DevOpsCon presents:

Best Practices for a Service Mesh – Based on Istio

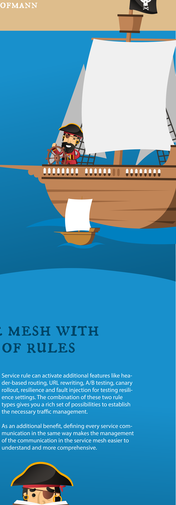
by Michael Hofmann

Use automatic sidecar injection

One of the basic elements in your service mesh tool is the usage of a sidecar. All communication to and from your application will be routed through this proxy. Therefore, it is essential the sidecar gets deployed together with your application container in the same Kubernetes pod.

This can be achieved with automatic sidecar injection. If this feature is activated, Istio injects a sidecar beside every application container without manual intervention. This is the preferred way. The usage of automatic side car injection guarantees that all your pods are part of the service mesh and metrics are available.

In some situations, e.g. debugging problems in your service mesh, it might be necessary to change the deployment of the sidecar to activate some more metrics, which are deactivated by default in the proxy. You can switch from automatic sidecar injection to manual injection and change the deployment settings. But this should only be an exception to this best practice.



Manage your service mesh with a comprehensive set of rules

Running your service mesh without Istio's traffic rules is possible, but then you establish a mesh communication which is unmanaged and therefore difficult to handle. With a few Istio rules activated for every application in your mesh, life is much easier.

Communication entering your cluster should not only be distributed by a load balancer. In addition to this infrastructure component, the communication should also be controlled by an ingress gateway. An Istio gateway can start to monitor your communication flow and can activate route rules that get applied to the traffic entering your cluster. The same arguments are true for using an Istio egress gateway applied to all communication leaving the service mesh.

Every communication to a service inside your mesh should be defined with an Istio VirtualService and a DestinationRule. A DestinationRule can define subsets of routing destinations based on different versions of a service coexisting in the mesh. These subsets can be referenced in the VirtualService rule. A VirtualService rule can activate additional features like header-based routing, URL rewriting, A/B testing, canary rollout, resilience and fault injection for testing resilience settings. The combination of these two rule types gives you a rich set of possibilities to establish the necessary traffic management.

As an additional benefit, defining every service communication in the same way makes the management of the communication in the service mesh easier to understand and more comprehensive.

Double check rules

Istio rules with a faulty configuration will not be applied by the sidecar. Communication with another service can be possible, but the incorrect rules will be ignored. Therefore, it is essential to double check the rules defined for your mesh to verify that they are correctly applied. You also should make sure these rules do what you want them to do. This can be done by testing communication errors or different communication behavior with faulty rules or missing rules.

Istio's new component called Galley checks if the rule is syntactically and semantically correct. By using a newer version of Istio, less faulty configurations can be applied. Another tool to control the correctness of the rule definition is Kiali. It provides additional semantic validation on top of the existing validations executed by Galley. Istio's resources in your mesh will be shown on Kiali's dashboard, and errors will be flagged.

If you use Istio rules where you specify different routing rules, e.g. in a VirtualService where you set a match on header values or set percent values on a routing definition, it is also recommended to define a default routing that gets applied if the conditions in a preceding part of a rule do not match.

There are some requirements a Kubernetes pod and service must follow to be part of an Istio service mesh. A detailed list can be found Istio's documentation (https://istio.io/docs/setup/kubernetes/additional-setup/requirements/).

All these recommendations and tools will help you to get correct and applicable rules.

Delegate resilience to service mesh

Communication in a distributed system should always be resilient. Problems in calling another service are usually the result of infrastructure problems. Therefore, it is a good advice to compensate for these problems in the infrastructure itself. A service mesh tool can do this.

In Istio, it is possible to define rules for the following resilience patterns: timeout, retry, circuit breaker, and bulkhead. If you want to make your service communication more resilient, then use the corresponding settings in the Istio rules. This keeps your application free from implementing these resilience patterns. Handling resilience in both the application and the service mesh rules would be counterproductive.

Fallback, as another resilience pattern, always has something to do with your business logic. That is why Istio has no rule setting for executing a fallback. This must be done in the application itself and cannot be delegated to the service mesh tool.

Define clear boundaries for service mesh security and application security

Larger applications usually need security. The same is also true of a service mesh. Security can be established by the service mesh tool as well as by the services. Therefore, you should define clear boundaries as to which part of the security must be done by the infrastructure (service mesh tool and container platform) and which part will be implemented in the services. Otherwise you will check for security twice or, and this is much worse, not at all.

When using JWT propagation, for example, verification of the JWT should be performed by the service mesh tool so no requests with a timed-out or invalid JWT can enter the service mesh.

If you want to use RBAC, you need to be careful where you establish it in your mesh. RBAC can be defined in Istio rules, but also in your application. Often, applications need to implement programmatic security to have granular control of the data processed by a user. This is normally based on a role definition that must be checked in the source code. A redundant definition and verification of this role in Istio, and in the application, can lead to some problems or complications, e.g. when you change a role definition or establish new roles. Such a redundancy must be carefully defined and controlled. A clear definition with examples of where to place the RBAC (service mesh, application, or both in your project can help to manage this situation.

Start service mesh tooling early in your project and integrate it into your CI/CD pipeline

A service mesh tool is complex, so it is important to establish knowledge in the project team about how to use it correctly. To get ahead of this steep learning curve it is better to start early in a project and use only a few features or rules of the service mesh tool. Subsequently, you can integrate additional features and rules into your project.

Integrate the service mesh tool into your CI/CD pipeline to get a high degree of automation into your project. This is essential to get a comprehensible configuration of your services and prevent so-called snowflakes. To avoid snowflakes, every Istio rule must be under version control together with your source code. This also helps you in mastering the service mesh tool.

After achieving this, you have a good starting point to establish new deployment and testing strategies. You can switch to zero downtime deployments or integrate A/B or canary deployments into your project. You can implement new testing strategies by using Istio's rules for fault injection or traffic shadowing.

Generate and collect logs, metrics and traces

Service mesh communication can be very complex and so when there is an error or problem it can be very difficult to analyze the root cause. Generating and collecting logs, metrics and traces are the basis for successful problem handling. That's why Istio's installation scripts also install a lot of tools like Prometheus and Grafana for metrics data, and Jaegar or Zipkin for tracing. Istio can easily be combined with Fluentd, Elasticsearch and Kibana to manage the log output.

By using a service mesh tool, you double the components taking part in service-to-service communication. More components can produce more problems, so you have to monitor and log these components together with your applications. In the default configuration, the components installed with Istio send this information to the appropriate systems so all you have to do is integrate your application into these information systems.

If you use commercial tools or different open source tools for your logs, metrics and trace data, then you can integrate these tools with your service mesh tool.

Visualize your service mesh

As your service mesh grows in size, it also grows in complexity. A service mesh consisting of just a few services can be difficult to understand, so visualizing your service mesh is necessary to maintain an overview of your services. Manual documenting of the service mesh is a lot of work and cannot be kept up to date with the real world; therefore, a tool-based approach is necessary.

Istio's default tool for helping you is Kiali. It shows the service mesh topology and other features used in Istio's rules like circuit breakers, VirtualService definitions and service versioning used in DestinationRules. It also includes Jaeger to provide distributed tracing information. Other dynamic parts of the service mesh can be visualized by Kiali, which shows request rates, latency, metrics and so on.