

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

Abstract—Elastic scalability in a microservice architecture improves performance by efficiently allocating resources based on incoming requests, resulting in better responsiveness, cost optimization, and adaptability to varying workloads. It ensures that the system can handle high traffic loads without becoming overwhelmed, while optimizing costs during periods of low traffic. The primary objective of this project is to enhance performance and enable elastic scalability for an e-commerce platform called SockShop, which serves as a demonstration of microservices architecture implementation. This work aims to identify performance bottlenecks within the system, apply optimizations to address them, and ultimately enable dynamic scaling to handle varying traffic levels. By analyzing the architecture, performing load tests, and leveraging autoscaling approaches, this study aims to enhance the performance and scalability of Sock Shop, offering valuable insights for optimizing microservice-based systems.

Index Terms—microservices, performance, elastic scalability, Sock Shop, autoscale

I. INTRODUCTION

In today’s rapidly evolving software development landscape, architects and developers are continuously exploring new ways to design and build scalable and maintainable applications. One architectural approach that has gained significant popularity in recent years is the microservices architecture. This approach emphasizes breaking down complex applications into smaller, loosely coupled services that can be independently developed, deployed, and scaled.

While microservices architecture has gained significant popularity, it’s important to note that it is not the only architectural approach available. Traditional monolithic architecture, for instance, involves building an application as a single, self-contained unit. This approach simplifies development and deployment, but can lead to challenges as the application grows in complexity and scale.

To provide a concrete example of a microservices architecture, one can look at the Sock Shop application. Sock Shop is a popular open-source reference application that demonstrates the principles of microservices in action. It simulates an e-commerce platform for buying socks, where each aspect of the

system, such as inventory management, user authentication, and payment processing, is implemented as an independent microservice.

II. RELATED WORK

Fan et al. [1] compare the performance of microservices architecture and serverless architecture in the context of a cloud-native web application. To conduct the performance comparison, the authors design and implement an application using both microservices and serverless architectures. They evaluate the performance in terms of response time, throughput, and scalability. The paper suggests that no architecture fits all scenarios. Therefore, developers should closely analyze the problem at hand and consider implications of the chosen architecture. According to the authors, microservices deployment has advantages over serverless in terms of cost efficiency and small repetitive requests. Although the serverless deployment suffers from the cold-start problem, it has better scalability capacity, making it suitable for rapidly changing workloads.

Fourati et al. [2] propose an autoscaler by the name of EPMA (Elastic Platform for Microservices-based Applications). It includes an analyzing component, which detects the root cause of reduced performance by examining issues at application (e.g., spike in number of requests or faulty request), container and VM level (e.g., high memory or CPU usage). EPMA also includes a planning module, which creates an action plan to handle the problem identified by the analyzing component. Compared to Kubernetes’ HorizontalPodAutoscaler, EPMA’s scaling approach does not impair response time and performs better in terms of resource allocation efficiency.

III. SYSTEM DESCRIPTION

As already mentioned, Sock Shop¹ serves as a demonstration of a traditional E-commerce application that is designed and built following the principles of microservices. The

¹<https://microservices-demo.github.io/>

provided features in the Sock Shop include the creation of new users and login with credentials, listing the items in a catalogue, viewing an item with a detailed description, adding and removing items from the cart, and purchasing items. By decomposing the different functions into microservices with separated data stores, Sock Shop demonstrates the advantages of the architecture by using different programming languages for the diverse functionalities and services. The communication is realized using REST over HTTP, aiming to simplify the development and testing. The architecture is depicted on figure 1. The front-end service is implemented using Node.js and accessed by a reverse proxy Traefik² that acts as an entry point and internally performs the routing of the request to the appropriate microservices. The services implementing the cart, users and orders features use as a technology for their database mongoDB, but are developed in various languages - Java, Go and .Net respectively. The payment and catalogue feature are developed in Go, whereby the catalogue uses MySQL as a database. The shipping and the Queue Master are developed in Java and are connected with the message broker RabbitMQ.

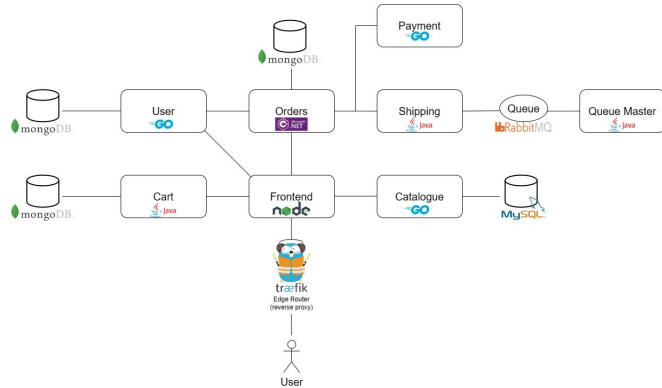


Fig. 1. Diagram of Sock Shop's architecture

The Sock Shop does not have built-in autoscaling capabilities out-of-the-box, because the primary purpose of the application is to showcase microservices architecture and best practices rather than focusing on advanced scaling mechanisms. However, the application can be deployed and managed using container orchestration platforms like Kubernetes. With Kubernetes, the developer can define auto-scaling rules and policies based on metrics such as CPU utilization or request rates, whereby scaling mechanisms, e.g. such as Kubernetes' Horizontal Pod Autoscaler (HPA) that will be further researched during this project, are used to scale the microservices up and down based on the current load.

Sock Shop also provides built-in monitoring capabilities using Prometheus and Grafana where each service has at least one metric. With Prometheus the metrics are collected from different sources, stored, and analyzed, while Grafana is used to create interactive and customizable dashboards. The metrics themselves can vary depending on the deployment and configuration, though some of the most monitored metrics are

request rates, response times, CPU usage, memory consumption, or other service specific metrics.

IV. APPROACH AND IMPLEMENTATION OUTLOOK

In this chapter we outline the technologies we use for real-time testing and monitoring of the performance of Sock Shop. In addition, we analyze the bottlenecks and suggest some approaches for enhancement of the elastic scalability.

A. Load test

The Sock Shop repository provides a load test python script that initializes the connection to the microservices and performs requests against the API. The load tests use Locust.

Locust is an open-source load testing tool written in Python. It allows users to define and simulate user behavior on a web application or API to measure its performance under different loads. Locust is designed to be highly scalable, enabling users to generate thousands of concurrent users from a single machine. It provides an intuitive user interface for defining the test scenarios and collecting real-time statistics during the load test.

Using the provided example, we have defined a custom load test that simulates a realistic user behavior for the Sock Shop microservices. It simulates user behavior by sending HTTP requests to various endpoints of the Sock Shop application, such as browsing products, adding items to the cart, and checking out. The test includes different types of user actions, such as GET and POST requests, with varying load patterns. By simulating real user interactions, the test can evaluate the performance and scalability of the microservices in handling concurrent requests and identify potential bottlenecks, such as slow response times or high resource utilization. This helps in understanding how the microservices perform under realistic conditions and allows for optimization based on the identified bottlenecks.

- 1) Loading the initial page
- 2) User 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

Once the simulation of end user's requests is completed, Locust provides a comprehensive result metric and error report. This output includes information about the requests simulated, specifying which API services were targeted, the number of failed requests, and the throughput of different services. Furthermore, Locust generates additional metrics that depict the percentage of requests completed within specific time intervals for each service.

²<https://traefik.io/traefik/>

B. Analysis of observations

In the microservice-based application Sock Shop, as our task requires, our aim is to discover performance bottlenecks, which cause decrease of handled requests as well as increase of latency. With regard to solving this issue, we rely on an open-source monitoring tool called WeaveScope³, which is also designed by Weaveworks and is used to give a high-level, real-time view of our application’s infrastructure and services. It achieves this by creating an interactive visual representation of the application’s components, including containers, hosts, and network connections. This map is updated in real time, providing us with a dynamic and intuitive view of the application’s architecture. Additionally, this tool offers real-time metrics like CPU and memory usage, which are helpful to observe which of the microservices use most of the resources and thus accent on them as potential bottlenecks.

Figure 2 presents a screenshot from the monitoring tool while executing some operations in the application (e.g., going to the user menu, clicking on the order list and then returning to the main catalogue). Consequently, we are able to examine the connections between the different components of Sock Shop’s architecture, where each one of the lines represents a request being handled between two services. We also utilize this real-time relationship diagram to locate which points are critical and where most of the requests pass through.

In addition to the abovementioned features, WeaveScope provides us with the possibility to optimize performance by manually scaling up and down a service. Furthermore, in combination with the load test described in Section IV-A we are able to test how the performance of the application reacts to different workload conditions.

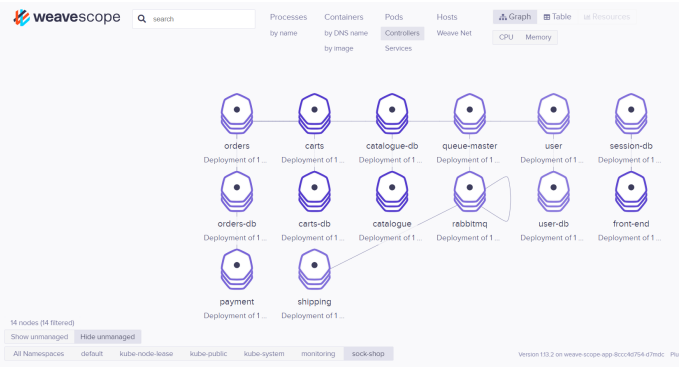


Fig. 2. Communication between services monitored via WeaveScope

1) *Identification of a bottleneck*: To identify the performance bottleneck, we used an approach similar to that applied by the authors in [3]. Here, the authors rely on a load-testing service by WeaveScope using the Locust framework, as mentioned in Section IV-A. This load-testing service enables users to generate a large number of concurrent requests and measure the application response. The load-testing service takes two parameters, namely, “number of concurrent requests” and

“number of clients”, as an input. With the help of the service, we performed baseline and chaos testing of the application. Due to the limited CPU resources at hand, we kept the parameters minimum, just enough, to analyze the application bottlenecks. We started with two users and 50 concurrent requests to gain an understanding of the architecture of the system. Then, we gradually increased the number to 1000 requests. With that many requests, we were able to see that the POST request to the `/orders` endpoint failed for some requests. While monitoring the system via WeaveScope as the load test was running, we found that also other services depend on the `orders` container. As seen in Figure 3 the order service leads to a potential bottleneck issue. In fact, during the tests it had not only the highest number of received and sent requests from and to other services respectively, but also had the highest CPU usage even for a few requests. Similarly, we were able to find other potential bottlenecks from the `user` service, which exhibits a high level of communication with the user database even with a small number of requests.

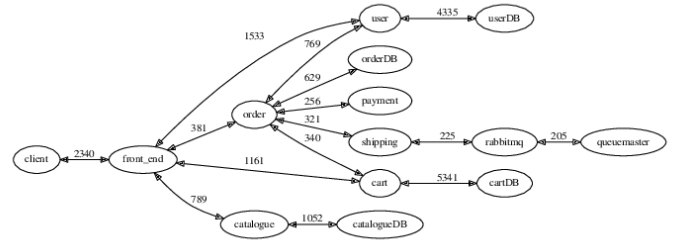


Fig. 3. Architectural view of different services with a number between them depicting the total number of requests and responses between them. The image originates from [3]

C. Refactoring approach

Our refactoring goal is to optimize the microservice architecture for performance and scalability. Since the microservices are deployed in Kubernetes, we will focus on two optimization approaches – scalability of the pods using HorizontalPodAutoscaler (HPA) and providing more resources (CPU and RAM) to each single pod.

Kubernetes HorizontalPodAutoscaler (HPA) is a feature that automatically adjusts the number of replica pods in a Kubernetes deployment based on observed CPU utilization or custom metrics. It helps optimize the performance and scalability of a microservice architecture by dynamically scaling the number of pods to match the current workload. The HPA continuously monitors the CPU or custom metrics of the target pods and compares them against predefined thresholds. If the observed metrics exceed the thresholds, the HPA automatically scales up the number of pods to handle the increased load. Conversely, if the metrics fall below the thresholds, it scales down the number of pods to conserve resources.

By utilizing HPA, microservices can be automatically scaled up or down based on real-time demand, ensuring that there are enough resources available to handle increased traffic or load

³See <https://github.com/weaveworks/scope>, visited on 11/06/2023

spikes. This improves the performance of the microservices by preventing resource bottlenecks and reducing response times.

In order to use HPA efficiently, we will analyse the results of the load testing with respect to microservices that seem to have significant increase of latency compared to the rest of the system. The found bottleneck microservices will be further analysed and their HPA configuration will be introduced and tuned by repeatedly executing the load tests under various conditions.

Furthermore, in a Kubernetes deployment YAML, resource limits and requests can be defined as part of the container configuration. These settings specify the compute resources (such as CPU and memory) that are requested and allowed for each container in a pod. Here's a brief explanation of resource limits and requests:

1) Resource Requests:

- Resource requests define the minimum amount of resources that a container requires to run effectively. It informs the Kubernetes scheduler about the amount of CPU and memory that should be allocated to a container.
- Requests ensure that the necessary resources are available for a container to start running.
- The scheduler uses these requests to make decisions about pod placement and scheduling on cluster nodes.

2) Resource Limits:

- Resource limits specify the maximum amount of resources that a container is allowed to consume. It restricts the upper bound of resources that a container can utilize.
- Limits prevent a container from using excessive resources and potentially impacting the stability and performance of other containers or nodes in the cluster.
- When a container reaches its resource limit, Kubernetes may take actions like throttling or terminating the container.

Both resource requests and limits are expressed in terms of CPU and memory. The CPU resource is measured in CPU units or millicores, indicating the fraction of a CPU core, while memory is measured in bytes or other units like megabytes (Mi) or gigabytes (Gi).

Each container will be individually analysed and the respective resource requests and limits will be defined. The resource configuration is strongly dependent on the technology and the environment used to implement the microservice. That is why we will not only investigate the execution times, but also the start-up times and requirements for each individual technology. By tuning the resources for each container, we will also ensure that the HPA will work smoothly since the scale-up can be bound by the available system resources.

Lastly, after close examination of the typical execution times and start-up times of each individual microservice and considering the work of Fan et al. in [1], we will evaluate

which of them needs to be migrated to FaaS. In fact, a major candidate for a migration to FaaS is the *orders* service because of its event-driven nature and the regarding variable load based on the number of requests. The workload-driven elastic autoscaling this cloud computing model supports is a deciding factor for us, because it represents the aim of our task. In addition to this feature, FaaS provide us with stateless execution of requests, meaning that a large number of them can be handled concurrently and thus enhance both performance and speed of the application. Furthermore, we will use this advantage in order to make sure that no unexpected bottleneck impede the normal request processing. Finally, serverless tends to offer better elasticity than microservices, but still a solid analysis on the trade-offs should be conducted.

APPENDIX

Contribution - the following list defines how the design document is divided:

- Georgi Kotsev - Analysis of observations, Refactoring approach
- Georgi Ivanov - System description
- Kalin Iliev - Load test, Refactoring approach
- Krutarth Parwal - Load test, Identification of bottleneck
- Momchil Petrov - Abstract, Introduction and Related work

Creation Log - here, the prompts we used to generate content are specified, as well as the parts of the text, where the AI answers were used.

- Abstract
 - **Prompt:** 1) How does elastic scalability (meaning scaling up, when there are a lot of incoming requests and scaling down, when there is no huge traffic) affect the performance of a microservice architecture? 2) Summarize all of this in 2 sentences
 - **Answer:** Elastic scalability in a microservice architecture improves performance by efficiently allocating resources based on incoming requests, resulting in better responsiveness, cost optimization, and adaptability to varying workloads. It ensures that the system can handle high traffic loads without becoming overwhelmed while optimizing costs during periods of low traffic.
- Introduction
 - **Prompt:** Write me an introduction to what microservices are, what other types of architectures there are and give Sock Shop as an example of a microservices architecture.
 - **Answer:** In today's rapidly evolving software development landscape, architects and developers are continuously exploring new ways to design and build scalable and maintainable applications. One architectural approach that has gained significant popularity in recent years is the microservices architecture. This approach emphasizes breaking down complex applications into smaller, loosely coupled services that can

be independently developed, deployed, and scaled. ... While microservices architecture has gained significant popularity, it's important to note that it is not the only architectural approach available. Traditional monolithic architecture, for instance, involves building an application as a single, self-contained unit. This approach simplifies development and deployment, but can lead to challenges as the application grows in complexity and scale. ... To provide a concrete example of a microservices architecture, one can look at the Sock Shop application. Sock Shop is a popular open-source reference application that demonstrates the principles of microservices in action. It simulates an e-commerce platform for buying socks, where each aspect of the system, such as inventory management, user authentication, and payment processing, is implemented as an independent microservice

- Related Work

- **Prompt:** Summarize the paper "Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application" by Chen-Fu Fan, Anshul Jindal and Michael Gerndt
- **Answer:** The full answer to the prompt is longer, therefore we only used a part of it:

...

compare the performance of microservices architecture and serverless architecture in the context of a cloud-native web application. To conduct the performance comparison, the authors design and implement an application using both microservices and serverless architectures. They evaluate the performance in terms of response time, throughput, and scalability. The paper suggests that no architecture fits all scenarios. Therefore, developers should closely analyze the problem at hand and consider implications of the chosen architecture. According to the authors, microservices deployment has advantages over serverless in terms of cost efficiency and small repetitive requests.

...

- System Description

- **Prompt:** What is Traefik edge router
- **Answer:** Traefik is an open-source, cloud-native edge router and reverse proxy designed for modern microservices architectures. It acts as an entry point for incoming traffic to your applications, routing requests to the appropriate microservices based on various rules and configurations. As an edge router, Traefik provides several key features:

...

- **Prompt:** Let's go back to Sock Shop, can you tell me about the autoscaling capabilities of the application?
- **Answer:** The Sock Shop microservices demonstration application does not have built-in autoscaling capabilities out-of-the-box. The primary purpose of the Sock Shop application is to showcase microservices architecture and best practices rather than focusing on advanced scaling mechanisms.

However, the Sock Shop application can be deployed and managed using container orchestration platforms like Kubernetes, which provide native autoscaling capabilities. With Kubernetes, you can define autoscaling rules and policies based on metrics such as CPU utilization or request rates. These rules can automatically adjust the number of running instances of the Sock Shop microservices based on the defined thresholds.

...

- Approach and Implementation Outlook

- **Place in Text:** Identification of a bottleneck
- **Prompt:** Explain more about WeaveScope
- **Answer:** The primary goal of Weave Scope is to give you a high-level, real-time view of your application's infrastructure and services. It achieves this by creating an interactive visual representation, or "map," of your application's components, including containers, hosts, and network connections. This map is updated in real-time, providing you with a dynamic and intuitive view of your application's architecture.
- **Place in Text:** Load test
- **Prompt:** I am writing a university assignment where I am supposed to analyse an existing microservice architecture and optimize it with respect to performance and scalability. The existing system is Sock Shop by Weaveworks. The GitLab repositories are here: <https://github.com/microservices-demo>. The microservices are deployed in Kubernetes. Using their load-test repository and tools, I am supposed to perform a realistic load test on the microservices and find the bottlenecks that need to be optimized. The load tests are implemented using python's package locust. First, write a 5-6 sentences description of locust package. Afterwards, using the provided python code that implements such test in locust, provide a technical specification of the implemented test and describe why it is a realistic use case for the Sock Shop example.
- **Answer:** Locust is an open-source load testing tool written in Python. It allows users to define and simulate user behavior on a web application or API to measure its performance under different loads. Locust is designed to be highly scalable, enabling users to generate thousands of concurrent users from a single machine. It provides an intuitive user interface for defining the test scenarios and collecting real-time

statistics during the load test. The provided Python code implements a realistic load test for the Sock Shop microservices. It simulates user behavior by sending HTTP requests to various endpoints of the Sock Shop application, such as browsing products, adding items to the cart, and checking out. The test includes different types of user actions, such as GET and POST requests, with varying load patterns. By simulating real user interactions, the test can evaluate the performance and scalability of the microservices in handling concurrent requests and identify potential bottlenecks, such as slow response times or high resource utilization. This helps in understanding how the microservices perform under realistic conditions and allows for optimization based on the identified bottlenecks.

- **Place in Text:** Refactoring approach
- **Prompt:** Describe shortly what Kubernetes HorizontalPodAutoscaler is and how can it be used to optimize the performance and scalability of a microservice architecture
- **Answer:** Kubernetes HorizontalPodAutoscaler (HPA) is a feature that automatically adjusts the number of replica pods in a Kubernetes deployment based on observed CPU utilization or custom metrics. It helps optimize the performance and scalability of a microservice architecture by dynamically scaling the number of pods to match the current workload. The HPA continuously monitors the CPU or custom metrics of the target pods and compares them against predefined thresholds. If the observed metrics exceed the thresholds, the HPA automatically scales up the number of pods to handle the increased load. Conversely, if the metrics fall below the thresholds, it scales down the number of pods to conserve resources. By utilizing HPA, microservices can be automatically scaled up or down based on real-time demand, ensuring that there are enough resources available to handle increased traffic or load spikes. This improves the performance of the microservices by preventing resource bottlenecks and reducing response times.
- **Place in Text:** Refactoring approach
- **Prompt:** Describe the resource limits and requests that are defined in the Kubernetes deployment YAML as part of the container configuration In a Kubernetes deployment YAML, resource limits and requests can be defined as part of the container configuration. These settings specify the compute resources (such as CPU and memory) that are requested and allowed for each container in a pod. Here's a brief explanation of resource limits and requests: 1. Resource Requests: - Resource requests define the minimum amount of resources that a container requires to run effectively. It informs the Kubernetes scheduler about the amount of CPU and memory that should be

allocated to a container. - Requests ensure that the necessary resources are available for a container to start running. - The scheduler uses these requests to make decisions about pod placement and scheduling on cluster nodes. 2. Resource Limits: - Resource limits specify the maximum amount of resources that a container is allowed to consume. It restricts the upper bound of resources that a container can utilize. - Limits prevent a container from using excessive resources and potentially impacting the stability and performance of other containers or nodes in the cluster. - When a container reaches its resource limit, Kubernetes may take actions like throttling or terminating the container. Both resource requests and limits are expressed in terms of CPU and memory. The CPU resource is measured in CPU units or millicores, indicating the fraction of a CPU core, while memory is measured in bytes or other units like megabytes (Mi) or gigabytes (Gi).

REFERENCES

- [1] Fan, Chen-Fu, Anshul Jindal, and Michael Gerndt. "Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application." In CLOSER, pp. 204-215. 2020.
- [2] Fourati, Mohamed Hedi, Soumaya Marzouk, and Mohamed Jmaiel. "Epma: Elastic platform for microservices-based applications: Towards optimal resource elasticity." Journal of Grid Computing 20, no. 1 (2022): 6.
- [3] Chanh Nguyen, Amardeep Mehta, Cristian Klein and Erik Elmroth. (2019). Why cloud applications are not ready for the edge (yet). 250-263. 10.1145/3318216.3363298.