Improving Microservice Reliability with Istio

Will Zhang
University of British Columbia, Vancouver, Canada
will.chi.zhang@gmail.com

1 INTRODUCTION

Microservice-based application architecture is an increasingly popular technique for building software, especially suited for frequent releases, scaleable, and decentralized applications. The function calls that connect each component of a monolithic application are converted to network requests. This conversion allows modules of an application to be implemented using different programming languages, but it also creates chaos of network traffic inside an application. This raises the concern of the reliability of a microserivce application being shipped, because it is extremely difficult to analyze network flow and trace where the issue is unless developers in code provide detailed logging and govern inter-service communication. This report provides the definition of reliability to better understand this concern of microservice systems and discusses aspects that improve their reliability. Although tracing and controlling network traffic are achievable in the implementation, it requires extra efforts during development. This report introduces Istio, a service mesh, to observe and control the network communication within an application without requiring code changes. Next, a detailed analysis is provided to show how Istio can improve microservice reliability along with a case study. Finally, we discuss the limitation of Istio and service mesh.

2 RELIABILITY

In this section, we discuss the definition of software reliability in both academic and the industry and demonstrate design patterns of building more reliable microservices.

2.1 Definition

In academic research, *software reliability* (SR) is defined as “the probability of failure-free software operation for a specified period of time in a specified environment” [18]. Software failure is the transition from *correct service* (i.e. with expected service responses) to an *incorrect service* (i.e. failed requests or unexpected service responses). Researchers build up models of software systems to evaluate reliability. However, the models are platform-dependent, and the main focus of this report is on microservice [14], so the reliability models will be skipped. Apart from modeling, software testing is the main approach to reliability assessment. For microservice, since the software and the operational profile change continuously due to the frequent releases, service upgrades, dynamic service interactions, researchers also proposed a run-time testing method to estimate the reliability of the microservice application, which monitors the failure rate upon requests to a specific service instance [21].

The term *software availability* (SA) is also used in research papers and blog posts, sometimes, interchangeably with SR. SA is a description of SR from customers’ viewpoints, and it is defined as "a software-intensive systems are available at a given time point, under the specified environment" [29]. SA is also considered as “the reliability for customers” or “the reliability with maintainability” [29]. It is worth noting that the correctness of a request does not reflect on SA. In other words, users are less anxious when their banking website shows maintaining than a "page not found" error. Both SA and SR only care about acknowledging software failures, and correcting failures are tasks for developers. The industry often uses SR when referring to SA, because a customer-oriented measurement can better fit in their engineering goal. This report does not distinguish the two terms.

In the industry, Google’s view on SR is an approach called Site Reliability Engineering (SRE)[1], which is "a discipline that incorporates aspects of software engineering and applies them to infrastructure and operations problems" [30]. SRE assumes that in nearly all systems there’s a very small (but nonzero) acceptable quantity of unavailability. The system downtime can be thought of as an error budget. As long as a system is down less than its budget, it is considered healthy. For example, if a system is required to be available 99.9% of the time. That means it’s acceptable for the system to be unavailable 0.1% of the time (for any given 30-day month, that is 43 minutes). Once you blow the error budget (99.9%), however, you need to spend 100% of your engineering time writing code that fixes the problem and generally makes your system more stable [23]. In SRE, failure is no longer an unexpected event. This is fundamentally different from SR definitions in academic where the probability of a failure-free system is no longer important, and yet the system downtime is all we care about. Furthermore, Google defines SR as "a function of mean time to failure (MTTF) and mean time to repair (MTTR)", so we only care how often we encounter failure and how efficient the response team can resolve the failure [7].

AWS[2] formally defines SR as "the percentage of time that an application is operating normally", mathematically written as:

$$Availability = NormalOperationTime/TotalTime$$

For a given time period (i.e. Total Time), the status of the system is flipping between available and unavailable (i.e. repairing). We can estimate availability by calculating, the percentage of average system available time in both average system available and unavailable time. The normal operation time can be considered as the average time between failures, We define MTBF to be Mean Time Between Failure, then the estimation of availability can be described as:

$$AvailabilityEstimate = MTBF/(MTBF + MTTR)$$

When a service depends on other services (e.g. dependency 1, dependency 2, etc.), the availability is calculated to follow the probability

[1] The book [8] is highly recommended for all operation team.
[2] Amazon Web Services

© 2020 Association for Computing Machinery.

product rule (e.g. service * dependency 1 * dependency 2...). When redundant components exist, however, the availability is generally improved because the probabilities of all of the components going down at the same time are lower. Given N instances of a component, the availability is formally defined as:

$$Availability = 100\% - \sum_{i=1}^{N} AvailabilityOfN$$

Availability	Max Disruption (per year)	Application Categories
99%	3 days 15 hours	Batch processing, data extraction, transfer, and load jobs
99.9%	8 hours 45 minutes	Internal tools like knowledge management, project tracking
99.95%	4 hours 22 minutes	Online commerce, point of sale
99.99%	52 minutes	Video delivery, broadcast systems
99.999%	5 minutes	ATM transactions, telecommunications systems

Figure 1: Common application availability design goals and the possible length of interruptions that can occur within a year while still meeting the goal from AWS [2]

For a software system to have good availability, AWS considers that the system should have the ability to recover from infrastructure or service disruptions, dynamically acquire computing resources to meet demand, and mitigate disruptions such as misconfigurations or transient network issues [2].

To summarise, in academic, SR is evaluated by the probability of a failure-free system, in other words, the prediction of future failures is the key to evaluate SR. In the industry, the probability of a single failure is ignored, and SR is evaluated by the availability of the software system. Although the two measurement methodologies are different, they are two dimensions to SR. The higher the probability is, the less unavailable the software system is. Note that, the discussion above is not restricted to any specific software design.

2.2 Design Principles of Reliable Microservice Applications

The definition of reliability emphasizes on understanding and evaluating software reliability. However, in practice, people care about how to determine whether a system is available, how to recover from failures if a system becomes unavailable, how to improve system resilience, and how to discover issues before deployment.

2.2.1 System Monitoring - Determine System Status. As mentioned earlier, SRE treats failures as expected events during operation, then discovering the failures and knowing the system status becomes crucial, and for monitoring the application, SRE considers the four golden signals that can answer basic questions about the application [8]:

Traffic a measure of how much demand is being placed on the system.

Errors requests failure rate, which can be explicit (e.g., HTTP 500s), implicit (for example, an HTTP 200 success response, but coupled with the wrong content), or by policy (for example, "If you committed to one-second response times, any request over one second is an error").

Latency the time it takes to service a request. It's important to distinguish between the latency of successful requests and the latency of failed requests. For example, an HTTP 500 error triggered due to loss of connection to a database or other critical backend might be served very quickly; however, as an HTTP 500 error indicates a failed request, factoring 500s into the overall latency might result in misleading calculations.

Saturation how "full" your service is. A measure of your system fraction, emphasizing the resources that are most constrained (e.g., in a memory-constrained system, show memory; in an I/O-constrained system, show I/O). Note that many systems degrade in performance before they achieve 100% utilization, so having a utilization target is essential.

Besides, it is also suggested to implement a health check API to quickly validate the status of a service and its dependencies. In other words, a health check API is a separate REST service that is implemented within a microservice component that quickly returns the operational status of the service and an indication of its ability to connect to downstream dependent services [1].

2.2.2 Recovery-Oriented Computing - Recover from Failures. Reducing MTTR is critical to improve software availability. There exists a concept called Recovery-Oriented Computing, which emphasizes recovery from failures rather than failure-avoidance for a software system [25]. It mainly focuses on the follow aspects:

Isolation and Redundancy the ability to isolate portions of the system. Isolation is crucial for fault containment and safe online recovery, and it naturally demands redundancy, as redundancy allows continued service delivery while portions of the system are isolated.

System-wide Support for Undo provides an undo facility that covers all aspects of system operation, from system configuration to application management to software and hardware upgrades.

High Restartability and Modularity "Software aging" related problems, such as memory arena corruption, very complicated and difficult-to-reproduce timing-related concurrency bugs, etc. are often best solved by a total or partial restart of the affected components. In some cases, proactively restarting components before they fail can improve overall availability. Naturally, high restartability requires high modularity, so we can restart seprate components instead of the whole software system.

2.2.3 Fault Tolerant - Improve System Resilience. Designing robustness software architecture is the key to improve system resilience. There are several design patterns/principles suggested by Microsoft Azure [3], AWS [2], and Microservice.io [19] to make the system fault tolerant[3]:

[3]There are more design patterns available, but only listed the ones that are more related to microservice

Retry enables an application to handle anticipated, temporary failures when it tries to connect to a service by transparently retrying an operation that's previously failed.

Queue-Based Load Leveling the queue, which acts as a buffer between a task and a service. It invokes decouples the tasks from the service, and the service can handle the messages at its own pace regardless of the volume of requests from concurrent tasks.

Circuit Breaker prevents an application from performing an operation that is likely to fail by automatically rejecting new requests if the requests have exceeded the service capacity, so it takes less time to encounter failures and allow the service to cool down.

Bulkhead elements of an application are isolated into pools so that if one fails, the others will continue to function.

2.2.4 Testing - Discover Issues Early. It is also suggested to use automation to simulate different failures or to recreate scenarios that led to failures before.

Canarying release refers to a partial and time-limited deployment of a change in a service and the evaluation of this deployment. Canarying allows the deployment pipeline to detect defects as quickly as possible with as little impact to your service as possible. The canary process also lets us gain confidence in the change as we expose it to larger amounts of traffic. Introducing the change to actual production traffic also enables to identify problems that might not be visible in testing frameworks like unit testing or load testing, which are often more artificial. When you catch system defects early, users are minimally impacted. Canarying can also provide confidence in frequent releases and improve reliability [8].

In complex systems, chaos engineering takes the approach that regardless how encompassing your test suite is, once your code is running on enough machines and reaches enough complexity errors are going to happen [9], and chaos testing is then defined as deliberately introduce errors to ensure your systems and processes can deal with the failure. This is an important testing strategy to ensure the simulation of failures to the system, and it is extremely useful when assessing the Mean Time to Recovery (MTTR).

3 SERVICE MESH AND ISTIO

In this section, we explain service mesh, an architectural paradigm, and how is it related to microservice architectures. Then, we introduce Istio, an open-source project that delivers a service mesh to a microservice application.

3.1 Service Mesh

Service instances of a microservice application communicate through a network; service mesh handles these service-to-service communications to resolve the problems of connecting, securing, controlling, and observing a mesh of services. Formally, the *service mesh* is defined as an architectural paradigm that provides a transparent and language-independent way to flexibly and easily automate microservice application network functions [11]. The transparent way means that the application code base must not be affected by adding in a service mesh and developers can safely ignore its existence. The language-independent way means that a service mesh is independent of service implementation. Therefore, a service mesh is an infrastructure layer that is designed to handle the inter-service communication of a microservice application.

Although the logic that governs inter-service communication can be coded into each service without a service mesh layer, adopting a service mesh allows the application to be decoupled from the network stack and becomes beneficial as the application complexity increases [22]. A service mesh focuses on helping the following four aspects of a microservice application [11]:

Connect controls network traffic flow and application program interface (API) calls between service instances. Some useful features in this aspect including: *load balancing*, *retries*, *health check*, *canary releases*, and *circuit breaker*.

Secure manages and secures communications between services. These features include authentication, authorization, and firewall.

Control applies user-defined policies (for example, routing, rate limits, quotas) and enforces them across services.

Observe provides insights on performance and distributed traffic flow. For example, improve visibility to monitor service status and detailed logs to quickly identify issues.

As mentioned earlier, the service mesh is an architectural paradigm, or in other words, an infrastructure design pattern. So, how is a service mesh implemented? A service mesh is constructed with two components:

Data Plane handles network traffic between service instances. Typically, this is implemented through deploying a network proxy, called a *sidecar proxy*[4], alongside each service instance [27]. The sidecar proxy intercepts and rules network traffic flow coming in and out of a service instance, so developers can focus on achieving service core logic without governing inter-service communication. On the one hand, by analyzing and tracing packets on the network layer, the sidecar proxy improves microservice application observability and traceability. On the other hand, by controlling requests to a particular service instance, the sidecar proxy can prevent service from overloading and easily respond to user-defined network behaviors. Regardless of service implementation, network communication will be the outcome upon a service request, and the data plane only interacts with the network stack. Hence, the data plane makes service mesh transparent and language-independent of a microservice application.

Control Plane deploys user configuration and controls the data plane's behaviors. It usually provides an interface for the user to configure the service mesh and manage network traffic within the microservice application. Conceptually, the control plane is the central unit that connects the user's policies to sidecar proxy behaviors. It interprets and validates each user-defined policy (e.g. retry policy), and synchronizes the rules for all proxies as user specifying new policies dynamically. Additionally, when adopting a service mesh, the

[4] Some blogs [17] claim sidecar proxy is the data plane of service mesh

control plane is responsible for interacting with the underlying infrastructure that the microservice application built on [17].

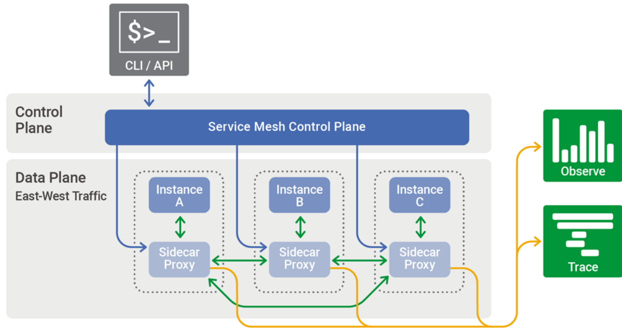

Figure 2: The control plane in a service mesh distributes configuration across the sidecar proxies in the data plane [27]

To further illustrate the connections between the control plane and the data plane, the control plane bootstraps a service instance along with its sidecar proxy and constantly reports a global status of service instances inside a microservice system. All the rules that restricting sidecar proxies' behaviors along with a credential management system are kept inside the control plane. When the user applies a policy, the control plane first validates the policy against the system's current configuration before deploying it, then announces the rules to sidecar proxies to regulate their behaviors. Although the data plane actually touches every packet, the control plane empowers the sidecar proxies to become a distributed system and makes service mesh valuable to complex microservice systems.

In summary, a service mesh is a dedicated infrastructure layer built right into a microservice application, and the goal of using a service mesh with your microservice system is better security, more reliability, lower cost, scale, and better resiliency within a set of closely intercommunicating systems [11].

3.2 Istio

Istio is an open-source application, initially developed by Google, IBM and Lyft, that delivers service mesh to microservice applications. Currently, Istio is only usable with Kubernetes[5]. Because of the nature of the service mesh, installing Istio does not require any change to the existing code base.

A service mesh is constructed with two components, such as Istio. Istio deploys Envoy proxies alongside each service instance acting as the data plane[6], and it implemented a set of components acting as the control plane.

The data plane, Envoy, is a Layer 7 proxy and a communication bus designed for large service-oriented architectures [13]. A Layer 7 proxy operates at the high-level application layer, which deals with the actual content of each packet. In other words, Layer 7 proxy can make a load-balancing decision based on the content of the message (the URL or cookie, for example) [20].

The control plane is the core of Istio. As discussed in 3.1, the responsibilities of the control plane: bootstrapping service instances with their sidecars, parsing and validating user-defined policies, announcing new rules and synchronizing rules to all sidecar proxies, credential management, and authentication, and interacting with the underlying infrastructure. Istio implements five components to achieve these responsibilities [5]:

Galley centralized configuration management and distribution. Galley insulates the rest of the Istio components from the underlying platform (i.e. Kubernetes). Galley ingests user-defined policies and validates them before distributing across the application.

Pilot configures all the Envoy proxy instances deployed in the Istio service mesh. It fetches configurations, for example, timeouts, retries, and circuit breakers, from Galley for traffic management, then pushes these configurations across sidecar proxies.

Mixer authorization policy enforcement and telemetry collection. Before handling each request, the Envoy proxy calls Mixer to perform precondition checks, then it decides whether the request should be allowed (e.g. authenticated request). During this process, telemetry is collected in Mixer for user observation.

Citadel provides strong service-to-service and end-user authentication using mTLS, with built-in identity and credential management.

Adapter connecting to a variety of infrastructure services, such as metrics, and logs.

As described in 3.1, the service meshes aim to help with the four aspects of a microservice application, we now discuss how Istio achieves that goal [11]:

Connect Istio helps handle the flow of traffic and API calls between services intelligently; its Envoy sidecar proxy provides load balancing, retries, service health check, circuit breakers, and canary releases feature for the microservice system. The configurations are stored inside proxies and dynamically configurable by the users.

Secure Istio secures communications between services through managed authentication, authorization, and encryption, and The Mixer and the Citadel are responsible for managing this. Each service has an identity asserted by an X.509 certificate that is automatically provisioned and used to implement two-way (mutual) Transport Level Security (TLS) for authorization and encryption of all API exchanges.

Control applies user-defined policies (for example, routing, rate limits) and enforces them across services. Inbound and outbound

[5] Google claims Istio is designed to run in any environment on any cloud, and Kubernetes is the first step [28]

[6] Although the majority of the articles regarding Istio considers Istio is a service mesh, there is a classification [17] considering Istio is only the control plane of a service mesh, just for the fact that Istio did not implement Envoy, its sidecar proxy. This report considers Istio to be a service mesh, as discussed earlier, the control plane makes service mesh valuable.

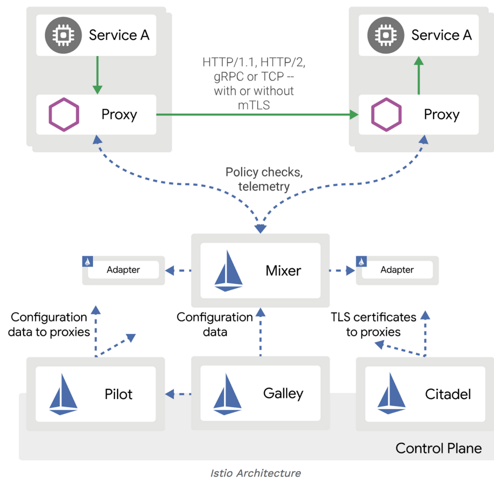

Figure 3: Istio Architecture from Google [11]

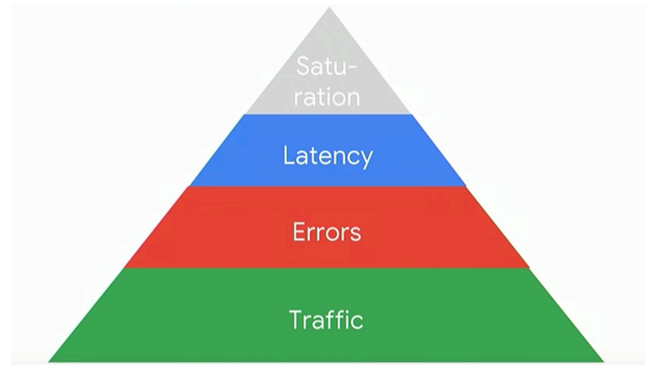

Figure 4: The four golden signals [8]

communications are controlled, including requests that go to external systems. Users can dynamically update the configurations, and the Galley and the Pilot are responsible for distributing the changes across the Envoy proxies.

Observe Istio ensures visibility with automatic tracing and operational logging of services. Mixer will collect telemetry of each request, and metrics visualization tools like Grafana and request tracing tools like Zipkin are supported in Istio.

4 ISTIO AND MICROSERVICE RELIABILITY

Istio is an excellent infrastructure layer that manages microservices without touching the existing code base. Recall that we defined four design patterns that can improve microservice reliability; we now discuss how Istio's features can help build a more reliable microservice-based application.

4.1 System Monitoring

Istio ensures visibility with automatic tracing and operational logging of services. It provides insights on performance (using *Grafana*) and distributed traffic flow (using *Jaeger*). It also offers visibility to monitor service status and detailed logs to quickly identify issues. Istio's sidecar proxies provide health check APIs to monitor service status and maintain service reliability.

Recall in 2.2.1, SRE suggested the four golden signals indicating system status. Istio automatically gathers latency, errors, and traffic for any microservice system. Saturation is different for every service, so Istio is unable to provide metrics for that signal [8].

4.2 Recovery-Oriented Computing

Unfortunately, Istio does not contribute to recovery-oriented computing. Service mesh only acts on the network layer of the application. Therefore, Istio cannot control service replication and system-wide recoverability, and these are parts of the infrastructure's responsibilities (i.e. Kubernetes).

4.3 Fault Tolerant

Istio provides the following features to build resilient and fault tolerant software [26] [6]:

Circuit Breakers when we apply a circuit breaker to an entity, and if failures reach a certain threshold, subsequent calls to that entity should automatically fail without applying additional pressure on the failed entity and paying for communication costs. It's nearly always better to fail quickly and apply back-pressure downstream as soon as possible. Envoy enforces circuit-breaking limits at the network level, as opposed to having to configure and code each application independently.

Control Connection Pool and Request Load Istio can also be used to specify maximum active connections to a microservice or maximum pending requests. We can set destination microservice maximum connections to X and maximum pending requests to Y. Thus, if we sent more than X+Y requests at once to the microservice, it will have Y pending requests and deny any additional requests until the pending requests are processed.

Health Checks Istio can perform health checks against a load-balancing pool. A microservice in a load-balancing pool can have multiple deployed instances, and Istio distributes traffic across those instances. If some of those instances are broken, Istio can perform health checks and eject any broken instance in your load-balancing pool to avoid any further failure.

Retries/Timeouts Istio allows you to create route rules for your destination microservices where you can specify the timeout and retry policies. For example, you can create a routing rule if your microservice does not respond within n seconds. Once applied, this rule will time out all the responses that take more than n seconds in the destination service. You also can apply the retries rule by telling Istio how many retries you want if a particular microservices is not reachable and what the timeout should be for your retry.

So if at first attempt, your destination microservice is not reachable in n seconds, you can tell Istio to do m number of retries and also increase the timeout for retries beyond n.

4.4 Testing

Istio enables the system to perform chaos testing and canarying releases.

4.4.1 Canarying Releases. In 2.2.4, SRE suggests using canarying releases can improve service reliability. Basically, canarying releases mean to introduce a new version of service by first testing it using a small percentage of user traffic, and then if all goes well, increase, possibly gradually in increments, the percentage while simultaneously phasing out the old version. The Istio routing rules can be used to route traffic based on specific criteria. Istio provides the control necessary to manage traffic distribution with complete independence from deployment scaling [10].

4.4.2 Chaos Testing. Istio provides a valuable feature called fault injection. With this feature, failures can be injected at the application layer like HTTP Errors or Delays to test the resiliency of the application. You can configure faults to be injected into requests that match specific conditions. You can inject either delays or faults into the requests. This will mimic service failures and latency between service calls [4].

5 CASE STUDY

We performed a case study on an Istio demo project [12] to demonstrate reliability improvements after installing Istio. Recall that request failures in a software system are random and expected, and software reliability is an evaluation of overall available time (i.e. MTTF Mean Time to Failure and MTTR Mean Time to Repair) of the application during a certain period. Hence, the selected demo project focuses on simulating real deployment environment and transient network issues. Because MTTR is an operation team-specific measurement that this report is unable to evaluate, the case study only uses random failures to simulate MTTF and applies retry policies to demonstrate network resiliency improvement of the application.

The demo project does not provide actual functionalities, and it directly replies a request with one of the three status codes, 200, 400, and 500. The server implementation is set to rejects a request at a 25 percent chance, and the rejected request status code is set randomly to 400 or 500. Ideally, 75 percent of the total requests will be responded with 200, and in the failed requests, half of them will be 500 and the other half will be 400. After constantly sending requests to the server for 1 hour, as we can observe from the Grafana charts, the success rate matches our expectations, and this implementation is certainly not a reliable software.

In the previous stage, Istio has already been installed without any traffic management configuration. The reason is that it is impossible to learn system status if the application does not explicitly collect service metrics. Without monitoring the application, the operation team cannot answer the four golden signals of the system, so it is hard to react to failure if the failure is not alerting anyone. With Istio, however, network traffic flow is automatically collected regardless of service implementation, and the Grafana dashboard can automatically query system status to answer the question of service latency, request errors, and network traffics.

Because we consider the system failure as random and potential transient network issues, we apply the retry policies to evaluate overall system availability. For each request, sidecar proxies will retry three times before rejecting it. By observing the Grafana chart, the global success rate is greatly improved. Also observe that the latency of each request becomes higher than before (i.e. P50 Latency, P90 Latency, and P99 Latency in 5), and that is because, on average, retrying requires more time to respond to a request. This is the trade-offs between service availability and its performance.

6 CRITICISMS OF ISTIO

Despite its attractive features and reliability improvements, Istio relies on certain infrastructure setup[7]. Typical concern regarding Istio and service meshes is their performance impacts on the application.

Adopting sidecar proxies cost extra computation resources[8], which is crucial especially when company budget restricts available computation power (e.g. deploying the application on Google Cloud Platform). Latency is also an important aspect to consider. Istio introduces latency for each request because the network flow is handled by sidecar proxies injected into each service instance.

Here is the performance summary of Istio 1.4.2 [15]:

- The Envoy proxy uses 0.5 vCPU and 50 MB memory per 1000 requests per second going through the proxy.
- The istio-telemetry service uses 0.6 vCPU per 1000 mesh-wide requests per second.
- Pilot uses 1 vCPU and 1.5 GB of memory.
- The Envoy proxy adds 6.3 ms to the 90th percentile latency.

7 CONCLUSIONS

This report discussed definitions of software reliability and principles that help to build a reliable microservice, then it introduced service mesh, an architectural paradigm, that helps to manage the network layer of the microservice application along with Istio, an application that delivers service mesh. Finally, the report analyzed how Istio improves software reliability by providing observability and improving network resiliency and its limitations and performance impacts.

Despite the apparent popularity of Istio, there are other service meshes exist in practice, for example, Linkerd. Future studies could focus on comparing the implementation trade-offs between different service mesh and their performance. As for software reliability, the service mesh is only a small portion of microservice architecture, so future studies could focus on other paradigms that improve microservice reliability to form a coherent reliable microservice architecture model.

REFERENCES

[1] Ingo Averdunk and Hans Kristian Moen. 2019. Implement health check APIs for microservices. https://www.ibm.com/garage/method/practices/manage/health-check-apis. (2019). [Online; accessed December 2019].

[2] AWS. 2019. Istio: a modern approach to developing and managing microservices. https://d1.awsstatic.com/whitepapers/architecture/AWS-Reliability-Pillar.pdf. (2019). [Online; accessed December 2019].

[7] As of writing, Kubernetes is the only platform Istio supports

[8] Istio costs 50% more CPU resources than another service mesh, Linkerd [16]. However, Linkerd provides less network traffic control policies than Istio [24]

Figure 5: Grafana Charts for Successful Rate Improvements

[3] Microsoft Azure. 2017. Resiliency patterns. https://docs.microsoft.com/en-us/azure/architecture/patterns/category/resiliency. (2017). [Online; accessed December 2019].

[4] Samir Behara. 2019. Chaos Testing Your Microservices With Istio. https://dzone.com/articles/chaos-testing-your-microservices-with-istio. (2019). [Online; accessed December 2019].

[5] Samir Behara. 2019. Istio Service Mesh Control Plane. https://dzone.com/articles/istio-service-mesh-control-plane. (2019). [Online; accessed December 2019].

[6] Samir Behara. 2019. Istio Service Mesh Control Plane. https://dzone.com/articles/resilient-microservices-with-istio-circuit-breaker. (2019). [Online; accessed December 2019].

[7] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering*. O'Reilly Media.

[8] Betsy Beyer, David K. Rensin, Kent Kawahara, Stephen Thorne, and Niall Richard Murphy. 2016. *The Site Reliability Workbook: Practical Ways to Implement SRE*. O'Reilly Media.

[9] Ben E. C. Boyter. 2016. What is Chaos Testing / Engineering. https://boyter.org/2016/07/chaos-testing-engineering/. (2016). [Online; accessed December 2019].

[10] FRANK BUDINSKY. 2019. Canary Deployments using Istio. https://istio.io/blog/2017/0.1-canary/. (2019). [Online; accessed December 2019].

[11] Dino Chiesa and Greg Kuelgen. 2019. Got microservices? Service mesh management might not be enough. https://cloud.google.com/blog/products/api-management/got-microservices-service-mesh-management-might-not-be-enough. (2019). [Online; accessed December 2019].

[12] Sandeep Dinesh. 2019. Istio101. https://github.com/thesandlord/Istio101. (2019). [Online; accessed December 2019].

[13] Envoy. 2019. What is Envoy? https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy. (2019). [Online; accessed December 2019].

[14] A. L. Goel. 1985. Software Reliability Models: Assumptions and Limitations and and Applicability. *IEEE Transactions on Software Engineering* SE-11, 12 (Dec 1985), 1411–1423. https://doi.org/10.1109/TSE.1985.232177

[15] Istio. 2019. Performance and Scalability. https://istio.io/docs/ops/deployment/performance-and-scalability/. (2019). [Online; accessed December 2019].

[16] Michael Kipper. 2019. Benchmarking Istio and Linkerd CPU. https://medium.com/@michael_87395/benchmarking-istio-linkerd-cpu-c36287e32781. (2019). [Online; accessed December 2019].

[17] Matt Klein. 2017. Service mesh data plane vs. control plane. https://blog.envoyproxy.io/service-mesh-data-plane-vs-control-plane-2774e720f7fc. (2017). [Online; accessed December 2019].

[18] Michael R. Lyu. 1996. *Handbook of Software Reliability Engineering*. McGraw-Hill, Inc.

[19] Microservice.io. 2019. A pattern language for microservices. https://microservices.io/patterns/index.html. (2019). [Online; accessed December 2019].

[20] Nginx. 2019. What Is Layer 7 Load Balancing? https://www.nginx.com/resources/glossary/layer-7-load-balancing/. (2019). [Online; accessed December 2019].

[21] R. Pietrantuono, S. Russo, and A. Guerriero. 2018. Run-Time Reliability Estimation of Microservice Architectures. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. 25–35. https://doi.org/10.1109/ISSRE.2018.00014

[22] Redhat. 2019. What's a service mesh? https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh. (2019). [Online; accessed December 2019].

[23] Dave Rensin. 2016. Introducing Google Customer Reliability Engineering. https://cloud.google.com/blog/products/gcp/introducing-a-new-era-of-customer-support-google-customer-reliability-engineering. (2016). [Online; accessed December 2019].

[24] Diógenes Rettori. 2019. Linkerd or Istio? https://medium.com/solo-io/linkerd-or-istio-6fcd2aad6e42. (2019). [Online; accessed December 2019].

[25] ROC-group. 2004. Recovery-Oriented Computing Overview. http://roc.cs.berkeley.edu/roc_overview.html. (2004). [Online; accessed December 2019].

[26] Animesh Singh. 2017. Make your microservices resilient and fault-tolerant using Istio. https://developer.ibm.com/code/2017/08/17/make-microservices-resilient-fault-tolerant-using-istio/. (2017). [Online; accessed December 2019].

[27] Floyd Smith and Owen Garrett. 2018. What Is a Service Mesh? https://www.nginx.com/blog/what-is-a-service-mesh/. (2018). [Online; accessed December 2019].

[28] Varun Talwar. 2017. Istio: a modern approach to developing and managing microservices. https://cloud.google.com/blog/products/gcp/istio-modern-approach-to-developing-and. (2017). [Online; accessed December 2019].

[29] Tokuno, Koichi, Yamada, and Shigeru. 2003. *Software Availability Theory and Its Applications*. Springer London, London, 235–244. https://doi.org/10.1007/1-85233-841-5_13

[30] Wikipedia. 2016. Site Reliability Engineering. https://en.wikipedia.org/wiki/Site_Reliability_Engineering. (2016). [Online; accessed December 2019].