster instances running in different subregion. The distribution of users in subregion can be seen in figure14. Moreover AWS provides with a free auto-scaler which helps to scale up and

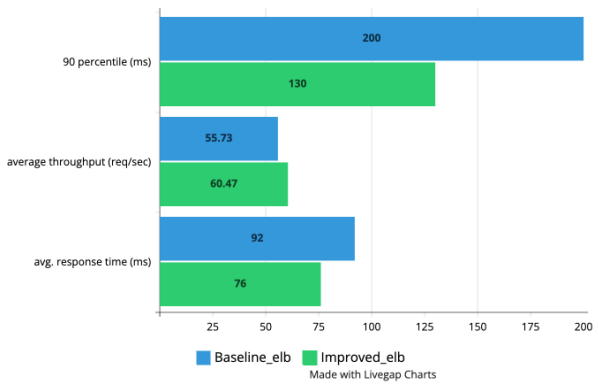

Fig. 13. AWS deployed baseline and improved sock-shop application with elastic load balancer

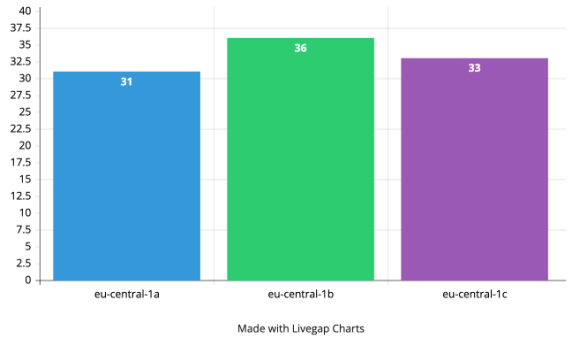

Fig. 14. User distribution in sub regions of Frankfurt (eu-central-1) with ELB

down instances when a certain amount of CPU utilization is reached. The auto-scaler was also tested by deploying the system manually using Amazon AWS dashboard. The auto-scaler uses an insytance template which was created manually also on the dashboard and used during autoscaling. The template contains user data which helps the instance to install all the required packages which is similar to the shell script present in /cnae-sockshop/microservices-demo/deploy/kubernetes/AWS_deployment/deployment.sh.For scalability the certain aws services are the best approach to create or terminate instances on demand. For the performance of microservices it can hinder the result hence auto-scaler was not used while deploying application and the cluster was only deployed in one sub-region.

VII. Conclusion

Appendix

Contribution - the following list describes the individual contributions for each team member:

- Georgi Kotsev - Refactoring steps, Test Scenario and Implementation, Architectural Changes, Results (Introduction part, Users Microservice)
- Georgi Ivanov - Architectural Changes, Orders microservice optimization
- Kalin Iliev - Final Report structure, Initial Baseline measurements (execution of measurements and preparation of graphs), Test Scenario, Improved measurements collection and visualization, Results (Front end and Cart microservices analysis and improvement, Gluing all improvements, Final analysis and optimization, execution of simulation scenarios on the improved microservices, collection and plotting of results, comparison with baseline measurements)
- Krutarth Parwal - Deployment on cloud provider, Cloud deployment chapter, Comparison between on-premise and cloud system deployment, deployment documentations, cloud auto-sclaing, and AWS ELB
- Momchil Petrov - Introduction, Test Scenario and Implementation, Refactoring steps (added metrics-server to the system, description and image for it)

Link to our repository - https://github.com/Smoothex/cnae-sockshop

Link to cloud deployment documentation - Could be found in our github Repo or at https://tasty-ursinia-c85.notion.site/Cloud-deployment-81a84cafef4c440ebca34b823d4d9383?pvs=4