doi:10.1109/IISA52424.2021.9555536

# A Look at Service Meshes

Arne Koschel
Faculty IV, Department of Computer Science
University of Applied Sciences and Arts Hannover
Hannover, Germany
akoschel@acm.org

Marvin Bertram
Faculty IV, Department of Computer Science
University of Applied Sciences and Arts Hannover
Hannover, Germany
marvin.bertram@stud.hs-hannover.de

Richard Bischof
Faculty IV, Department of Computer Science
University of Applied Sciences and Arts Hannover
Hannover, Germany
richard.bischof@stud.hs-hannover.de

Kevin Schulze
Faculty IV, Department of Computer Science
University of Applied Sciences and Arts Hannover
Hannover, Germany
kevin.schulze@stud.hs-hannover.de

Marc Schaaf
Institute of Information Systems
University of Applied Sciences Northwestern Switzerland
Olten, Switzerland
marc.schaaf@fhnw.ch

Irina Astrova
Department of Software Science, School of IT
Tallinn University of Technology
Tallinn, Estonia
irina@cs.ioc.ee

*Abstract*—**Service meshes can be seen as an infrastructure layer for microservice-based applications that are specifically suited for distributed application architectures. It is the goal to introduce the concept of service meshes and its use for microservices with the example of an open source service mesh called Istio. This paper gives an introduction into the service mesh concept and its relation to microservices. It also gives an overview of selected features provided by Istio as relevant to the above concept and provides a small sample setup that demonstrates the core features.**

***Keywords—distributed systems, Docker, Istio, Kubernetes, microservices, service meshes***

## I. Introduction

Nearly every organization is driven by providing the best customer service. In the past this was mainly achieved by addressing the requirements of users with functional software features. But recently the focus shifted to non-functional requirements. Important aspects of today's software development and operation are key performance indicators like short time-to-market, high availability and elastic scalability. The fundamental soft skills were laid down by the Agile Manifesto [1], which covered four principles of work in the software development process and emphasized customer satisfaction.

Therefore, four principles of modern work have been introduced that aim on deploying new software features to customers as early as possible. The major technical step was the introduction of Cloud Computing [2] and the rise of public cloud service providers like Amazon Web Services, Google Cloud Platform and Microsoft Azure. Nowadays anyone can have access to high performance infrastructures, given only cheap client devices and internet connection. This evolution empowered anybody to build, run and offer services without the need of big initial investments. Based on this mindset and technology, modern project management methods like Scrum and DevOps have been developed and are widely adopted in the industry today. In most agile environments the traditional monolithic design of applications was too cumbersome. Therefore, approaches like Domain Driven Design (DDD) [3] helped to split monoliths into smaller pieces alongside bounded contexts. Those pieces were finally orchestrated and integrated in Software Oriented Architecture (SOA) via an Enterprise Service Bus (ESB) [4].

With the release of the container technology *Docker* in 2013, microservice architectures gained a new momentum. Containers were a kind of game changer because they enabled microservices according to the modern definition by Martin Fowler:

"[...] there are common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data" [5].

To run container-based microservice architectures in production grade environments, orchestrators like *Kubernetes* and *Apache Mesos* have been developed. The speed of innovation and agility led to extremely distributed and chaotic deployments. Adrian Cockroft described this scenario as "Death Star of Microservices" [6]. Fig. 1 shows a visualization of communication between microservices at *Netflix*.

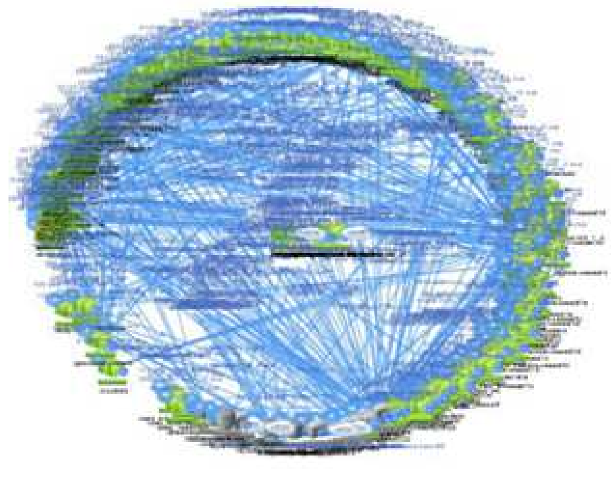

Fig. 1. Death-Star of Microservices at Netflix [6].

Obviously, it seems quite hard to run and maintain this architecture in an agile manner, while still having to reach high availability and reliability. Because of that, resilience frameworks like Netflix OSS (https://netflix.github.io/) have been developed and provide mechanisms like service discovery, circuit breaking and latency and fault tolerance. For popular programming languages and frameworks libraries are available, e. g., for Java and Spring Boot (https://spring.io/ projects/ spring-cloud-netflix).

Nevertheless, the approach of implementing this functionality into the microservices themselves has a number of disadvantages:

- Increased complexity of microservice code
- Limited choice of programming languages
- Low reusability of microservice artifacts
- Difficult collaborative development.

Service meshes are an approach to fix those points by extracting any infrastructure related code from the microservices into a dedicated layer.

## II. Fundamentals

### A. *Service Mesh Concept*

The service mesh layer is located between the microservice and orchestration layers. This stack is shown in Fig. 2. In the orchestration layer point of view, the service mesh is just another application that needs to be managed.

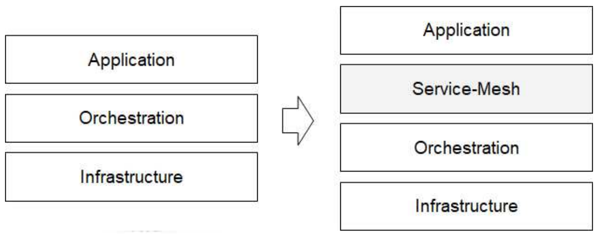

Fig. 2. Service mesh layer between application and orchestration.

The basic idea is to route all ingress and egress traffic from microservices through the service mesh layer. This way, the traffic can be observed and modified through the service mesh. The service mesh includes *Application Programming Interfaces* (APIs) that let it for an example integrate into logging platforms, telemetry or policy systems. Basically, a service mesh can provide features in four areas:

- Security (encryption, decryption)
- Traffic management (routing, canary deployments)
- Policies and telemetry (authentication, fault detection)
- Observability (logging, metrics).

For implementing service meshes seamlessly into microservice architectures the sidecar pattern is used. This is explained next.

### B. *Sidecar Pattern*

In the sidecar pattern all microservices are appended with so called “sidecars” that can carry out supporting tasks for the microservice. In a simple example of a webhosting microservice, sidecars can periodically update the webspace directory with content of a remote repository [7]. In the context of service meshes they proxy all network traffic. This scenario is shown in Fig. 3. The sidecar is independent configurable and can intercept and modify all packets.

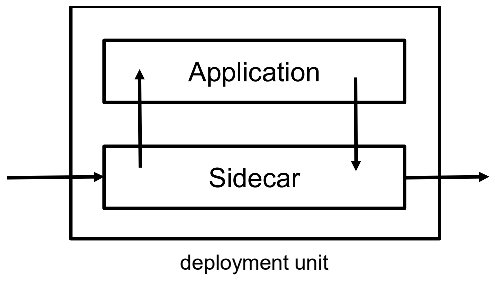

Fig. 3. Sidecar pattern.

This concept can be applied to various systems and is agnostic to specific technologies or products. It is applicable for environments that already use dedicated deployment units such as virtual machines or containers.

### C. *Dynamically Configurable Proxies*

The main requirement are proxy sidecars that can handle configuration changes at runtime. Common proxy servers like *Apache Httpd* and *Nginx* are not suitable. Instead *Istio* and other service meshes (https://www.hashicorp.com/products/consul/service-mesh/, https://aws.amazon.com/de/app-mesh/) use the *Envoy* (https://www.envoyproxy.io/) proxy, which can be configured at runtime and handle all sorts of Transmission Control Protocol (TCP) connections.

### D. *Docker and Kubernetes*

As mentioned before, service meshes can be used in different technology stacks. Because *Docker* and *Kubernetes* are used in this work, a short overview about those and their meaning in the context of service meshes are given next.

*Docker* is the de-facto standard for containers in cloud computing. It uses immutable software artifacts (images), which can be instantiated to running processes (containers).

*Kubernetes* is a framework for orchestration of containers. It abstracts multiple machines as one cluster, where workloads can be run as containers. It can be managed over an API that is accessed via kubectl Command Line Interface (CLI). The API is extensible over Custom Resource Definitions (CRDs) but contains a number of basic types. The most important types for this work are as follows:

- *Namespaces* are organizational units of resources in a cluster. They enable global configuration of resources inside a namespace as well as general network policies between namespaces.
- *Labels* are metadata for resources and enable dynamic selections of resources. For example, this is useful in scenarios like load-balancing, where you want to split traffic to dynamically changing group of containers. In the context of service meshes, namespaces can be labeled with specific attributes, so that sidecar containers get automatically injected in all contained Pods without any further configuration.
- *Pods* are deployment units of workloads and consist of one or more containers. Containers in a Pod share their network addresses, port space, hostname and can communicate via Inter Process Communication (IPC). Because of that, they are always deployed on the same cluster node. In the context of a service

mesh, Pods are used to deploy a microservice container along with its own sidecar proxy container.

- *Services* abstracts the access to services provided by Pods. Because Pods are scheduled to cluster nodes depending on current resource requisition, their IP addresses are permanently changing and not suitable for referencing. Instead, services define static identifiers and use labels and annotations to route traffic correctly.

## III. Example: Istio

### A. Architecture

The main architectural components in Istio are divided into so called Data Plane and Control Plane, each of which with a distinct responsibility. The planes themselves have additional parts that provide the actual functionality of Istio. Overall, Istio as a service mesh is designed to be independent of the underlying technology of the orchestration layer. Though the reference implementation and examples do prefer Kubernetes. The control plane consists of the four parts Mixer, Citadel, Galley and Pilot, whereas in the data plane everything is handled through a proxy [8] [9]. Fig. 8 shows an overview of Istio's architecture. Next, each part will looked in more detail.

*1) Data Plane:* The data plane handles the ingress and egress network traffic. This functionality is implemented transparently to the actual application and enables the addition of further capabilities without the need of implementing it in the application itself. As mentioned, Istio uses Envoy as proxy. Envoy itself is specifically designed to be used in service-oriented architecture-type applications and is placed alongside them. Kubernetes auto-injects the Envoy proxy inside a microservice Pod and configures the Pods routing tables. The overall flow of communication is shown in Fig. 4. This design leads to two advantages: First, the language of the application is no longer relevant. That means the services in a mesh can be implemented in a polyglot set of languages and all the communication is handled by Envoy. Second, Envoy can be upgraded and deployed in a transparent way with help of library upgrades. [10]

Features and technologies that are supported by Envoy include gRPC (https://grpc.io/), local balancing, HTTP1.1, HTTP2, metrics aggregation, health checking and more (https://www.envoyproxy.io/). Furthermore, Envoy is written in C++, which should be beneficial for performance.

*2) Control Plane:* The role of the control plane is to act as a source from where policies and the configuration is pushed to the data plane. Besides the three main components Mixer, Pilot and Citadel, Galley acts as the more top-level part for the Istio configuration. It shields the other components of specifics of getting the actual configuration from the platform (e.g., Kubernetes) that lies beneath [11].

***Mixer:*** The main responsibility of Mixer consists of communication with the Istio proxies to get telemetry data, to push policies, to apply custom metrics, to perform access control checks and to deploy rate-limiting rules [9].

The architecture of Mixer is shown in Fig. 4; it is designed to abstract the aforementioned functionality from the services and the rest of the Istio components. It also provides a plugin-model with Mixer as an intermediary. Plugins in this model are called adapters. Through the adapter API different backend system can interact with Mixer. Prior to forwarding a request to the service, the Envoy sidecar proxies communicate with Mixer to check if rules apply. They also obtain telemetry data after the request. The Envoy sidecar proxies have caching and buffering functionality so that Mixer is not called frequently.

Mixer's main configuration mechanism are attributes. These attributes describe individual requests as well as the environment around them. They are comprised of a type and name. They describe, e.g., the origin IP address of a request and response codes. The main source of those attributes is the Envoy proxy even though the adapters plugin model allows for further attribute data. Communication is then initiated to the different backends after the attributes are handled by Mixer [8].

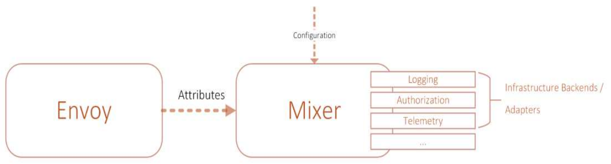

Fig. 4. Mixer (source: https://istio.io/docs/reference/config/policy-and-telemetry/mixer-overview/).

***Pilot:*** The traffic management responsibility lies in the Pilot component of Istio. Pilot is organizing and managing the sidecar proxies. When a change occurs in the topology all sidecars are informed by Pilot. That means that each of the sidecars can adjust their information about available microservices and how to route traffic. Further features are service discovery and more detailed traffic routing [9].

***Citadel:*** It handles everything related to security and certificate distribution, signing and revocation. Due to the certificates, mutual Transport Layer Security (mTLS) can be used for identity and encryption between the microservices. All the traffic that traverses between the microservices is therefore encrypted transparently without changes to the actual application [9].

***Galley:*** Istio can be used with different platforms like Kubernetes. Which means that the configuration has to be abstracted from the platforms. Galley is the component, which provides this functionality. Moreover, it validates the configuration and is responsible for distributing it to the respective components inside Istio [8].

### B. Security

Security nowadays is a crosscutting concern in modern application development lifecycles and therefore also an important requirement in microservices deployments. Threats are not only coming from outside the service mesh but can also occur inside the mesh. Therefore, all the traffic in the mesh is automatically not trusted, which is called a zero-trust approach. Because of that assumption, different requirements arise. One of the foundations of every security concepts are strong identities. Istio provides means to assign identities to services as well as to end users/devices. The service identities are based on the underlying platform, in case of Kubernetes the Kubernetes service accounts are used. But if the platform itself does not provide this functionality, Istio can interface with for example AWS IAM user/role accounts, GCP service accounts and others (see [8]).

For user/device identities and authentication different providers can be used via OAuth (https://oauth.net). Furthermore, with the use of JWT (JSON Web Tokens) authentication can be broken down to the request level [8].

Overall Istio provides support for the typical security properties:

- Authentication
- Authorization
- Auditing.

***mTLS:*** mTLS stands for mutual Transport Layer Security and is used for actual authentication and encryption. It helps protecting against replay and man-in-the middle attacks. Istio uses standard X.509 certificates. The identities in the certificates are described in the SPIFFE (https://spiffe.io/) format. In Kubernetes storing the certificates and keys is done using Kubernetes built-in secrets. If certificates are no longer valid or revoked, Citadel will update and overwrite the Kubernetes secrets. Pilot pushes configuration to Envoy about which of the service accounts may run a specific service. In the authentication process one Envoy proxy is acting as the client and establishes a TLS handshake with the "server" side Envoy. To make sure that the server side is actually allowed to run the specific service, there exists a mapping, called secure naming, from the identity in the certificate to the name of the service in the DNS. The "client" checks this secure naming information and continues with establishing the TLS connection or aborts it altogether [8].

***Authentication Policies:*** Enabling mTLS can be done on a variety of levels (mesh, namespace, service) using an authentication policy. In general, such policies are checked whenever a service receives a request. It is also possible to configure rules regarding mTLS for traffic going to a service. This is done via *Destination Rules* [8].

Configuring mTLS for a specific service can be done in the following way:

```
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "servicename"
spec:
  targets:
    - name: service
  peers:
    - mtls: {} # STRICT mTLS
```

Instead of "Policy" there also exists a "MeshPolicy" for mesh configuration. Namespaces can be configured by adding "namespace" to the metadata and leaving out the target. The default mode of mTLS is *STRICT* meaning only mTLS connections are allowed and therefore encrypted traffic. Beside *STRICT* there is also a mode called *PERMISSIVE*. This mode also allows for plaintext traffic [8].

***Authorization:*** As the case with the authentication policies, authorization policies can be configured on a mesh, namespace and service level. On Kubernetes, Istio provides a single custom resource for the definition: AuthorizationPolicy. Rules can be defined to service-to-service and/or end-user-toservice communication. Policies are evaluated within Envoy at runtime via a specialized engine. After a request is examined the engine reports back ALLOW or DENY as a result and therefore grants or prohibits access to a resource. One can distinguish three cases:

- Authorization policy, which allows access
- No information in the authorization policy but applied. That leads to a deny all
- A simple allow which as the name suggests grants access to a resource.

The basic properties of an authorization rule are the question who is going to do something, which conditions have to be met in order to do it and what are they going to do. The rule definition allows for specifying exactly that. The who, what and condition parts have corresponding configuration sections in the policy, namely: from, to and when. Additionally, a selector section can be used to define the target resource in the service mesh [8].

A sample to deny all authorization rule for a service would look as follows:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: DenyAll
  namespace: mynamespace
spec:
  {}
```

This "DenyAll" rule uses an empty definition which leads to traffic being blocked. The next listing shows a rule where services from the namespace mynamespaceB are allowed to access services with the label myapp for version v1 in namespace mynamespaceA. But the only action that is allowed is the GET operation.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: authrule
  namespace: mynamespaceA
spec:
  selector:
    matchLabels:
      app: myapp
      version: v1
  rules:
  - from:
    - source:
      namespaces: ["mynamespaceB"]
    to:
  - operation:
      methods: ["GET"]
```

Configurations that would better fit certain security requirements are also possible.

## C. Traffic Management

Every microservice system needs some way to route requests between microservices and route a request from the outside to the correct microservice. One of the core tasks of Istio is the intelligent management of calls between services in the service mesh. In Istio traffic rules are specified via Pilot. Pilot is the core component for traffic management and consists of four components. The rule API is the point for specifying high-level traffic management. The platform adapter implements the necessary controllers for the used platform (e.g., Kubernetes). The abstract model hides all the platform specific functions and the Envoy API is the interface for the communication with other sidecars. [8] An overview of the Pilot architecture can be found in Fig. 5.

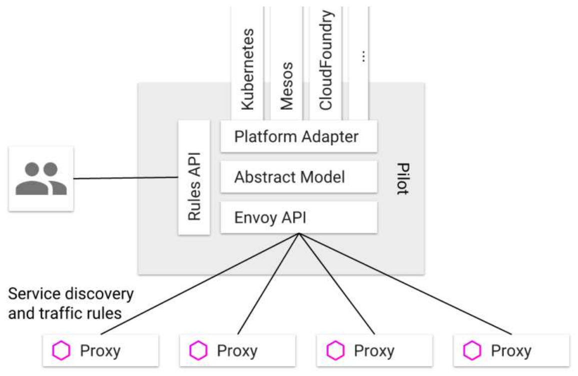

Fig. 5. Pilot Architecture [8].

It is only necessary to specify which call behavior is wanted without having to specify which Kubernetes Pod or VM should process these calls. The call behavior and the assignment of the calls to the runtime instances is handled by pilot and Envoy. Istio's traffic management features are:

- Circuit breakers
- Timeout and retries
- A/B testing
- Canary rollouts.

An interesting feature is the content-based traffic steering. As shown in Fig. 6, it is possible to direct traffic to specific versions based on the device (Android/iOS) which makes the request.

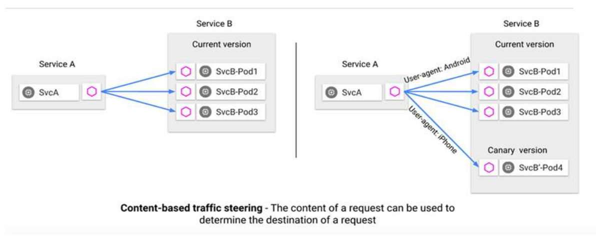

Fig. 6. Content-based traffic steering [12].

## D. Policies and Telemetry

With Istio it is possible to configure custom policies for an application. Policy management includes access controls (blacklists and whitelists), rate limits and quotas. The Mixer component is responsible for providing the policy controls and also does telemetry collection. To achieve this functionality all service-sidecars are calling the mixer before each request to perform precondition checks as well as after each request to report telemetry data. The sidecar has local caching so that precondition checks can be performed from cache. Also, the sidecar buffers outgoing telemetry so that it only calls Mixer infrequently and reduces the network traffic (first level pre-sidecar cache). The second level shared cache in the Mixer collects and buffers incoming data from the connected sidecars. The Adapter API of Mixer abstracts the details of different policy and backend system so that it is possible to deal with different infrastructure backends for logging, quotas, and telemetry [12]. Mixer's topology shows Fig. 7.

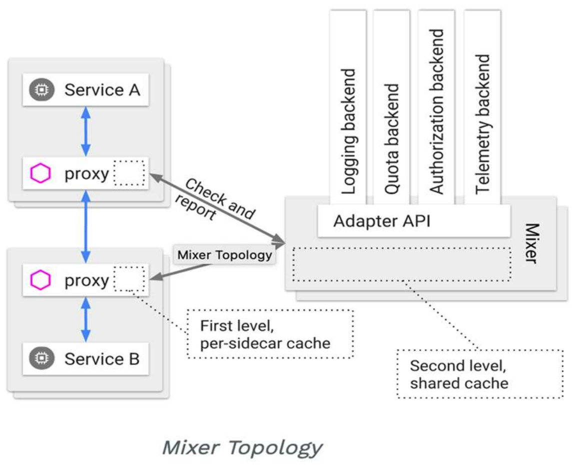

Fig. 7. Mixer Topology [12].

## E. Observability

Observability is a vital role in a service mesh especially when it comes to "debugging" and this is the main reason why people choose to adopt to a service mesh. Because in an complex environment it is crucial to know what happens in your system [13]. In Istio it is achieved by:

- Tracing
- Logs
- Metrics.

*1) Tracing:* If problems occur in distributed systems, it is often difficult to find the underlying cause. Troubleshooting within a service mesh therefore regularly presents a major challenge for development and operation. A request can traverse the mesh on different paths over several service calls. Problems can occur at different points in the process, and it can be difficult to find the reason of timeouts and latency problems without appropriate support. It is also difficult to detect error cascades without a suitable tool. Moreover, the amount of services in a big application can easily be reach a point where it is not obvious how everything interacts with one another [9]. Istio takes distributed tracing tools like Zipkin or Jaeger into account. For this purpose, Istio collects all necessary information via the sidecar and transfers it to the configured tracing system.

Besides the direct sidecar-to-tracing system communication it is also possible for Mixer to integrate with a tracing backend system and the collection. This gives more control over the configuration since not all backends are supported by the sidecar proxy Envoy [8].

Tracing depends on correlating the requests coming into the sidecar to the requests leaving the sidecar. For that to work properly, Istio is expecting certain HTTP headers to be present if HTTP is used to service communication. Subsequently, this adds a requirement to the microservices themselves, if tracing is necessary, the services have to add headers. Therefore, tracing is not transparent [14].

*2) Logs:* Logging in a service mesh is limited to network traffic because otherwise changes in the microservice themselves would be necessary. Istio can configure exactly what has to be logged and also the format of the logs being generated. For example, it can include HTTP service codes [14].

A basic form of logging is the one provided by Envoy. The proxy just sends access logs to the standard output. If Kubernetes is used, the logs can be examined using kubectl [8]. In addition to that, Mixer allows for a more fine-grained control of the logging configuration. Therefore, CRDs like rule instance and handler can be used.

Log entries are defined by an instance which is basically a template. In the instance all Istio attributes that should be part of a log entry are listed.

```
...
kind: instance
metadata:
  name: mylog
spec:
compiledTemplate: logentry
params:
severity: '"warning"'
timestamp: request.time
variables
...
user: source.user | "unknown"
```

A handler tells Istio how the log entries as instances are being handled. For example, should the output be in a JSON format. It also includes the desired levels.

```
...
kind: handler
metadata:
name: myloghandler
...
spec:
compiledAdapter: stdio
params:
severity_levels:
warning: 1
outputAsJson: true
```

Rules define which traffic in the mesh should be logged. A rule also glues the handler and instance together.

```
...
kind: rule
...
spec:
match: "true"
actions:
- handler: myloghandler
instances:
- mylog
```

The above examples are adapted from the Istio bookinfo samples [8]. A more sophisticated approach would be to forward the logs to a system with further analysis functionality like Elasticsearch.

*3) Metrics:* Istio can collect a variety of metrics about the network and services. These metrics include information about errors, latency and traffic volume which can be used to monitor the overall health of the service mesh [8]. For basic network metrics the Envoy proxies can automatically provide the necessary information. To get more detailed metrics from the network requests, Istio needs to know about the protocols involved in a communication. For example, errors rates can be deducted from HTTP status codes. Besides the Envoy metrics, additional ones are provided between services and from the components of the control plane [8]. Moreover, if the underlying platform can provide metrics this can also be used inside Istio [14].

The configuration of metrics collection is similar to the one regarding the logs. An instance, a handler and a rule is needed. The values for the metric are defined in the instance. Mixer gets attributes from itself and Envoy. A handler configures mappings between the instance to a format which can be understand by a backend system like Prometheus (https://prometheus.io/). And the last part defines a rule for the kind of traffic where metrics should be collected [8].

## IV. Experiments

Some functionality was tested using the sample BookinfoApp provided by Istio. The web app is composed of different microservices and quite simple. One product service written in Python, a reviews service with different versions written in Java, a details service written in Ruby and a ratings service written in NodeJs. To have some real-world deployment setting, the Google Cloud Platform (GCP) was used. First, a four-node cluster was setup. Second, Kubernetes as underlying platform was deployed. Third, within the CloudShell of GCP, Istio was installed and sidecar injection for the default namespace enabled. The goal was to understand and test different aspects of a service mesh, specifically the observability and traffic routing functionality. Regarding observability Istio can interface with a variety of tools and the one used in the experiment is called Kiali (https://kiali.io/). Kiali has features for viewing different metrics and visualizing the topology of the mesh. Furthermore, it has the capability to configure basic traffic management rules and validating them.

### *A. Scenario*

The main scenario was about routing traffic to the various versions of the reviews page depending on the user agent of the client calling the BookinfoApp. The difference among the versions of the reviews page is simply the fact that version one does not display review stars and is not calling the rating service, version two has grey stars and version three has red stars on the page. In conjunction with the user agent headers the following cases arise:

- iOS users with respective agent will be served reviews page version 1
- Android users will be served version 2
- Windows users will get version 3
- Everyone else will also get version 3.

### *B. Implementation*

The scenario was implemented using the two Kubernetes CRDs virtual service and destination rule. The virtual service is responsible for routing the traffic to a service and uses definitions from the destination rule. The destination rule is applied after the traffic is routed. In this case the destination rule is used to define subsets that comprise the review service. The excerpt of the virtual service looks as follows:

```
...
kind: VirtualService
  name: reviews-virtualservice
  namespace: default
spec:
  hosts:
    - reviews
  http:
  - match:
    - headers:
      user-agent:
        regex: .*iPhone.*
    route:
    - destination:
      host: reviews
      subset: version-v1
...
  - route:
    - destination:
      host: reviews
      subset: version-v3
```

Important is the “match” section for the http header and user agent. A regex is applied when a request is bound for the reviews service. When a match is true the request will be forwarded to the destination service with desired version. If nothing matches the last route is used. The rules for Android and Windows user agents look like the one for iOS and are therefore omitted. The subsets in the virtual service are a direct reference to the destination rule:

```
...
kind: DestinationRule
metadata:
name: dr-reviews
namespace: default
spec:
host: reviews
subsets:
- labels:
version: v1
name: version-v1
...
```

This rule defines three subsets for the reviews service. Again, the last two were omitted because they are similar. The name that was chosen in the rule is exactly the one in the virtual service. Therefore, both the rules work together to accomplish proper traffic management. To finally test everything two bash scripts were created that simulate requests from different user agents. The scripts contain a while loop with a call to the public facing IP of the cluster. The request is made via curl which has the “-A” switch for setting the user agent. One user agent in the scripts was then set to iOS users and the other one to Android users. While starting the requests, Kiali was simultaneously observed to see the traffic being routed through the mesh.

### *C. Results*

It was easy to setup Istio with Kiali on the Google Cloud with Kubernetes. Moreover, configuring basic traffic rules without changing anything in the microservices sample app worked out to be surprisingly straightforward. The visualization of the mesh in Kiali was also not a difficult task since the mesh is sending the required data automatically. An overall good experience with expected results.

## V. Conclusion

The service mesh concept is the result of an evolution in the software development process. Beginning with the Agile Manifesto and the introduction of Cloud Computing as well as public cloud platforms like AWS, GCP or Azure to container technology like Docker. State of the art are today’s microservices. But microservices have problems as well. Coupling infrastructure code with specific framework dependencies into microservices leads to an increased complexity of code and limited choice of programming languages.

Section II focuses on the fundamentals for a service mesh. The service mesh as an additional abstraction layer, the sidecar pattern for the microservices and container orchestration with Kubernetes were explained. Then the service mesh Istio was introduced. Showing the main architectural components Data Plane and Control Plane and the more fine-grained components Citadel, Mixer, Galley and Pilot. Each component has special functionalities which are explained on the basis of the four features Istio can provide: security, traffic management, policies and telemetry and observability. These four features were further analyzed by using the BooInfoApp provided by Istio. The setup includes GCP, a four-node cluster, Kubernetes and Kiali for viewing metrics and visualizing the topology of the mesh. The scenario was routing traffic to different versions of the review page depending on the user. It was possible to differentiate between iOS, Android, Windows and all other user of the application.

The needed steps for this implementation are explained in Section IV. The service mesh is an additional abstraction level in microservice environments, which can simplify administration and allows flexibility.

A service mesh can simplify the application architecture because it decouples the application from the infrastructure code. This is possible because a service mesh uses the sidecar pattern. So, the development can focus on the domain weather the deployment environment. The use of a sidecar makes a service mesh language independent so there is consistency across all of the used microservices and no constraints in what programming language can be used. In exchange a service mesh adds a higher overall complexity to the running system. Having a service mesh increases the number of runtime instances in a system. It also adds extra hops for communication because each service call has to go through a service mesh sidecar proxy. Furthermore, it looks like it is only useful in highly automated environments because it is the only place it can show its advantages for security, traffic management and observability.

## References

[1] K. Beck et al. Manifesto for Agile Software Development. 2001. URL: http://www.agilemanifesto.org/

[2] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Tech. rep. 800-145. Gaithersburg, MD: National Institute of Standards and Technology (NIST), Sept. 2011. URL: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[3] E. Evans and M. Fowler. Domain-driven Design: Tackling Complexity in the Heart of Software. AddisonWesley, 2004. ISBN: 9780321125217. URL: https://books.google.de/books?id=7dlaMs0SE CsC

[4] D. Chappell. Enterprise Service Bus. O’Reilly Media, Inc., 2004. ISBN: 0596006756.

[5] M. Fowler. Definition of Microservices. https://martinfowler.com/ articles/microservices.html. (Accessed on 01/15/2020). Apr. 2014.

[6] A. Cockcroft. Migrating to Microservices. https://gotocon.com/dl/ goto-berlin-2014/slides/AdrianCockcroftMigratingToCloudNativeWithMicroservices.pdf (Accessed on 01/15/2020). Nov. 2014.

[7] B. Burns. Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. 1st. O’Reilly Media, Inc., 2018. ISBN: 1491983647.

[8] Istio. https://istio.io/. (Accessed on 10/24/2019).

[9] B. Sutter and C. Posta. Introducing Istio Service Mesh for Microservices. O’Reilly Media, Incorporated, 2019. ISBN: 978-1-492-05260-9.

[10] What is Envoy? https://www.envoyproxy.io/docs/envoy/v1.12.0/intro/ what is envoy. (Accessed on 11/15/2019).

[11] Istio Github. github.com/istio/istio/tree/master/galley. (Accessed on 12/01/2019).

[12] Istio Documentation 1.0. https://archive.istio.io/v1.0/ (Accessed on 01/09/2020).

[13] Z. Butcher. Practical Istio (Docker Con’19). https:// www.youtube.com/watch?v=uRXzRfthYeU. (Accessed on 01/12/2020). May 2019.

[14] H. Prinz and E. Wolff. Service Mesh – The New Infrastructure for Microservices. innoQ Deutschland GmbH, 2019. ISBN: 978-3-9821126-1-9.

[15] Eine Einfuhrung in Istio: Keine Angst vorm Service-Mesh bei Microservices-Architekturen - JAXenter. https: //jaxenter.de/istio-einfuehrung-microservices-cloudteil-1-71261 (Accessed on 10/24/2019).

[16] L. Calcote and Z. Butcher. Istio: Up and Running: Secure, Manage, and Connect Your Microservices with Service Mesh. O'Reilly Media, Incorporated. ISBN: 9781492043782.

[17] Bornkessel et. Prinz. Alle 11 Minuten verliebt sich ein Microservice in Linkerd heise Developer. https://www.heise.de/developer/artikel/Alle-1-Minuten- verliebt- sich- ein-Microservice-in-Linkerd-4511406.html (Accessed on 10/24/2019). July 2019.

[18] Dino Chiesa and Greg Kuelgen. APIs, Microservices, and the Service Mesh (Cloud Next'19). https://www.youtube.com/watch?v=IblDMVwSSk4 (Accessed on 11/11/2019). Apr. 2019.

[19] M. O'Keefe. Istio in Production: Day 2 Traffic Routing (Cloud Next'19). https://www.youtube.com/watch?v=7cINRP0BFY8 (Accessed on 11/11/2019). Apr. 2019.

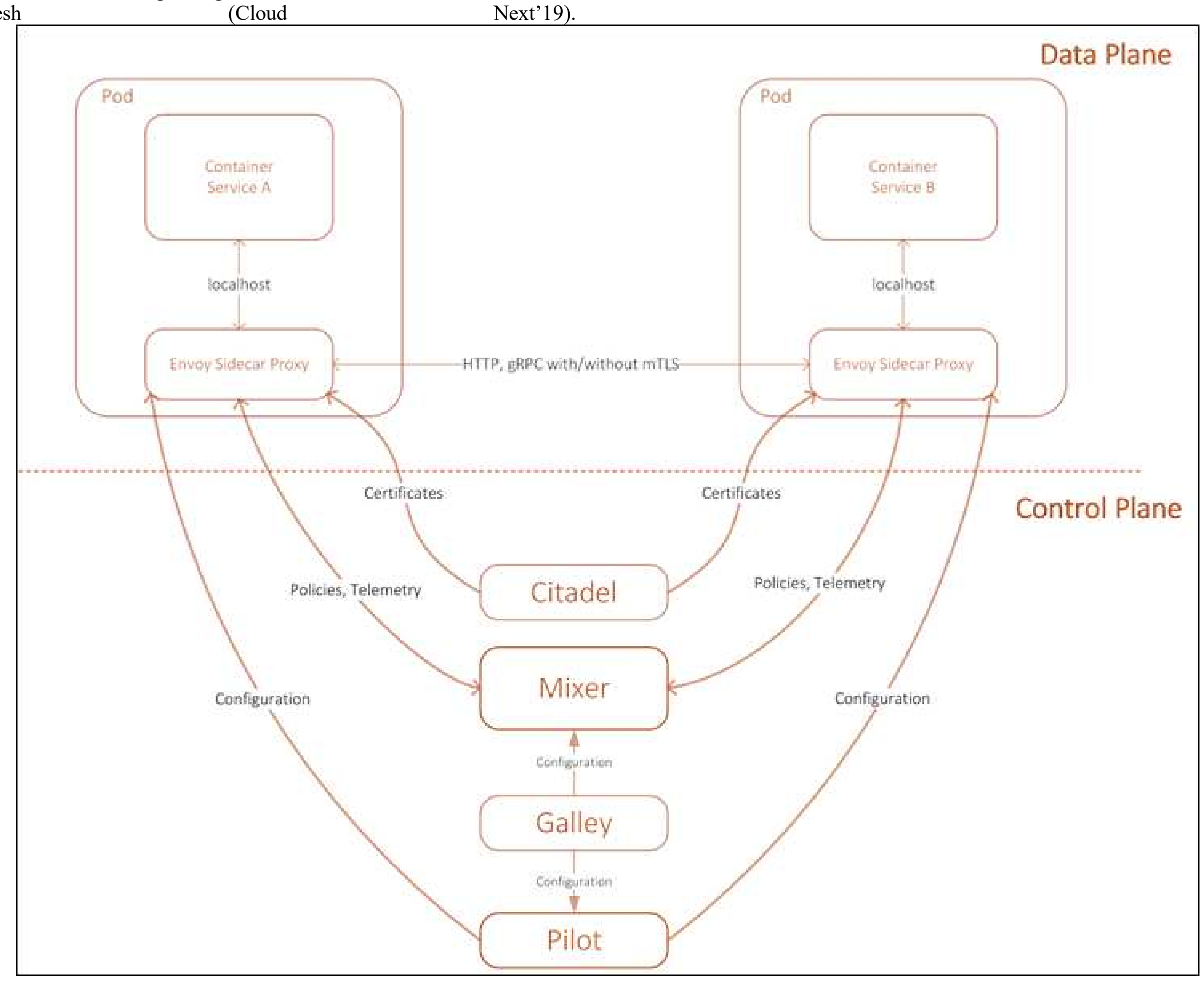

Fig. 8. Istio architecture overview [12].