P1 Report - Service Meshes like Istio

Exploring the potential of service meshes, monitoring capabilities and load testing

Michael Mente, 01634435
University of Vienna
Vienna, Austria
a01634435@unet.univie.ac.at

Abstract

While monitoring microservice applications is an area where much research has been conducted and solutions been developed, most of the implementations for monitoring purposes are often part of each microservice itself, creating additional work for developers to match infrastructural requirements.

However the recent rise of the service mesh pattern allows application developers to reduce development effort as well as introducing new functionality in already existing infrastructure, whilst significantly reducing the susceptibility to software bugs.

In this work I explore the possibilities of monitoring microservice applications by applying existing service mesh implementations to reference microservice applications, as well as measuring and analyzing the performance impact of this additional layer onto an existing system to deduct possible impacts on user experience for such applications, in multiple testing scenarios inspired by realistic use cases.

CCS Concepts: • **Applied computing** → **Service-oriented architectures**; • **Networks** → *Cloud computing*; • **Software and its engineering** → *Software as a service orchestration system*; • **Human-centered computing** → Information visualization.

Keywords: Service Mesh, Grafana, Istio, load testing, Kubernetes, K8s, PromQL, Minikube

1 Introduction

Service meshes are a separate layer of infrastructure introducing observability, security and resiliency deployed in microservice-based application architectures. In the context of my work, I focused on the open-source service mesh implementation Istio[1]. Istio was designed to function optimally with regard to the combination of containerized services in the use of the orchestration framework Kubernetes. Here, service meshes usually control the way the individual components communicate with each other as seen in figure 1. Among other things, service meshes can also provide additional functionality such as service discovery, encryption or load balancing.

In order to provide the mentioned functionality, so-called sidecar injection is performed. This means that in a containerized environment, an additional service is added to each container of the application, which acts as a proxy for this service and manages all network communication of the service. As shown in figure 2 the individual sidecar proxies together form a service mesh as network traffic is only routed from sidecar to sidecar. Configuration and management of the sidecars is provided by a dedicated interface to create a uniform configuration of individual services via the control plane.

The reasoning employing this technology is to improve productivity of developers by taking away certain implementation necessities as service to service communication and therefore also reducing the surface for undetected errors. Since service meshes are a rather new pattern, the overall adoption rate in enterprise technology stacks is still relatively low. [2, 6]

1.1 Service Meshes and alternative Solutions

The Istio Service mesh allows developers to offload a number of development tasks. These include network traffic encryption, logging, load balancing, service discovery, API gateway, traffic splitting for canary releases when new software versions are available, and integration with systems outside of a Kubernetes cluster, to name a few. However, a special feature of Istio is the support of third party plugins to integrate additional specialized functionalities.

These features allow to reduce the time to market for application development and at the same time to develop more robust software, since the implementation of the business cases is in the foreground. The centralized management of the sidecars through the control plane as shown in figure 1 allows to create consistent configurations for all sidecars at once and thus avoids pitfalls caused by single misconfigured microservices.

However, service meshes also have disadvantages. Since a sidecar is added to each service of an application, this increases the computational overhead required to process a query, ultimately adding another step in the processing of a request. This can increase latency, which cannot be directly reduced by developers because it is caused by a third system, which is critical depending on the application. This problem can occur predominantly with particularly small and lightweight microservices where a relatively large number of microservices are used. In such cases, total processing time can grow significantly. In many cases, the decision to use

[1] https://github.com/istio/istio

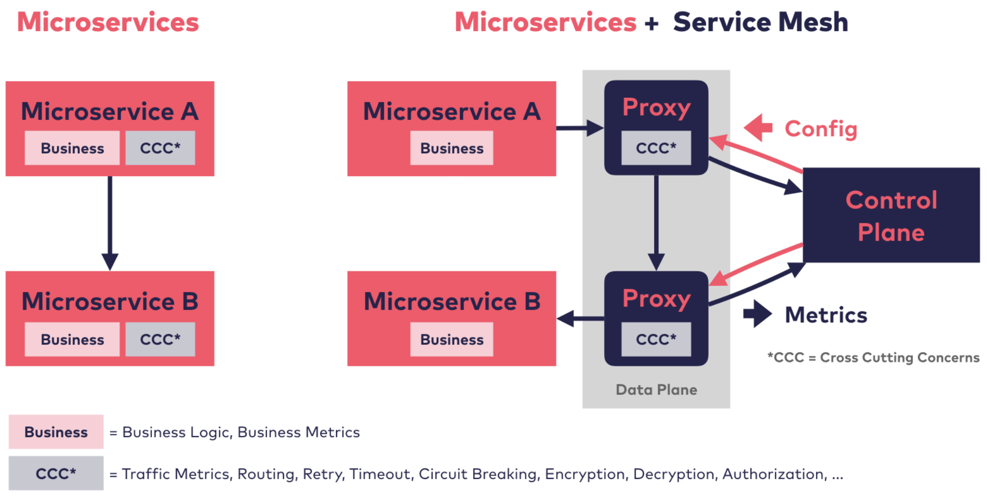

Microservices
Microservices + Service Mesh
Microservice A
Business
CCC*
Microservice B
Business
CCC*
Microservice A
Business
Proxy
CCC*
Config
Control Plane
Microservice B
Business
Proxy
CCC*
Metrics
*CCC = Cross Cutting Concerns
Data Plane
Business = Business Logic, Business Metrics
CCC* = Traffic Metrics, Routing, Retry, Timeout, Circuit Breaking, Encryption, Decryption, Authorization, ...

Figure 1. Service mesh architecture [5]

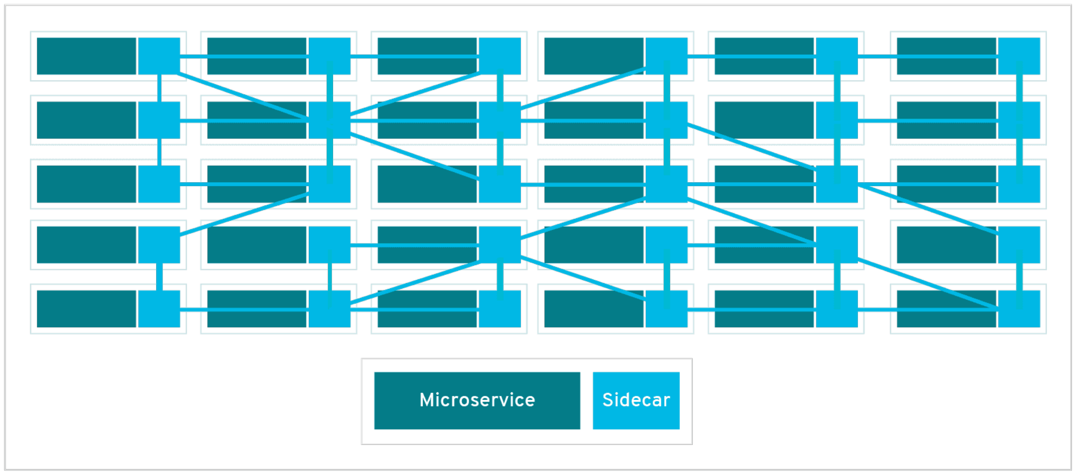


Figure 2. Multiple sidecars form a service mesh collectively[6]

this technology is made at the beginning of the application development process, as otherwise the features introduced by service meshes are often already implemented in existing applications, making a later introduction obsolete, if possible at all.

An alternative approach to service meshes that can provide similar functionality is the adoption of libraries to replace these features. In this case, these libraries are bound into each individual microservice instance, which in turn means that the configuration is organized in a decentralized manner and thus increases the potential for errors. [2, 5, 8]

2 Microservice applications

To explore the features mentioned in the previous section, it was necessary to use a microservice application. I examined various applications for their features, deployment possibilities aswell as their suitability for further experiments and load testing.

2.1 Google microservices demo

The first application I considered using for further benchmarks was the "Online Boutique" application provided by GoogleCloudPlatform [2] for demonstration purposes. This

[2]https://github.com/GoogleCloudPlatform/microservices-demo

application provides the functionality of an online store consisting of 11 individual microservices as shown in figure 3. This application is originally designed to showcase the functionality of the Google Kubernetes Engine (GKE). Therefore, it is technically feasible to run this application locally using Minikube, which emulates a single node cluster. One advantage of this application is that it comes with a load generator that already simulates user behavior and generates network traffic.

To generate the load the open source load generator Locust [3] is used. It is important to note that the load is generated within the system by its own microservice and not from external sources. There are several changes that would have been necessary to create a realistic scenario where all components of the service mesh can be tested as described in section 4.

Unfortunately I discovered that at the time, this application did not work reliably as multiple microservices were crashing repeatedly in my Minikube deployment resulting in a crashloop state and therefore this application was unsuitable for further experimentation.

2.2 Teastore reference application

The next application I analyzed for further proceeding was the Teastore reference application [4]. This application also provides the functionality of an online store. In contrast to the previous application, the Teastore was developed to provide a common basis for scientific experiments in this field of research, as this application uses modern technologies to be easy to deploy. Six different microservices are used as shown in figure 4. Although the number of microservices is lower than in the previous application, the Teastore was designed to generate a relatively high computational load at comparatively low request rates.

These requests can be generated with an included load generator using different technologies. On the one hand with the specially developed LIMBO http load generator, on the other hand with JMeter, an already established tool for load generation, and finally also Locust, which can also perform load generation benchmarks. It should be mentioned that the official support for locust was added after I had developed my own load generator for Locust.

However, a drawback in the context of my research is that this application was not designed with service meshes in mind. This means that this application already contains an integrated load balancer. It is possible to use the application without the Ribbon load balancer [5] which is developed by the Netflix company, however the comparability to other experiments based on the same application is lost, as in this case load balancing is done by another system, in this case Istio.

Finally, after some tests, I had to decide against the use of this promising application for the following reasons. Since I only had a single virtual machine available at the time, it was necessary to run the load balancing on the same system as the Minikube cluster. The load generation itself is comparatively resource intensive which in turn reduces the available resources for the system to be tested. I have already allocated resources for this scenario as described in chapter 3.2, but the load from this application was too heavy to achieve meaningful results because the number of successfully executed requests was very low.

2.3 Istio Bookinfo reference application

The last application I considered for further testing was the sample application provided with Istio itself [6]. This application provides a very basic functionality showing reviews to certain books, it offers a main page and a login functionality, which is implemented using four different microservices. This simplicity results in very light computational demands by design to run this application, which was one of the main concerns with the Teastore application.

In addition, this application has been designed to work with Istio and is therefore ideally suited to explore the functionalities of the service mesh, as compatibility is guaranteed. To highlight one of the advanced features of Istio, there are three different versions of the *review* microservice that can be used to load split the network traffic, as show in figure 5. One of the drawbacks, however, is that this application was not originally designed to perform load tests, so there is no included way to generate network traffic and thus computational load.

3 Deployed technologies

In order to deploy microservices appropriately and subsequently test them with regard to various aspects, knowledge of a wide range of technologies is necessary.

3.1 Kubernetes

Kubernetes (K8s) [7] is a tool to coordinate and orchestrate many individual microservices packed into individual containers. It provides a very broad range of functions to cover as many scenarios as possible in a production environment. Since Kubernetes itself is a distributed system, at least three different machines are recommended in a standard deployment scenario to ensure increased reliability. Those machines are then accessible as a single Kubernetes cluster abstracting away the fact that multiple physical (or virtual) machines are used as shown in figure 6. Kubernetes exposes the control plane interface via the command line application `kubectl`

[3] https://github.com/locustio/locust
[4] https://github.com/DescartesResearch/TeaStore
[5] https://github.com/Netflix/ribbon
[6] https://github.com/istio/istio/tree/master/samples/bookinfo
[7] https://github.com/kubernetes/kubernetes

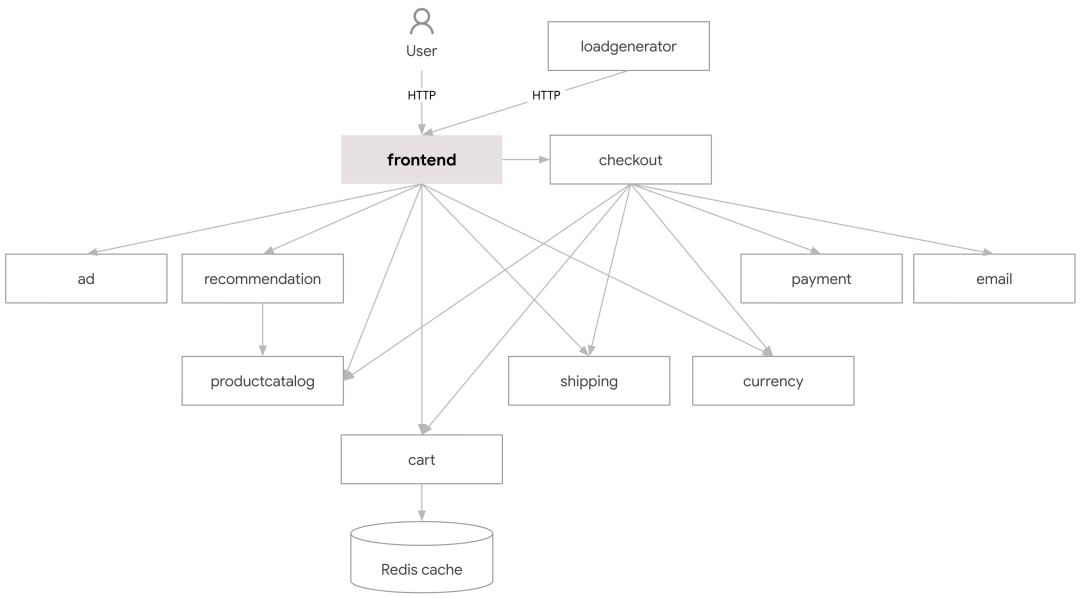


Figure 3. Architecture of google microservice application "Online Boutique" [3]

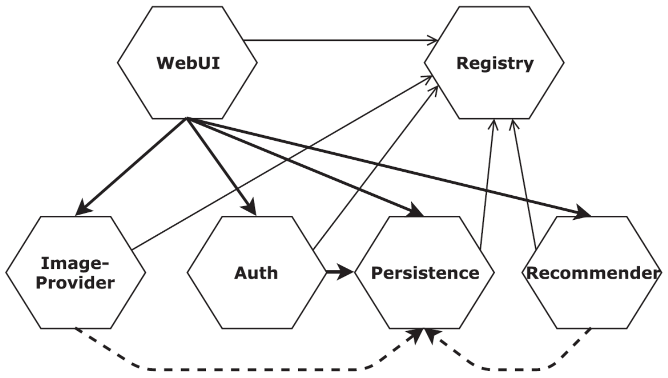


Figure 4. Architecture of Teastore reference application [9]

as well as via a web interface. Fundamentally, microservice applications are configured using custom configuration files in .yaml format that conform to the Kubernetes API in a declarative manner. This provides straightforward version management and transparent behavior in complex applications.

3.2 Minikube

To deploy the application using Kubernetes, I decided to use Minikube[8], which this is a tool supplied by the Kubernetes developer team. This technology allows to simulate a single node Kubernetes cluster within a virtual machine on a single computer instance. Initially I decided to perform the experiments on a single machine which required me to execute the load generation on the same computer. This in turn required me to use Minikube as installing Kubernetes itself on the provided computer would have not allowed to run any other software not part of the cluster. The aim of Minikube is primarily development and testing scenarios of applications and hence should not be used in a production environment, but I figured it would still fit for my purpose.

3.2.1 Setup. Similar to conventional virtual machines, Minikube also requires a hypervisor for virtualization. It implements the functionality to use multiple driver types, KVM, VirtualBox, Hyper-V to name a few.

To run my experiments I was provided with a Ubuntu Linux host system. Since the host system itself runs in a virtual machine, it was not possible to use the KVM hypervisor driver as nested virtualization is not enabled on this system. For this reason had resort using VirtualBox driver, which offers worse performance since this is a level 2 hypervisor in contrast to the KVM hypervisor.

The resource allocation during the installation I have largely attributed to Minikube and left only a reduced amount for the execution of the benchmarks as well as the necessary tasks of the operating system. I allocated 14 of the 16 available CPU cores to Minikube as well as 28 GB of the 32 gigabytes of main memory, as shown in listing 1.

```
minikube start --cpus 14 --memory
    28000 --driver=virtualbox
```

Listing 1. Minikube setup command

[8]https://github.com/kubernetes/minikube

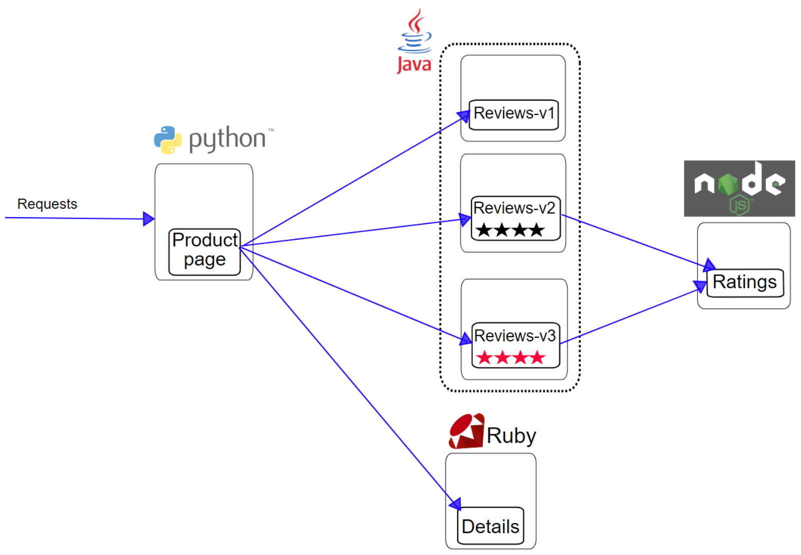


Figure 5. Architecture of Istio Bookinfo sample application [4]

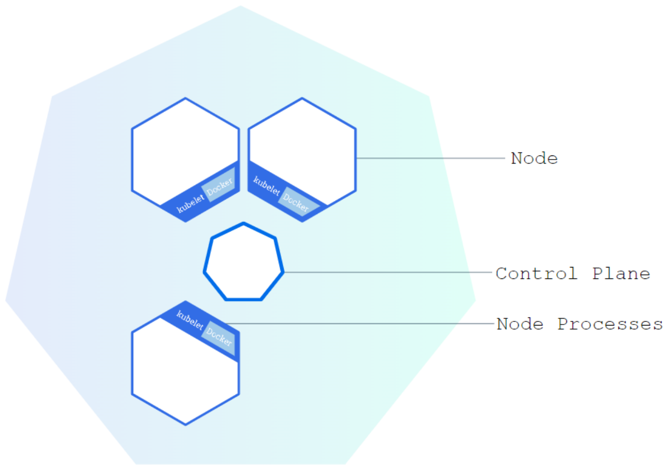


Figure 6. Kubernetes topology [1]

3.3 Istio

As already explained in section 1.1, I have chosen Istio as the service mesh to be tested, as it is one of the most mature implementations at the moment, as well as open source software.

3.3.1 Setup. Since istio was developed for use with Kubernetes, the installation procedure is smooth and straightforward. After downloading the current version of the software and adding it to the PATH environment variable, the installation can be performed with a single command. After that it is only necessary to indicate the desired namespaces of Kubernetes for which sidecars will be activated when an application is deployed. I decided to use the default namespace as applications would be deployed to this namespace if not indicated differently. Also I used Istio v1.14.1 which was the latest release at the time of installation, as shown in listing 2.

```
curl -L https://istio.io/downloadIstio
    | ISTIO_VERSION=1.14.1 TARGET_ARCH
   =x86_64 sh -
cd istio-1.14.1
export PATH=$PWD/bin:$PATH
istioctl install
kubectl label namespace default istio-
   injection=enabled
```

Listing 2. Istio installation & setup procedure

3.4 Prometheus

One of the key aspects of this project, in addition to exploring the features of service meshes, is monitoring cloud-enabled applications. The monitoring tool Prometheus[9] plays a central role here. Prometheus, like all the software components used so far, is open source software. Prometheus offers a extensive suite of functionalities ranging from data query of the sources, to an own time series database, to an own query language PromQL to specify and process the collected data. Data retrieval is based on a pull model, where Prometheus queries data sources at a regular time interval, and it works best if the queried data is numerical for metric generation. In combination with Istio, monitoring can be applied to microservice applications which do not implement this features as they are introduced via the sidecars.

[9] https://github.com/prometheus/prometheus

3.4.1 Setup.
As monitoring is an important aspect of service meshes, Istio delivers a built-in configuration for prometheus that enables monitoring of applications. For this reason, the installation, like that of Istio itself, is done with a single command as shown in listing 3.

```
cd istio-1.14.1
kubectl apply -f samples/addons/
    prometheus.yaml
```

Listing 3. Prometheus installation

3.5 Grafana

To visualize the data gathered with Prometheus, I chose to use the open source visualization software Grafana[10], another popular tool that works seamlessly together with Prometheus [7].

3.5.1 Setup.
As with prometheus, the installation within the Kubernetes cluster works smoothly with a single command as seen in listing 4.

```
cd istio-1.14.1
kubectl apply -f samples/addons/
    grafana.yaml
```

Listing 4. Grafana installation

3.5.2 Dashboard creation.
During the development of the dashboard, I decided to visualize mostly application-independent key performance indicators to keep the dashboard versatile, since among other things the decision of the application was not yet set. I chose factors that can tell a cluster operator at a glance whether the application is working reliably and correctly in a production environment as shown in figure 7.

In the event that one of these values exceeds the carefully chosen permissible range, it is highlighted in red color for easy detection of error conditions. Otherwise the values are shown in green color to indicate normal operation

I have divided the dashboard into two sections. on the one hand, I visualize important hardware factors such as CPU and memory utilization, as well as the number of requests per second and the uptime of the system. In the second section I show a detailed breakdown of the requests per second over a certain period of time, which can be selected by the user. In addition to this, a breakdown of whether the requests were answered successfully or not is displayed. Finally, the total number of bytes transferred is displayed as well as the average time it takes to process a request.

The last section depends on the application and shows the individual latencies of the individual microservices as well as the success rate of the responses, which can provide information about the state of individual components of the application.

[10] https://github.com/grafana/grafana

These queries are made using the PromQL query language, since Grafana uses Prometheus as its data source in this configuration, as shown exemplary in listing 5.

```
(sum(irate(istio_response_bytes_sum{
    source_workload_namespace!="istio-
    system", reporter="source"}[1m])) +
     sum(irate(istio_request_bytes_sum{
    source_workload_namespace!="istio-
    system", reporter="source"}[1m])))
    / 1000
```

Listing 5. Transmitted bytes/second PromQL query

3.6 Kiali

A crucial tool that I have used is Kiali[11]. It allows the user to visualize the network traffic within the microservice topology as a graph to verify the proper operation of individual components. Kiali is closely linked to Istio and even allows to modify Istio configurations, as it is an alternative management console for Istio. It enabled me to discover that the microservice application I selected at first (Section 2.1) was not functioning correctly as some microservices were stuck in an faulty state, as the individual requests can also be checked for their success.

3.6.1 Setup.
As with the previously described extensions, Kiali can be installed in a very similar, very user-friendly way by just executing a single command as shown in listing 6.

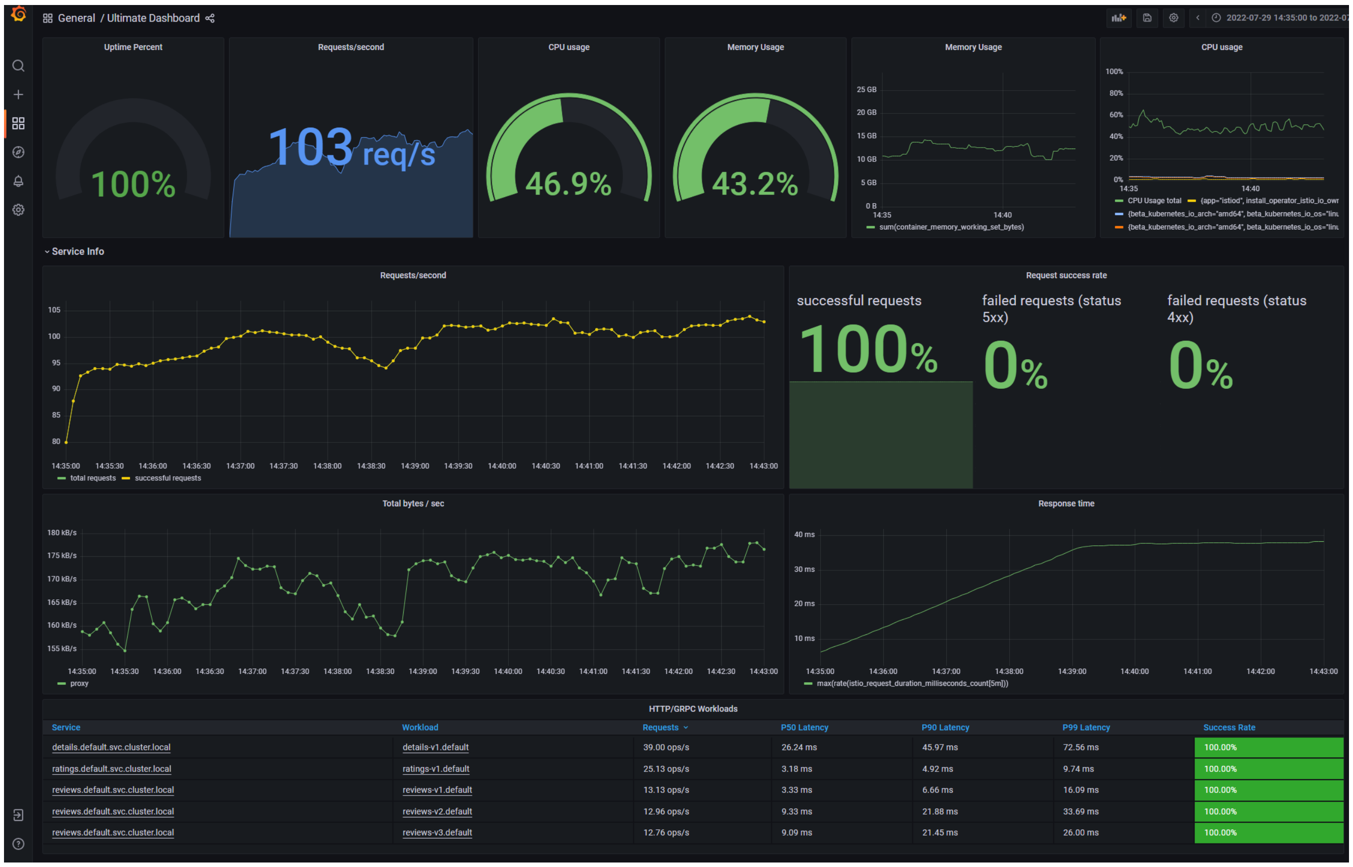
```
cd istio-1.14.1
kubectl apply -f samples/addons/kiali.
    yaml
```

Listing 6. Kiali installation

3.7 Locust

The last tool I used in my experiments is Locust[12]. It is an intuitive performance testing solution to generate load for diverse systems. Since this tool was developed with Python it is relatively easy to create your own extensions and load scenarios. The interesting approach is that generally the intended behavior of a single user is implemented, that is, what actions a single user will perform on a website on a per request basis. Subsequently, the number of users at runtime can be set and modified to increase or decrease the load. Using Locust in the first place was necessary due to the fact that the Bookinfo application (Section 2.3) I had decided to use for further testing did not provide an embedded way to generate load on the system.

[11] https://github.com/kiali/kiali
[12] https://github.com/locustio/locust



HTTP/GRPC Workloads

Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
details.default.svc.cluster.local	details-v1.default	39.00 ops/s	26.24 ms	45.97 ms	72.56 ms	100.00%
ratings.default.svc.cluster.local	ratings-v1.default	25.13 ops/s	3.18 ms	4.92 ms	9.74 ms	100.00%
reviews.default.svc.cluster.local	reviews-v1.default	13.13 ops/s	3.33 ms	6.66 ms	16.09 ms	100.00%
reviews.default.svc.cluster.local	reviews-v2.default	12.96 ops/s	9.33 ms	21.88 ms	33.69 ms	100.00%
reviews.default.svc.cluster.local	reviews-v3.default	12.76 ops/s	9.09 ms	21.45 ms	26.00 ms	100.00%

Figure 7. Custom Grafana dashboard showing key perfomance indicators

3.7.1 Benchmark implementation. Since the functional scope of the Bookinfo application is very manageable, the development of a load generator was also very straightforward. I decided to implement the user behavior[13] as follows: A user sends 5 requests to the main page in an interval between 1 and 10 seconds between two requests. At times I had also implemented a login routine, but this generated erroneous status codes which could possibly distort the test results. This behavior must be callable via the so called locustfile when the load generator is started subsequently.

4 Benchmark results

To run the performance benchmarks, I chose a predefined time interval instead of a variable one, as this makes it easier to compare different benchmark scenarios. I have set the duration of the load generation to 600 seconds, so that initial performance fluctuations due to possible initializations of any subsystems can be absorbed. To avoid sudden changes in load that could potentially bring other side effects, I increased the number of users every second until the desired number of simulated users was reached as it is shown exemplarily in listing 7, which depicts a low load scenario.

```
locust --host=http://swa-michaelmente.
    cs.univie.ac.at:32460 -u 400 --
    spawn-rate=80 --headless --run-time
    =600s --csv benchmarks/baseline/$(
    date "+%d-%m-%y_%H-%M-%S") --csv-
    full-history -f load-generator/
    locustfile.py
```

Listing 7. Benchmark execution for low load scenario

Since I want to measure the influence of Istio and individual features on the overall application performance, I have defined three different scenarios. The first scenario, which I use as a reference for further measurements, is based on the application running on the cluster with all default features of Istio enabled. I then deactivated individual features of Istio and analyze whether there are any differences in system performance.

Istio offers load balancing on several granularity levels, and load balancing can significantly improve the performance of an application. Therefore, for a second scenario, I bypass the ingress gateway provided by Istio, which queues

[13]https://github.com/whiskeywolke/P1-SS22/blob/main/bookinfo/load-generator/locustfile.py

incoming requests depending on the load of the system. This is a valid scenario because an ingress gateway cannot always be replaced by the one provided by Istio, as this component can also take over other system-relevant tasks depending on the application.

I have defined the third scenario as follows; in addition to the disabled ingress gateway, I have also disabled the virtual service functionality of Istio, which handles load balancing between the individual microservices within the application.

4.0.1 Selection of data source.

For the subsequent processing and analysis of the data, however, I used the values reported by the load generator Locust instead of the data I visualized with the Grafana dashboard. The reason for this is that Prometheus uses a polling strategy with a fixed time interval to collect data which may result in the accuracy of the data not being as precise as necessary. Also, with Prometheus the end to end latency can't be measured that effectively because Prometheus itself is running inside the Kubernetes cluster. These differences depending on the source of the data collected can be seen in Figure 11.

4.1 Loadtesting & Minikube on same VM

Initially, I ran the benchmarks on a single computer instance. This means that the Kubernetes cluster and the load generation with Locust are run simultaneously on one computer. Although Minikube has been allocated a large part of the system's resources, this does not mean that these resources are not longer available for other tasks, but only that Minikube does not use more than the predetermined resources. However, the generation of the load requires more resources than are still available, which are not allocated to Minikube. Therefore, there were resource requests that exceeded the capacities of the computer. As a result, the measurement results from this iteration of the test setup are not particularly meaningful because the impact of load generation on the results cannot be accurately measured, as the high CPU requirements of Locust as well as the requirements of Minikube would be scheduled by the operating system. I decided to generate a load of 400 (simulated) users.

Regression & Correlation analysis. To test the assumption that Locust had an influence on the results, I carried out several measurements, the results of which varied greatly. In addition, the response time for the very low number of queries per second is extremely high, which may indicate problems in the experimental setup. I visualized these results in figure 8 and performed a regression analysis as well as determining the correlation of the two parameters. We see that the correlation is very low and the values are strongly scattered, which confirms the impact of Locust on the Minikube cluster.

4.2 Loadtesting & Minikube on separate machines

Since it turned out that it is not practical to run the experiments on one computer, the next step was to generate the load on a separate computer in the same network. However, when setting up the testing conditions, it became clear that there is no simple solution for opening applications running in the Minikube cluster to the network. However, since I conveniently used the Virtualbox hypervisor, I was able to attach a NAT-network-interface to Minikube to handle requests from the network, as described in section A.4.

4.2.1 Low load scenario.

In the new testbed configuration I ran the previously used test scenario of 400 users with the same load again to determine how significant the difference is when running Locust on a separate computer. The results have shown that when the resources allocated to Minikube can be used effectively, the response time to requests is significantly reduced.

Regression & Correlation analysis. To confirm this assumption, I performed yet another regression analysis on the newly collected data as well as a measurement of the correlation between the two properties.

Here it can be seen that, compared to the previous experimental setup, the number of requests has increased by more than a factor of ten, while at the same time the response time has been reduced by half as shown in figure 9.

Another striking feature is that the measurements delivered very similar results, regardless of the scenario tested. This can be attributed to the fact that the hardware resources of the Minikube cluster are not yet maxed out at a load of 400 users.

4.2.2 High load scenario.

As a follow-up to the previous performance measurement, I have now introduced another test scenario. The test setup remains the same, as well as the different configurations of Istio, however, I have now increased the number of users to 2000 in order to generate a maximum load, as shown in listing 8. This is the largest possible number of users that can be simulated on the provided machine, as otherwise limits of concurrently open files are reached by the operating system, which shows a possible limitation of Locust.

```
locust --host=http://swa-michaelmente.
    cs.univie.ac.at:32460 -u 1500 --
    spawn-rate=100 --headless --run-
    time=600s --csv benchmarks/
    baselineHigh/$(date "+%d-%m-%y_%H-%
    M-%S") --csv-full-history -f load-
    generator/locustfile.py
```

Listing 8. Benchmark execution for high load scenario

Regression & Correlation analysis. However, analysis of the data collected for the new scenario has revealed some

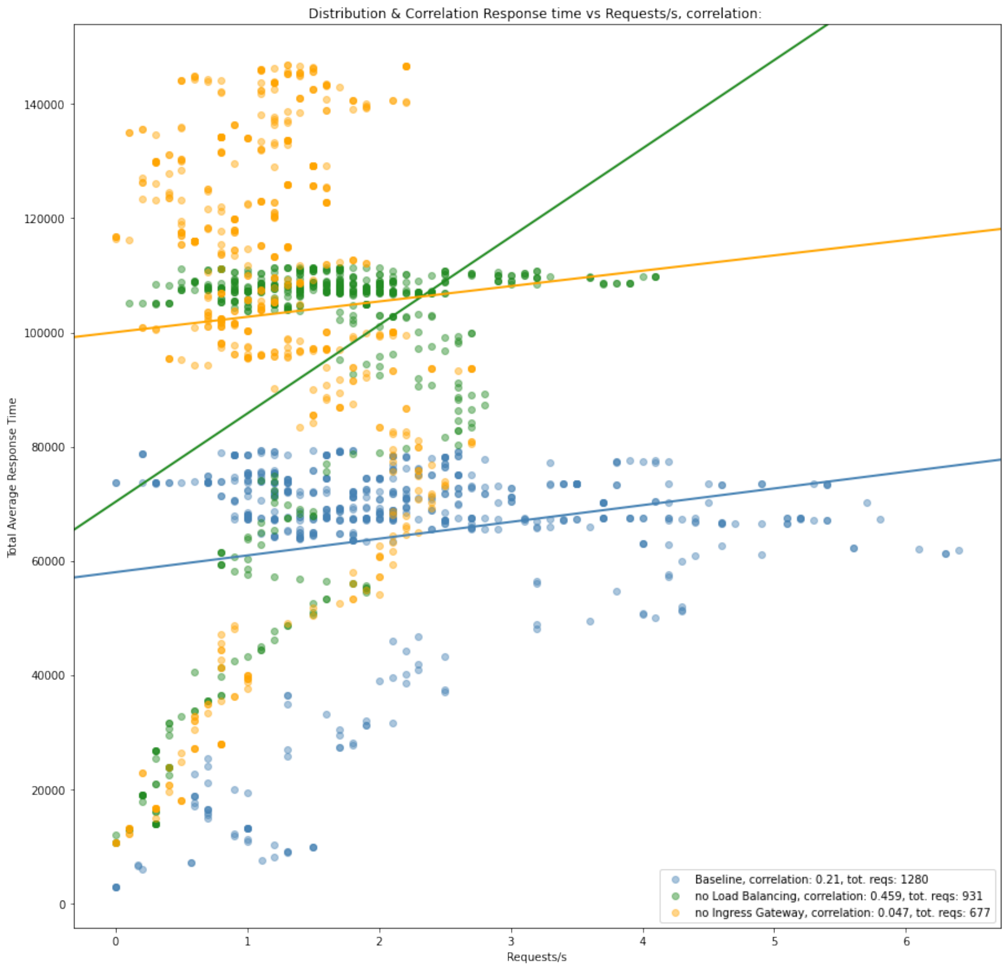


Figure 8. Regression analysis of requests/second & response time within single virtual machine

interesting facts: The time required to process the requests is actually a little less than in the previous performance measurement, whereas the variation in the number of requests per second has grown significantly. This is due to the fact that more requests per second are sent. However, I also checked whether the Istio has any influence at all on this load. Here it can be seen that the results of my measurements are all very similar to each other and only a small positive influence by Istio can be seen as shown in figure 10.

Data difference depending on reporter source. As already mentioned, significant differences in the data can be seen depending on whether one examines the data from Locust or from reports (Figure 11).

This is caused not only by the different reporting strategies, but also by the fact that the perspective of measurement is different. The reason that the data differs so much is also due to load balancing and request queuing of individual components of the application. On closer observation, it is also noticeable that the length of the benchmark differs, which is can also be attributed to the queuing of the incoming requests. In the upper left graph of figure 11, Locust reports the number of requests sent per second, whereas the upper right graph shows the number of requests processed, as reported by Prometheus.

4.2.3 Comparison of testbed configurations. As Figure 12 already suggests, the difference in the performance of the system depending on the different test setups is relatively small. Both in the total average response time, the difference in the medians of the different measurements can be classified in the same area, as well as in the number of requests per second. Here, however, a slight performance advantage can be seen in the query throughput when Istio is deactivated. This assumption is confirmed by the total number of requests executed in the defined benchmark time window as shown in table 1, which differ by a factor of only 3 percent at most.

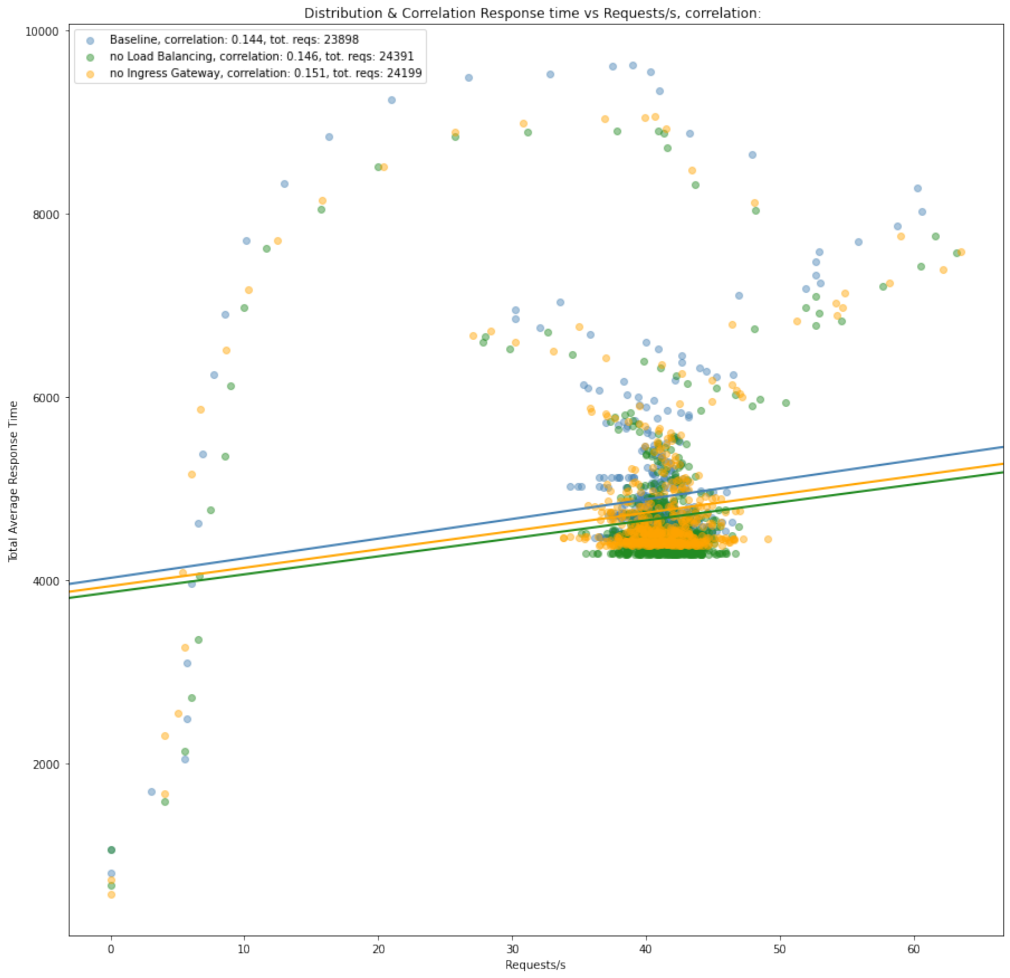


Figure 9. Regression analysis of requests/second & response time with dedicated load generation virtual machine in low load scenario

Benchmark configuration	Request count
Baseline	22399
no Load Balancing	22621
no Ingress Gateway	22753
no Istio	23094

Table 1. Total requests performed during high load scenario

5 Drawbacks in testbed setup

Apart from new insights into service meshes, the experiments just described have also shown that there is still room for improvement in the experimental setup. One of the aspects is that the use of the Virtualbox hypervisor, which is an L2 hypervisor, introduces another level of abstraction. This leads to reduced performance on the one hand, but on the other hand it also increases the number of variables that can potentially distort or at least influence further results. partly these problems could be addressed by using another L1 hypervisor like KVM, which was originally the preferred choice, but not applicable due to issues of nested virtualization.

Apart from that, the choice of a hypervisor only arose after the decision was made to use Minikube for cluster virtualization. Minikube itself is rather designed for testing purposes and therefore not well suited to receive network traffic that does not originate from the host machine, and thus another abstraction layer must be introduced to redirect external traffic as was necessary in the second iteration of my experimental setup. This network traffic redirection was only possible by using the Virtualbox hypervisor beforehand and is not an officially supported use case which could also introduce new variables into the measurements.

As a further consequence of the use of Minikube and Virtualbox, problems eventually arose that the Teastore application deployed had too high resource requirements to

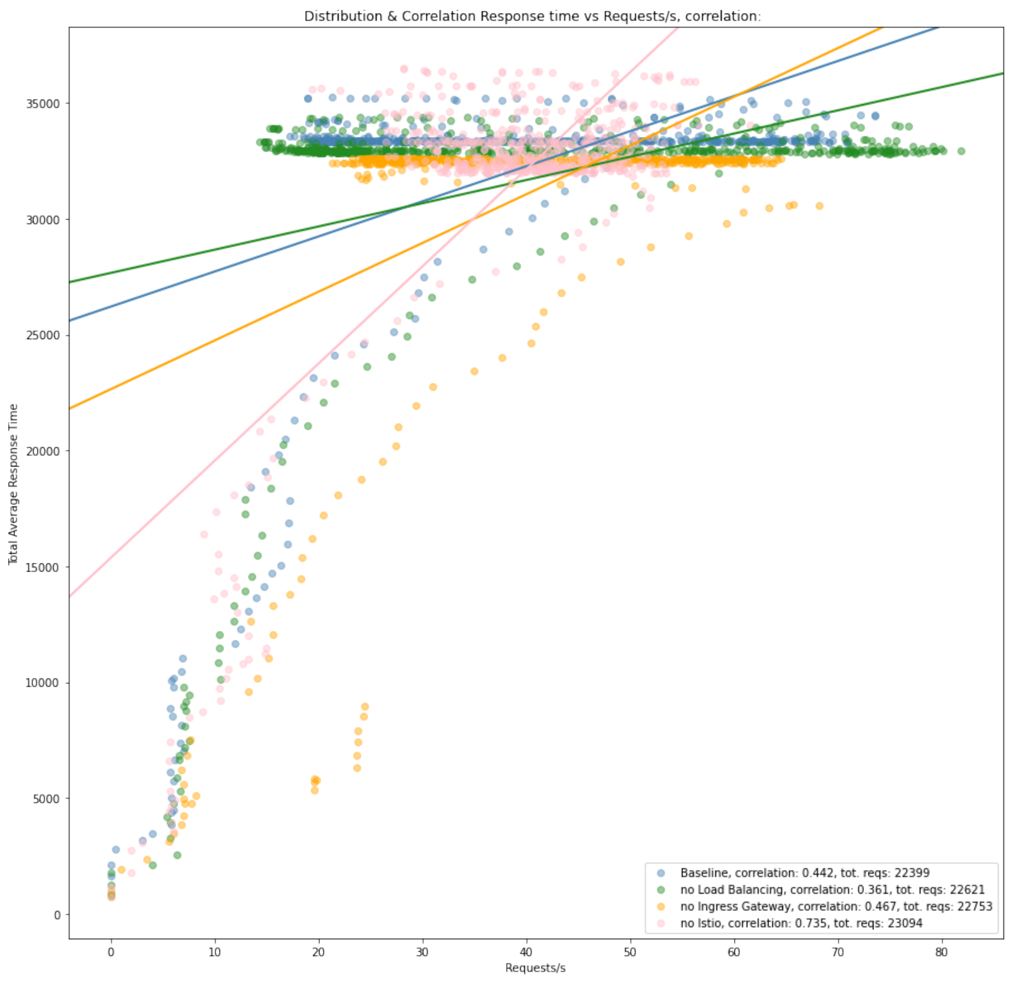


Figure 10. Regression analysis of requests/second & response time with dedicated load generation virtual machine in high load scenario, including reference of benchmark with Istio disabled

function reliably to deliver good measurement results, as the computational overhead grew significantly due to aforementioned factors. However, these problems only became apparent at a late stage, which would not have made it sensible to completely revise the current test setup.

Due to the application deployed, only simple load tests could be performed that do not necessarily correspond to real user behavior, which is due to the limited functionality of the Istio Bookinfo application.

5.1 Future Improvements

A large part of the variables in my measurements can be eliminated by incorporating the use of multiple virtual machines into the testbed setup from the beginning. In this way, it is possible to remove all abstraction layers from the system under test and at the same time create a more realistic setup. To do this, it is necessary to install Kubernetes natively on one of the machines. As a result, I can then test the proven Teastore application to get comparable and more realistic results, since this application generates a realistic load on the system. As another consequence, in a further step, more realistic load can be generated with already known tools, since the developers of the Teastore application now also support the Locust load generator. Furthermore, I have only scratched the surface regarding the features of Istio that I have enabled and disabled for different benchmark configurations, which leaves certain aspects of the impact of service meshes unexplored.

6 Conclusion

In this work, I have explored the pattern of service meshes as well as the associated monitoring mechanisms. By using modern state of the art technologies I learned about the possibilities of monitoring within an application between microservices. To achieve this, the use of Grafana and the associated proprietary query language PromQL was necessary. Another part of this work was the exploration of

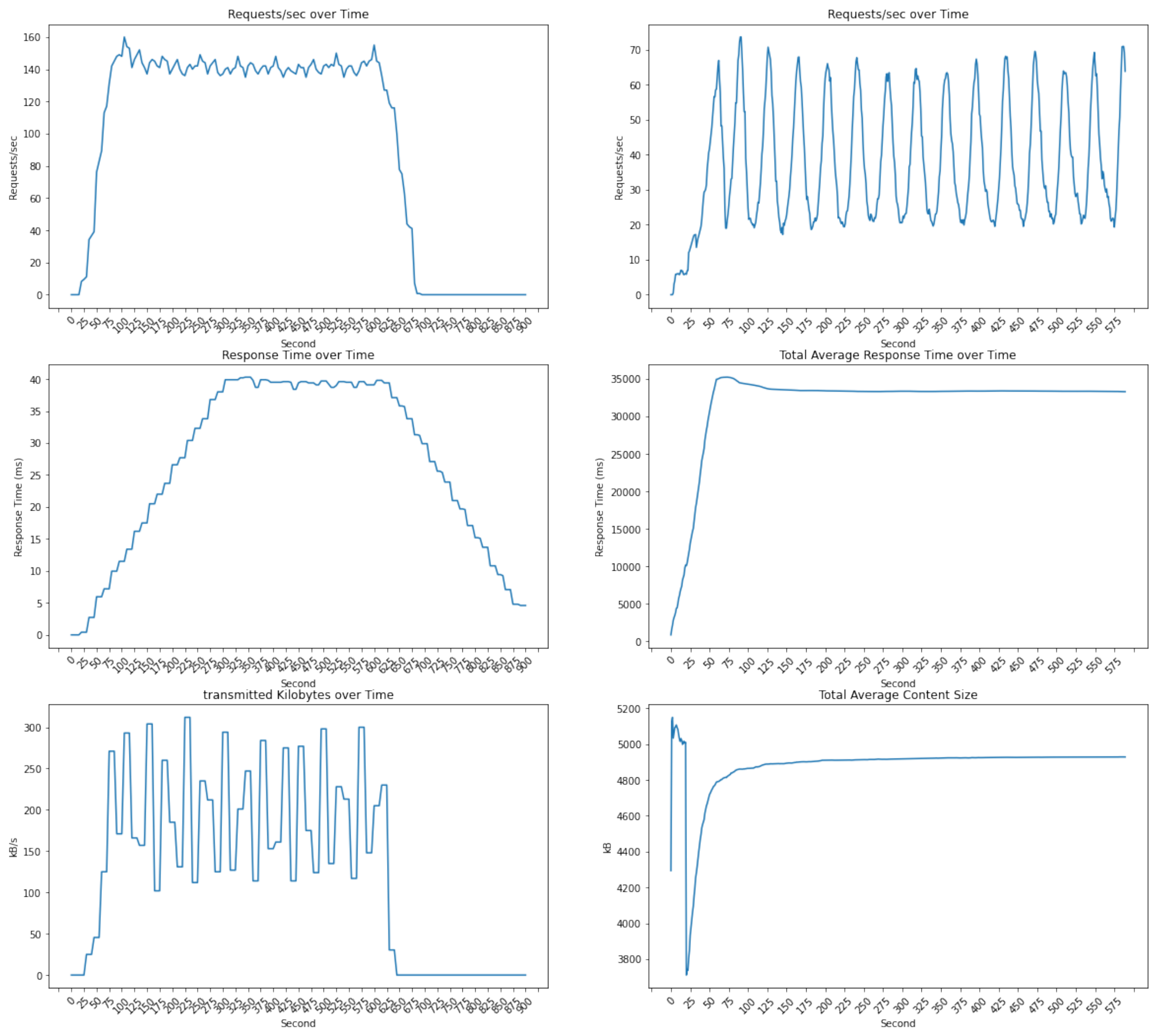


Figure 11. Benchmark data captured by Locust in left column, data captured by Grafana captured in right column

the performance requirements of the service mesh implementation Istio on the hardware resources and the resulting advantages and disadvantages for the end-user of an application. For this I created several test scenarios which I tested in different test setups. This showed that the influence of Istio is relatively small as long as the system is not fully utilized under load. Additionally, I analyzed the collected data with common methods and drew conclusions that I presented in the previous chapters. Finally, I have carefully examined various aspects of my experimental setup in order to obtain more meaningful results in further iterations of this experiment. In the course of this work, I have learned what advantages and disadvantages service meshes can have for cluster operators and application developers, and which tools are necessary to monitor an application in a production environment and to leverage them in a meaningful way in order to detect problems early and thus provide a continuous end-user experience.

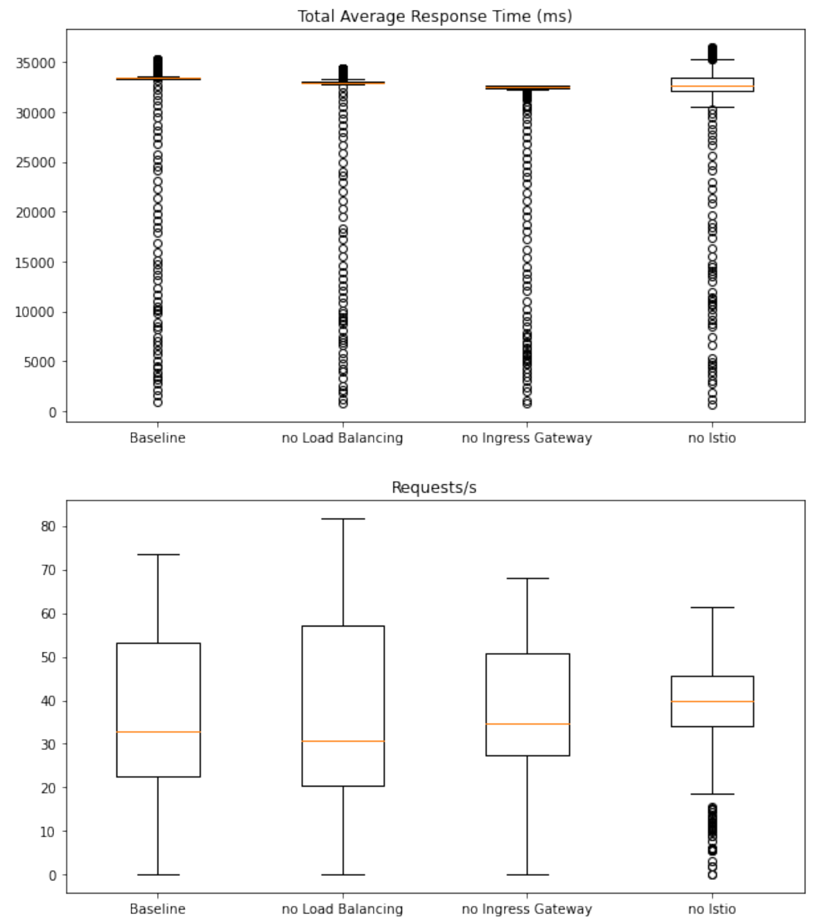


Figure 12. Difference between benchmark configurations for high load scenario

References

[1] The Kubernetes Authors. 2021. Using Minikube to create a cluster. https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/

[2] Alexander S. Gillis and James Montgomery. 2021. What is a service mesh and how does it work? https://www.techtarget.com/searchitoperations/definition/service-mesh

[3] GoogleCloudPlatform. 2018. Googlecloudplatform/microservices-demo: Sample Cloud-Native Application with 10 microservices showcasing Kubernetes, Istio, grpc and OpenCensus. https://github.com/GoogleCloudPlatform/microservices-demo

[4] Istio. 2017. Istio/samples/bookinfo at master · istio/istio. https://github.com/istio/istio/tree/master/samples/bookinfo

[5] Anja Kammer, Chrisitne Koppelt, Jörg Müller, Hanna Prinz, Christopher Schmidt, and Eberhard Wolf. 2022. Service Mesh Architecture. https://servicemesh.es/

[6] RedHat. 2018. What's a service mesh? https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh

[7] Ran Ribenzaft. 2021. Prometheus and Grafana: The perfect combo. https://epsagon.com/tools/prometheus-and-grafana-the-perfect-combo/

[8] Abdellfetah Sghiouar. 2022. When not to use service mesh. https://medium.com/google-cloud/when-not-to-use-service-mesh-1a44abdeea31

[9] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. 2018. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems* (Milwaukee, WI, USA) *(MASCOTS '18)*.

A Step by step guide

This section lists the commands I used to install and setup Istio, as well as configuring and performing the benchmarks for the chosen Microservice application for reproducibility. An extended version of this list of commands can be found at https://github.com/whiskeywolke/P1-SS22/blob/main/commands.txt

A.1 Setup Minikube & install Istio inside cluster

```
minikube start --cpus 14 --memory
    28000 --driver=virtualbox

curl -L https://istio.io/downloadIstio
     | ISTIO_VERSION=1.14.1 TARGET_ARCH
     =x86_64 sh -
cd istio -1.14.1
export PATH = $PWD / bin : $PATH
istioctl install
kubectl label namespace default istio-
    injection=enabled
istioctl install
```

A.2 Installing Istio addons for cluster management

```
kubectl apply -f samples/addons/
    grafana.yaml
kubectl apply -f samples/addons/
    prometheus.yaml
kubectl apply -f samples/addons/kiali.
    yaml

kubectl get all -n istio-system
```

A.2.1 Accessing dashboards on local machine.

This is how I configured port forwarding via SSH to my local machine to be able to access the dashboards.

```
ssh -L 20001:localhost:20001
    michaelmente@swa-michaelmente.cs.
    univie.ac.at
kubectl port-forward svc/kiali -n
    istio-system 20001

ssh -L 3000:localhost:3000
    michaelmente@swa-michaelmente.cs.
    univie.ac.at
kubectl port-forward svc/grafana -n
    istio-system 3000

ssh -L 9090:localhost:9090
    michaelmente@swa-michaelmente.cs.
    univie.ac.at
kubectl port-forward svc/prometheus -n
     istio-system 9090
```

A.3 Installing Bookinfo Istio reference application

```
kubectl apply -f samples/bookinfo/
    platform/kube/bookinfo.yaml
```

Verification of successful deployment

```
kubectl exec "$(kubectl get pod -l app
    =ratings -o jsonpath='{.items[0].
    metadata.name}')" -c ratings --
    curl -sS productpage:9080/
    productpage | grep -o "<title>.*</
    title>"
```

A.4 Configuring Minikube for access from Network

A.4.1 Publishing port for access via Istio ingress gateway.

This setup is used to access the Minikube cluster from the local network and to access the service by the provided via the Istio ingress gateway.

```
export INGRESS_PORT=$(kubectl -n istio
    -system get service istio-
    ingressgateway -o jsonpath='{.spec.
    ports[?(@.name=="http2")].nodePort
    }')
vboxmanage controlvm "minikube" natpf1
     "ingress,tcp,,$INGRESS_PORT,,
    $INGRESS_PORT"
```

Verify if setup is working:

```
curl -s "http://swa-michaelmente.cs.
    univie.ac.at:$INGRESS_PORT/
    productpage" | grep -o "<title>.*</
    title>"
```

A.4.2 Publishing port for direct service access.

This type of setup is used to access the the service directly to bypass the ingress gateway. This is how I configured the system for one of the benchmarks.

```
kubectl expose deployment productpage-
    v1 --type=NodePort --name=ingress-
    bypass
kubectl get svc ingress-bypass
export INGRESS_BYPASS=$(kubectl get
    service ingress-bypass -o jsonpath
    ='{.spec.ports[0].nodePort}')
vboxmanage controlvm "minikube" natpf1
     "ingress-bypass,tcp,,
    $INGRESS_BYPASS,,$INGRESS_BYPASS"
```

Verify if setup is working:

```
curl -s "http://swa-michaelmente.cs.
    univie.ac.at:$INGRESS_BYPASS/
    productpage" | grep -o "<title>.*</
    title>"
```

For the last type of benchmark I disabled the virtual-services which are provided in the same file as the gateway. It does not matter that I do not deactivate the ingress gateway for the previous benchmark as the gateway gets bypassed anyway

```
kubectl delete -f bookinfo-gateway.
    yaml
```

A.5 Benchmarking the system

These are the benchmark runs I used, each of them is limited to the same time, I chose 600 seconds.

A.5.1 Low load scenario.

I used the this load scenario with 400 users simultaneously interacting with the system where the user count increases by 80 users every second.

```
locust --host=http://swa-michaelmente.
    cs.univie.ac.at:32460 -u 400 --
    spawn-rate=80 --headless --run-time
    =600s --csv benchmarks/baseline/$(
    date "+%d-%m-%y_%H-%M-%S") --csv-
    full-history -f load-generator/
    locustfile.py

locust --host=http://swa-michaelmente.
    cs.univie.ac.at:32726 -u 400 --
    spawn-rate=80 --headless --run-time
    =600s --csv benchmarks/
    noIngressGateway/$(date "+%d-%m-%y_
    %H-%M-%S") --csv-full-history -f
    load-generator/locustfile.py

locust --host=http://swa-michaelmente.
    cs.univie.ac.at:32726 -u 400 --
    spawn-rate=80 --headless --run-time
    =600s --csv benchmarks/
    noLoadBalancing/$(date "+%d-%m-%y_%
    H-%M-%S") --csv-full-history -f
    load-generator/locustfile.py
```

A.5.2 High load scenario.

As a high load scenario I chose to have 2000 users accessing the system concurrently, where the spawn rate is 100 users per second.

```
locust --host=http://swa-michaelmente.
    cs.univie.ac.at:32460 -u 1500 --
    spawn-rate=100 --headless --run-
    time=600s --csv benchmarks/
    baselineHigh/$(date "+%d-%m-%y_%H-%
    M-%S") --csv-full-history -f load-
    generator/locustfile.py

locust --host=http://swa-michaelmente.
    cs.univie.ac.at:32726 -u 1500 --
    spawn-rate=100 --headless --run-
    time=600s --csv benchmarks/
    noIngressGatewayHigh/$(date "+%d-%m
    -%y_%H-%M-%S") --csv-full-history -
    f load-generator/locustfile.py

locust --host=http://swa-michaelmente.
    cs.univie.ac.at:32726 -u 1500 --
    spawn-rate=100 --headless --run-
    time=600s --csv benchmarks/
    noLoadBalancingHigh/$(date "+%d-%m
    -%y_%H-%M-%S") --csv-full-history -
    f load-generator/locustfile.py
```

Depending on the machine the limit of open files needs to be increased beforehand to enable such high user loads which can be achieved with

```
ulimit -Sn 1048576
```

Benchmark used for verification of setup with Istio completely disabled

```
locust --host=http://swa-michaelmente.
    cs.univie.ac.at:32726 -u 1500 --
    spawn-rate=100 --headless --run-
    time=600s --csv benchmarks/noIstio/
    $(date "+%d-%m-%y_%H-%M-%S") --csv-
    full-history -f load-generator/
    locustfile.py
```