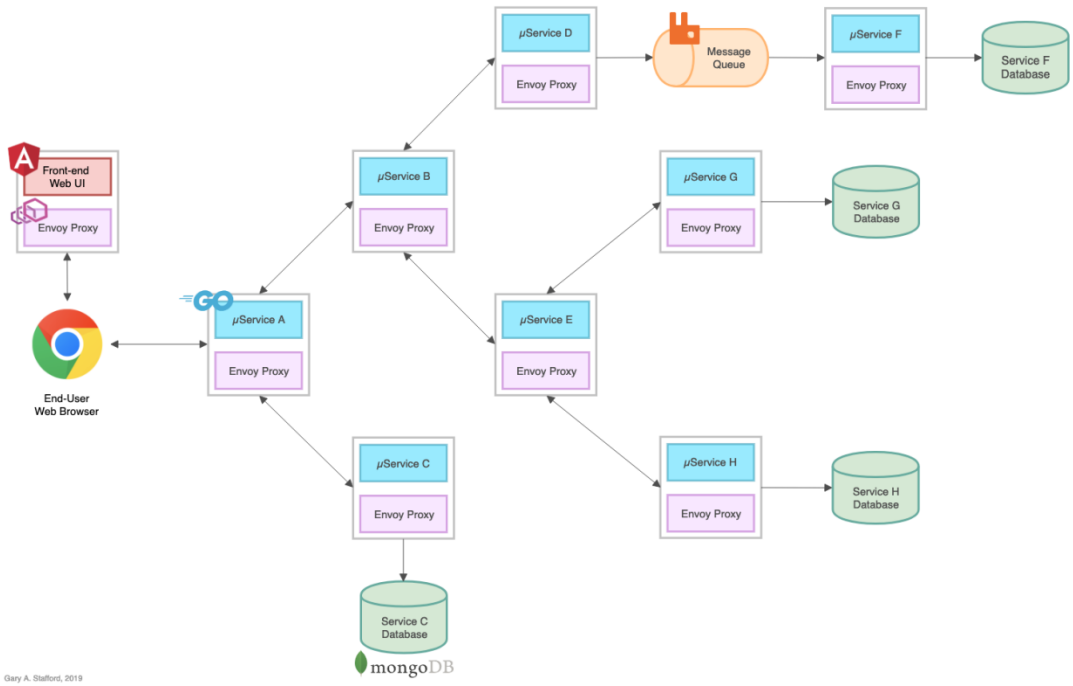
Kubernetes-based Microservice Observability with Istio Service Mesh: Part 2

In this two-part post, we are exploring the set of observability tools that are part of the latest version of Istio Service Mesh. These tools include Prometheus (https://prometheus.io/) and Grafana (https://grafana.com/) for metric collection, monitoring, and alerting, Jaeger (https://www.jaegertracing.io/) for distributed tracing, and Kiali (https://www.kiali.io/) for Istio service-mesh-based microservice visualization. Combined with cloud platform-native monitoring and logging services, such as Stackdriver (https://cloud.google.com/monitoring/) for Google Kubernetes Engine (https://cloud.google.com/kubernetes-engine/) (GKE) on Google Cloud Platform (GCP), we have a complete observability solution for modern, distributed applications.

Reference Platform

To demonstrate Istio's observability tools, in part one (https://programmaticponderings.com/2019/03/10/kubernetes-based-microservice-observability-with-istio-service-mesh-part-1/) of the post, we deployed a reference microservices platform, written in Go, to GKE on GCP. The platform is comprised of (14) components, including (8) Go-based (https://golang.org/) microservices, labeled generically as Service A through Service H, (1) Angular 7, TypeScript-based (https://en.wikipedia.org/wiki/TypeScript) front-end, (4) MongoDB databases, and (1) RabbitMQ queue for event queue-based communications.



(https://programmaticponderings.files.wordpress.com/2019/03/golang-service-diagram-with-proxy-v2.png)

The reference platform is designed to generate HTTP-based service-to-service, TCP-based service-to-database (MongoDB), and TCP-based service-to-queue-to-service (RabbitMQ) IPC (inter-process communication). Service A calls Service B and Service C, Service B calls Service D and Service E, Service D produces a message on a RabbitMQ queue that Service F consumes and writes to MongoDB, and so on. The goal is to observe these distributed communications using Istio's observability tools when the system is deployed to Kubernetes.

Pillar 1: Logging

If you recall, logs, metrics, and traces are often known as the three pillars of observability. Since we are using GKE on GCP, we will look at Google's Stackdriver Logging (https://cloud.google.com/monitoring/). According to Google, Stackdriver Logging allows you to store, search, analyze, monitor, and alert on log data and events from GCP and even AWS. Although Stackdriver logging is not an Istio observability feature, logging is an essential pillar of overall observability strategy.

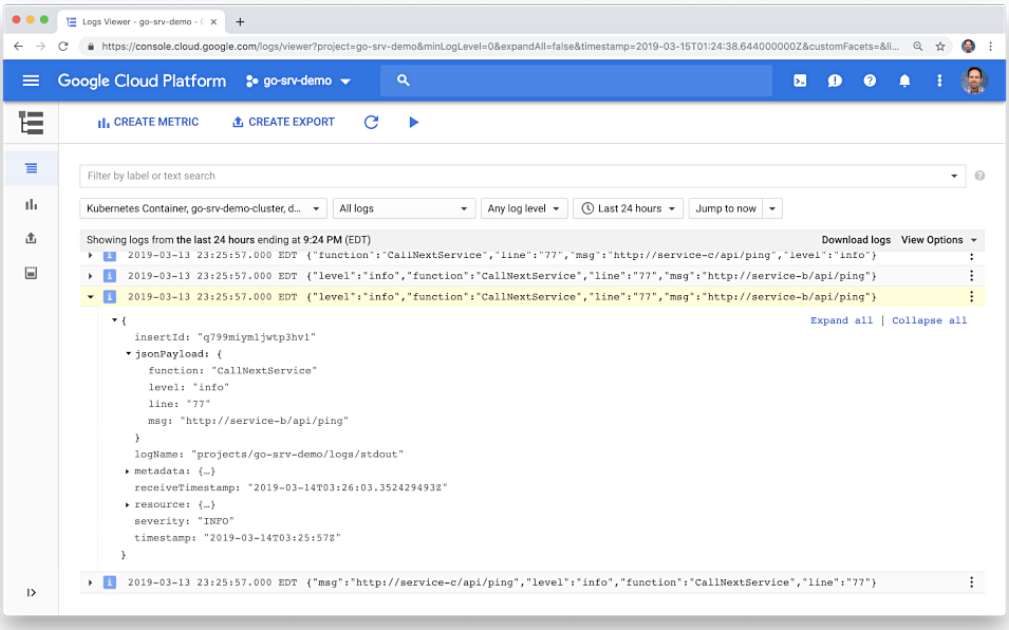
Go-based Microservice Logging

An effective logging strategy starts with what you log, when you log, and how you log. As part of our logging strategy, the eight Go-based microservices are using Logrus (https://github.com/sirupsen/logrus), a popular structured logger for Go. The microservices also implement Banzai Cloud's logrus-runtime-formatter (https://github.com/sirupsen/logrus). There is an excellent article on the formatter, Golang runtime Logrus Formatter (https://banzaicloud.com/blog/runtime-logging/). These two logging packages give us greater control over what we log, when we log, and how we log information about our microservices. The recommended configuration of the packages is minimal.

```
func init() {
   formatter := runtime.Formatter{ChildFormatter: &log.JSONFormatter{}}
   formatter.Line = true
   log.SetFormatter(&formatter)
   log.SetOutput(os.Stdout)
   level, err := log.ParseLevel(getEnv("LOG_LEVEL", "info"))
   if err != nil {
      log.Error(err)
   }
   log.SetLevel(level)
}
```

Logrus provides several advantages of over Go's simple logging package, log (https://golang.org/pkg/log/). Log entries are not only for Fatal errors, nor should all verbose log entries be output in a Production environment. The post's microservices are taking advantage of Logrus' ability to log at seven levels: Trace, Debug, Info, Warning, Error, Fatal and Panic. We have also variabilized the log level, allowing it to be easily changed in the Kubernetes Deployment resource at deploy-time.

The microservices also take advantage of Banzai Cloud's logrus-runtime-formatter (https://github.com/sirupsen/logrus). The Banzai formatter automatically tags log messages with runtime/stack information, including function name and line number; extremely helpful when troubleshooting. We are also using Logrus' JSON formatter. Note how each log entry below has the JSON payload contained within the message.

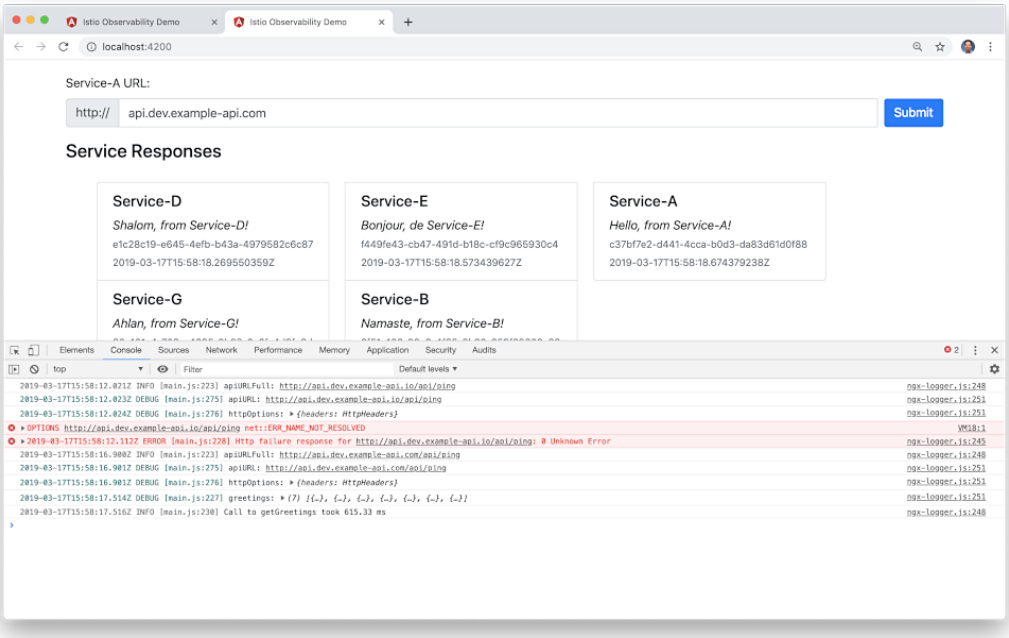


(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-14_at_9_28_09_pm.png)

Client-side Angular UI Logging

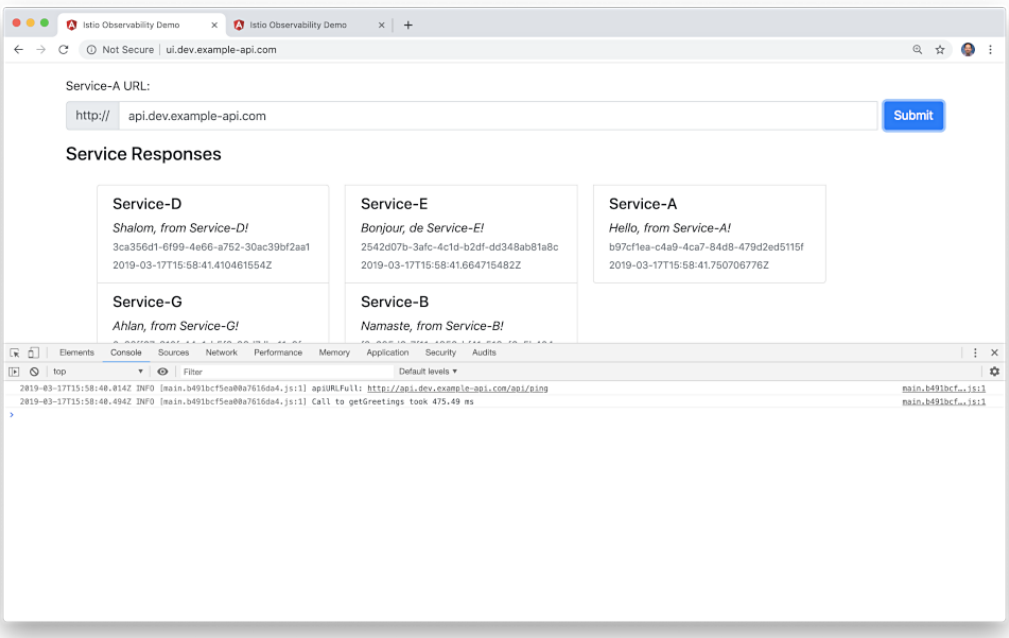
Likewise, we have enhanced the logging of the Angular UI using NGX Logger (https://www.npmjs.com/package/ngx-logger). NGX Logger is a popular, simple logging module, currently for Angular 6 and 7. It allows "pretty print" to the console, as well as allowing log messages to be POSTed to a URL for server-side logging. For this demo, we will only print to the console. Similar to Logrus, NGX Logger supports multiple log levels: Trace, Debug, Info, Warning, Error, Fatal, and Off. Instead of just outputting messages, NGX Logger allows us to output properly formatted log entries to the web browser's console.

The level of logs output is dependent on the environment, Production or not Production. Below we see a combination of log entries in the local development environment, including Debug, Info, and Error.



(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-17_at_11_58_31_am-1.png)

Again below, we see the same page in the GKE-based Production environment. Note the absence of Debug-level log entries output to the console, without changing the configuration. We would not want to expose potentially sensitive information in verbose log output to our end-users in Production.



(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-17_at_11_58_45_am.png)

Controlling logging levels is accomplished by adding the following ternary operator to the app.module.ts file.

```
LoggerModule.forRoot({
  level: !environment.production ?
    NgxLoggerLevel.DEBUG : NgxLoggerLevel.INFO,
    serverLogLevel: NgxLoggerLevel.INFO
})
```

Pillar 2: Metrics

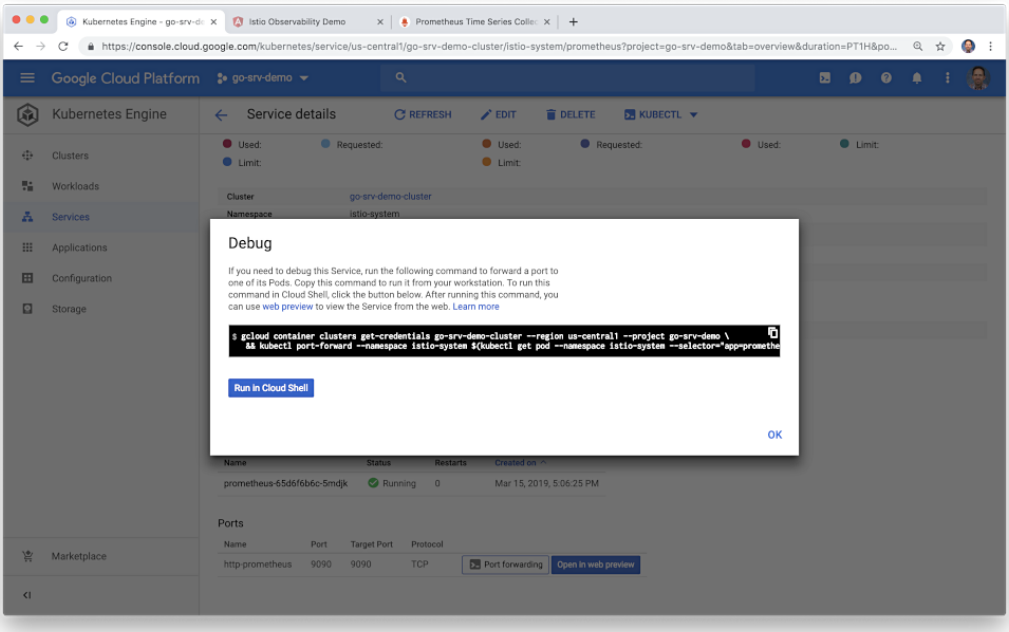
For metrics, we will examine at Prometheus (https://prometheus.io/) and Grafana (https://grafana.com/). Both these leading tools were installed as part of the Istio deployment.

Prometheus

Prometheus is a completely open source and community-driven systems monitoring and alerting toolkit originally built at SoundCloud, circa 2012. Interestingly, Prometheus joined the Cloud Native Computing Foundation (https://cncf.io/) (CNCF) in 2016 as the second hosted-project, after Kubernetes (http://kubernetes.io/).

According to Istio (https://istio.io/docs/tasks/telemetry/querying-metrics/), Istio's Mixer comes with a built-in Prometheus adapter that exposes an endpoint serving generated metric values. The Prometheus add-on is a Prometheus server that comes pre-configured to scrape Mixer endpoints to collect the exposed metrics. It provides a mechanism for persistent storage and querying of Istio metrics.

With the GKE cluster running, Istio installed, and the platform deployed, the easiest way to access Grafana, is using `kubectl port-forward` to connect to the Prometheus server. According to Google, Kubernetes port forwarding (https://kubernetes.io/docs/tasks/access-application-cluster/port-forward-access-application-cluster/) allows using a resource name, such as a service name, to select a matching pod to port forward to since Kubernetes v1.10. We forward a local port to a port on the Prometheus pod.



(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-15_at_7_32_23_pm.png)

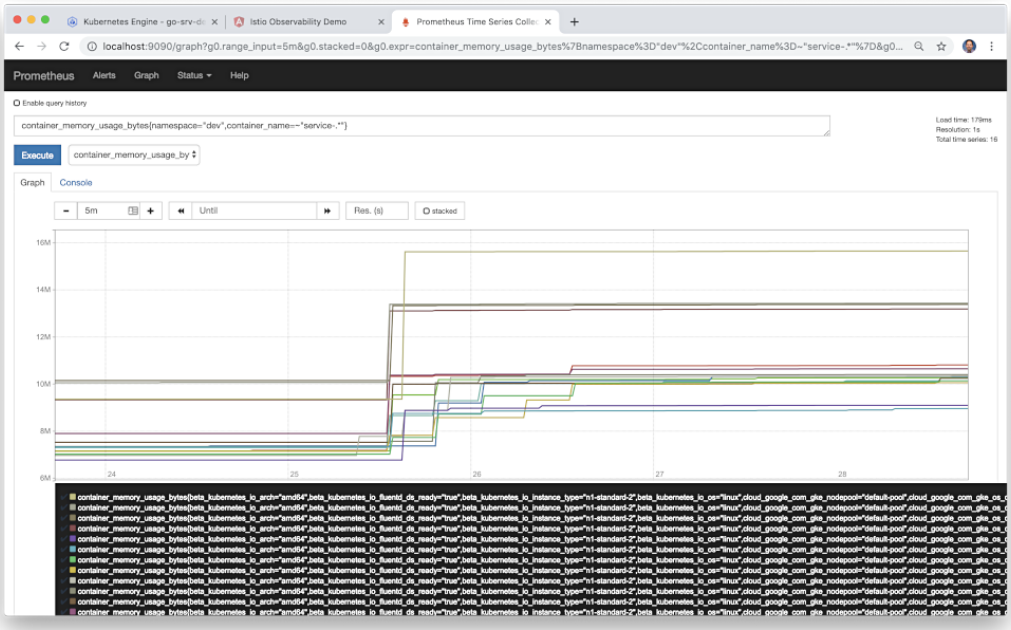
You may connect using Google Cloud Shell or copy and paste the command to your local shell to connect from a local port. Below are the port forwarding commands used in this post.

```
# Grafana
kubectl port-forward -n istio-system \
  $(kubectl get pod -n istio-system -l app=grafana \
  -o jsonpath='{.items[0].metadata.name}') 3000:3000 &

# Prometheus
kubectl -n istio-system port-forward \
  $(kubectl -n istio-system get pod -l app=prometheus \
  -o jsonpath='{.items[0].metadata.name}') 9090:9090 &

# Jaeger
kubectl port-forward -n istio-system \
$(kubectl get pod -n istio-system -l app=jaeger \
-o jsonpath='{.items[0].metadata.name}') 16686:16686 &

# Kiali
kubectl -n istio-system port-forward \
  $(kubectl -n istio-system get pod -l app=kiali \
  -o jsonpath='{.items[0].metadata.name}') 20001:20001 &
```

According to Prometheus (https://prometheus.io/docs/prometheus/latest/querying/basics/), user select and aggregate time series data in real time using a functional query language called PromQL (https://prometheus.io/docs/prometheus/latest/querying/basics/) (Prometheus Query Language). The result of an expression can either be shown as a graph, viewed as tabular data in Prometheus's expression browser, or consumed by external systems through Prometheus' HTTP API (https://prometheus.io/docs/prometheus/latest/querying/api/). The expression browser includes a drop-down menu with all available metrics as a starting point for building queries. Shown below are a few PromQL examples used in this post.

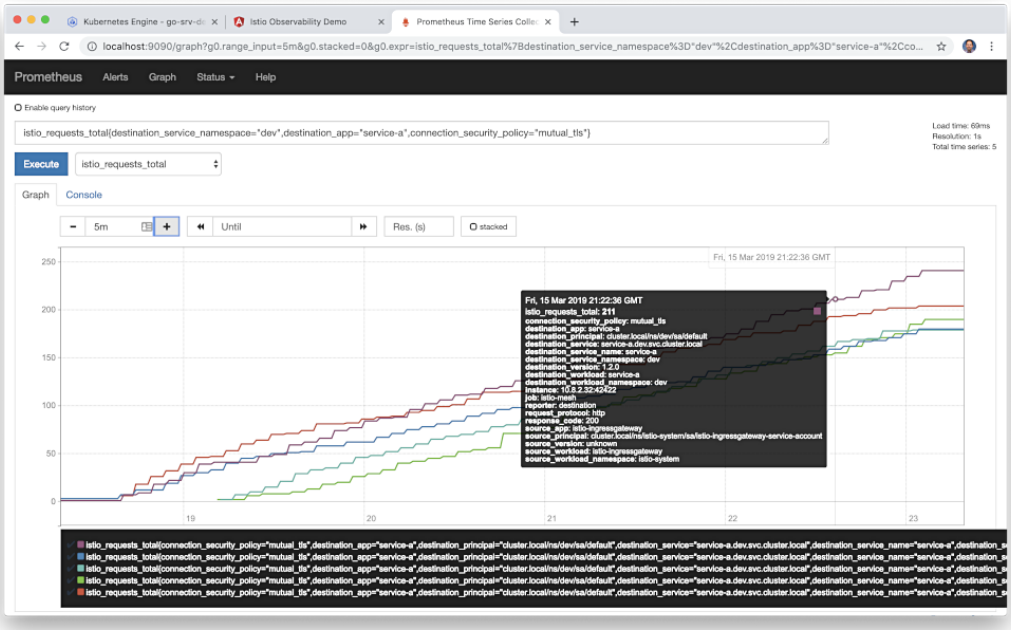
```
up{namespace="dev",pod_name=~"service-.*"}

container_memory_max_usage_bytes{namespace="dev",container_name=~"service-
container_memory_max_usage_bytes{namespace="dev",container_name="service-f'
container_network_transmit_packets_total{namespace="dev",pod_name=~"service

istio_requests_total{destination_service_namespace="dev",connection_securit
istio_response_bytes_count{destination_service_namespace="dev",connection_s
```

Below, in the Prometheus console, we see an example graph of the eight Go-based microservices, deployed to GKE. The graph displays the container memory usage over a five minute period. For half the time period, the services were at rest. For the second half of the period, the services were under a simulated load, using `hey`. Viewing the memory profile of the services under load can help us determine the container memory minimums and limits, which impact Kubernetes' scheduling of workloads on the GKE cluster. Metrics such as this might also uncover memory leaks or routing issues, such as the service below, which appears to be consuming 25-50% more memory than its peers.



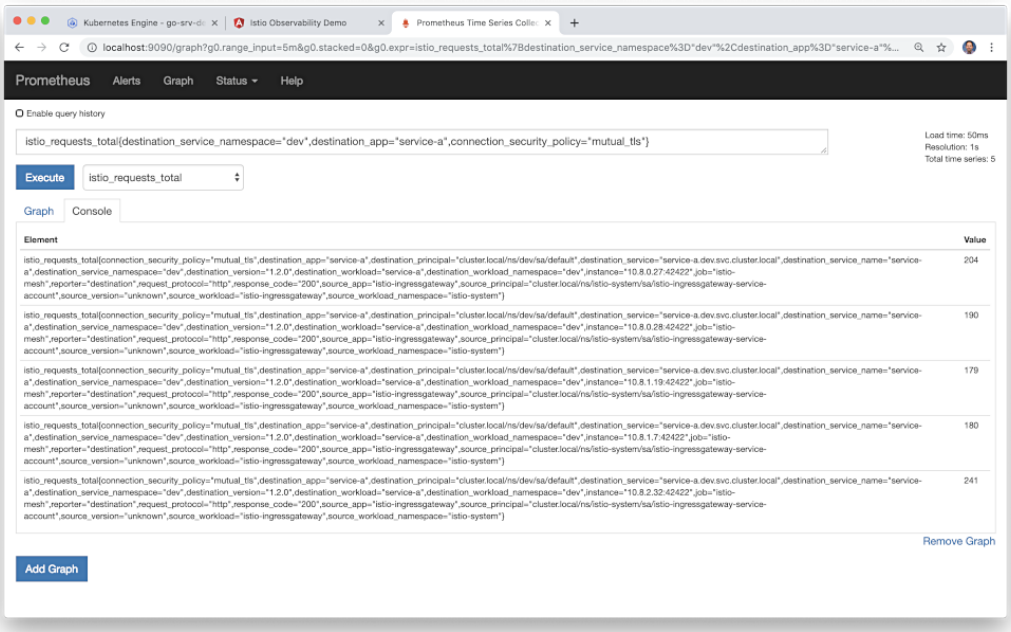
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-15_at_7_15_24_pm.png)

Another example, below, we see a graph representing the total Istio requests to Service A in the `dev` Namespace, while the system was under load.



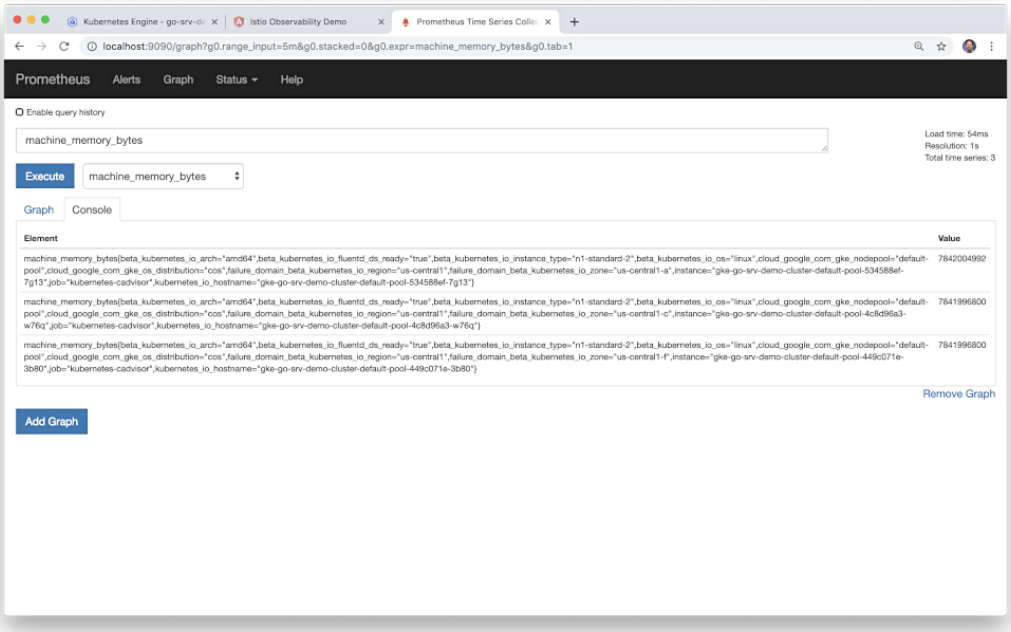
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-15_at_5_23_26_pm.png)

Compare the graph view above with the same metrics displayed the console view. The multiple entries reflect the multiple instances of Service A in the `dev` Namespace, over the five-minute period being examined. The values in the individual metric elements indicate the latest metric that was collected.



(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-15_at_5_24_12_pm.png)

Prometheus also collects basic metrics about Istio components, Kubernetes components, and GKE cluster. Below we can view the total memory of each n1-standard-2 VM nodes in the GKE cluster.



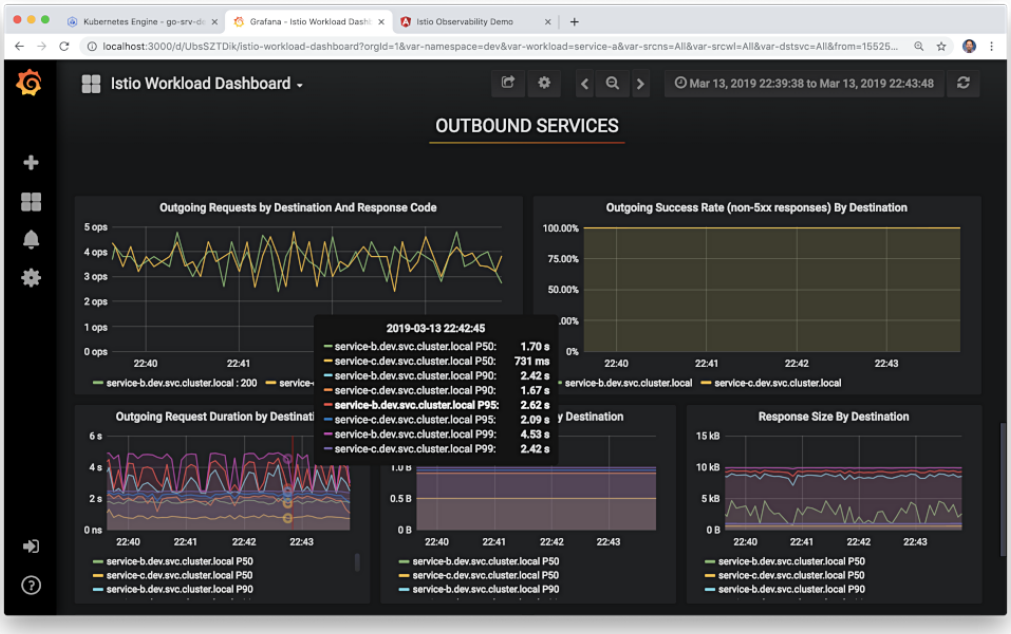
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-15_at_8_15_03_pm.png)

Grafana

Grafana describes itself as the leading open source software for time series analytics. According to Grafana Labs, (https://grafana.com/grafana) Grafana allows you to query, visualize, alert on, and understand your metrics no matter where they are stored. You can easily create, explore, and share visually-rich, data-driven dashboards. Grafana also users to visually define alert rules for your most important metrics. Grafana will continuously evaluate rules and can send notifications.

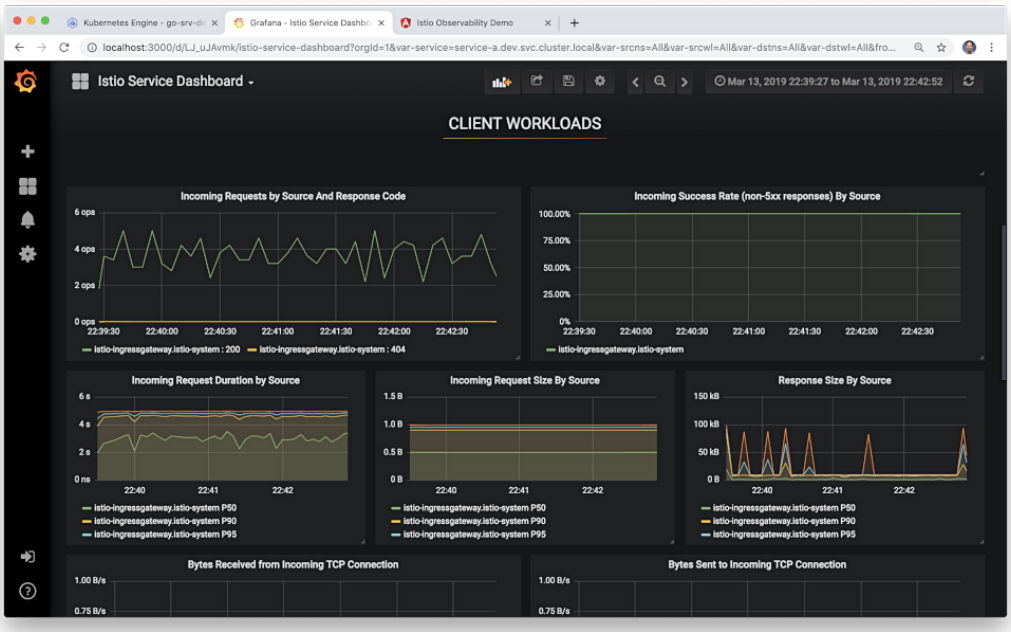
According to Istio (https://istio.io/docs/tasks/telemetry/using-istio-dashboard/#about-the-grafana-add-on), the Grafana add-on is a pre-configured instance of Grafana. The Grafana Docker base image has been modified to start with both a Prometheus data source and the Istio Dashboard installed. The base install files for Istio, and Mixer in particular, ship with a default configuration of global (used for every service) metrics. The pre-configured Istio Dashboards are built to be used in conjunction with the default Istio metrics configuration and a Prometheus back-end.

Below, we see the pre-configured Istio Workload Dashboard. This particular section of the larger dashboard has been filtered to show outbound service metrics in the `dev` Namespace of our GKE cluster.



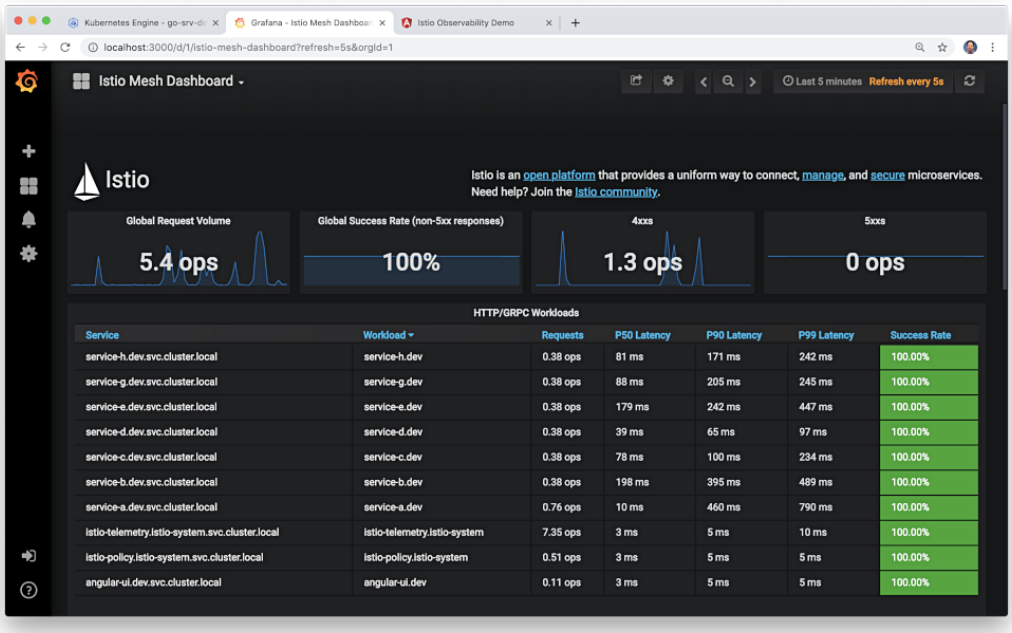
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-13_at_10_44_54_pm.png)

Similarly, below, we see the pre-configured Istio Service Dashboard. This particular section of the larger dashboard is filtered to show client workloads metrics for the Istio Ingress Gateway in our GKE cluster.



(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-13_at_10_43_11_pm.png)

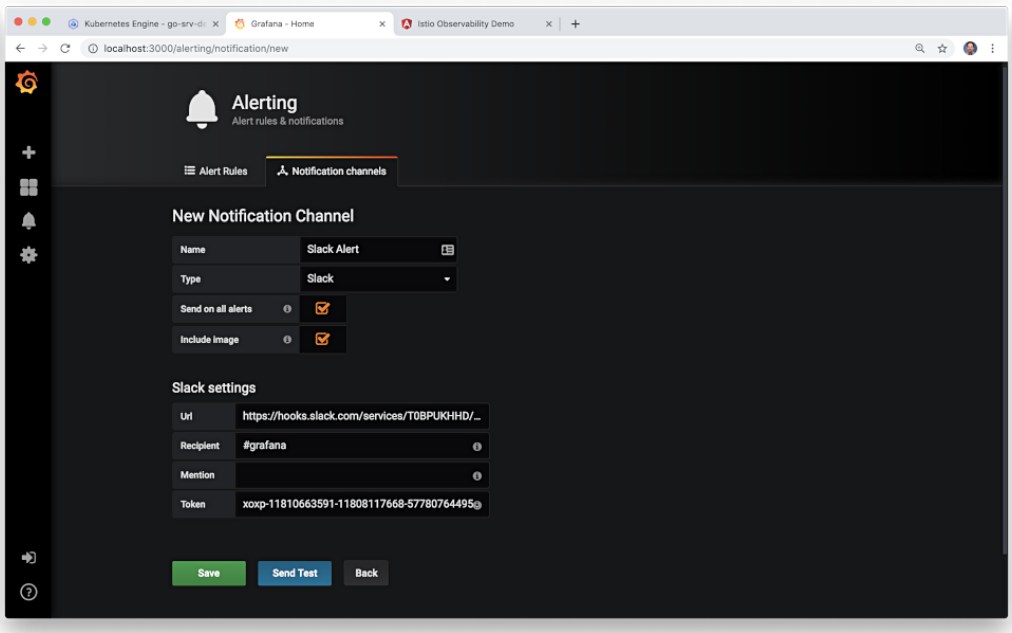
Lastly, we see the pre-configured Istio Mesh Dashboard. This dashboard is filtered to show a table view of metrics for components deployed to our GKE cluster.



(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-13_at_10_34_16_pm.png)

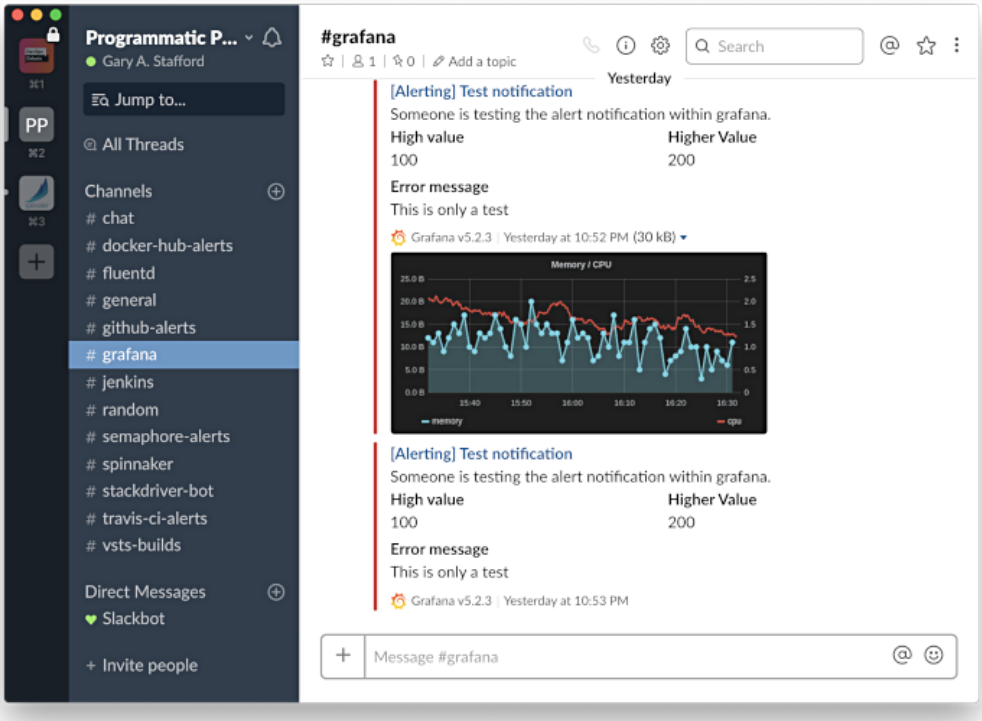
An effective observability strategy must include more than just the ability to visualize results. An effective strategy must also include the ability to detect anomalies and notify (alert) the appropriate resources or take action directly to resolve incidents. Grafana, like Prometheus, is capable of alerting and notification. You visually define alert rules for your critical metrics. Grafana will continuously evaluate metrics against the rules and send notifications when pre-defined thresholds are breached.

Prometheus supports multiple, popular notification channels (http://docs.grafana.org/alerting/notifications/#all-supported-notifier), including PagerDuty, HipChat, Email, Kafka, and Slack. Below, we see a new Prometheus notification channel, which sends alert notifications to a Slack support channel.



(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-13_at_10_55_09_pm.png)

Prometheus is able to send detailed text-based and visual notifications.



(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-14_at_6_06_22_pm-1.png)

Pillar 3: Traces

According to the Open Tracing website (https://opentracing.io/docs/overview/what-is-tracing/), distributed tracing, also called distributed request tracing, is a method used to profile and monitor applications, especially those built using a microservices architecture. Distributed tracing helps pinpoint where failures occur and what causes poor performance.

According to Istio (https://istio.io/docs/tasks/telemetry/distributed-tracing/#understanding-what-happened), although Istio proxies are able to automatically send spans, applications need to propagate the appropriate HTTP headers, so that when the proxies send span information, the spans can be correlated correctly into a single trace. To accomplish this, an application needs to collect and propagate the following headers from the incoming request to any outgoing requests.

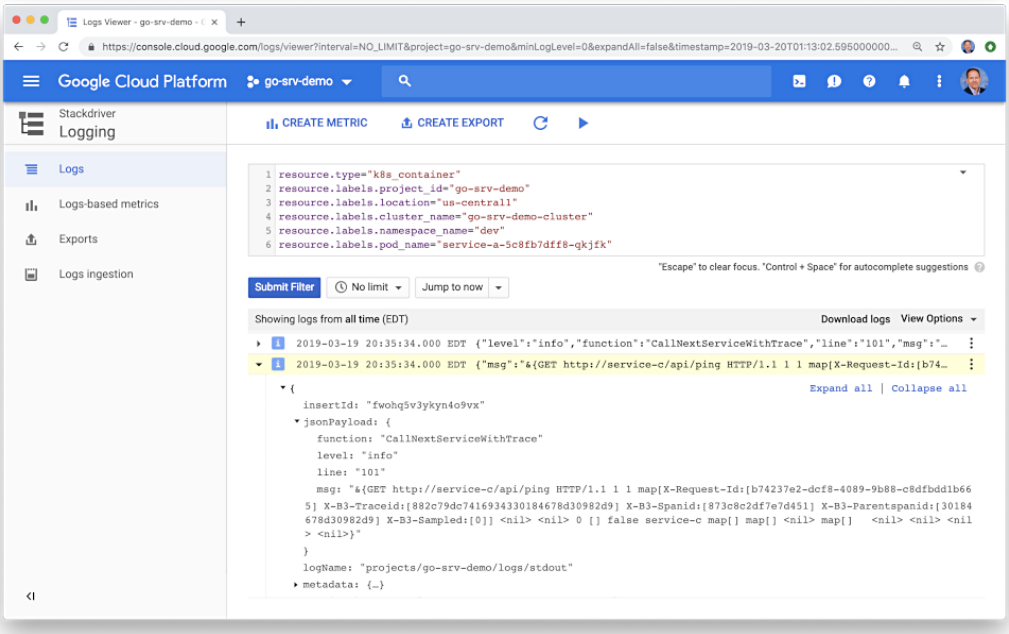
- `x-request-id`
- `x-b3-traceid`
- `x-b3-spanid`
- `x-b3-parentspanid`
- `x-b3-sampled`
- `x-b3-flags`
- `x-ot-span-context`

The `x-b3` headers originated as part of the Zipkin project. The B3 portion of the header is named for the original name of Zipkin, BigBrotherBird. Passing these headers across service calls is known as B3 propagation (https://github.com/openzipkin/b3-propagation). According to Zipkin (https://github.com/openzipkin/b3-propagation#b3-propagation), these attributes are propagated in-process, and eventually downstream (often via HTTP headers), to ensure all activity originating from the same root are collected together.

In order to demonstrate distributed tracing with Jaeger, I have modified Service A, Service B, and Service E. These are the three services that make HTTP requests to other upstream services. I have added the following code in order to propagate the headers from one service to the next. The Istio sidecar proxy (Envoy (https://www.envoyproxy.io/)) generates the first headers (https://www.envoyproxy.io/docs/envoy/latest/configuration/http_conn_man/headers#x-request-id). It is critical that you only propagate the headers that are present in the downstream request and have a value, as the code below does. Propagating an empty header will break the distributed tracing.

```
headers := []string{
  "x-request-id",
  "x-b3-traceid",
  "x-b3-spanid",
  "x-b3-parentspanid",
  "x-b3-sampled",
  "x-b3-flags",
  "x-ot-span-context",
}

for _, header := range headers {
  if r.Header.Get(header) != "" {
    req.Header.Add(header, r.Header.Get(header))
  }
}
```

Below, in the highlighted Stackdriver log entry's JSON payload, we see the required headers, propagated from the root span, which contained a value, being passed from Service A to Service C in the upstream request.

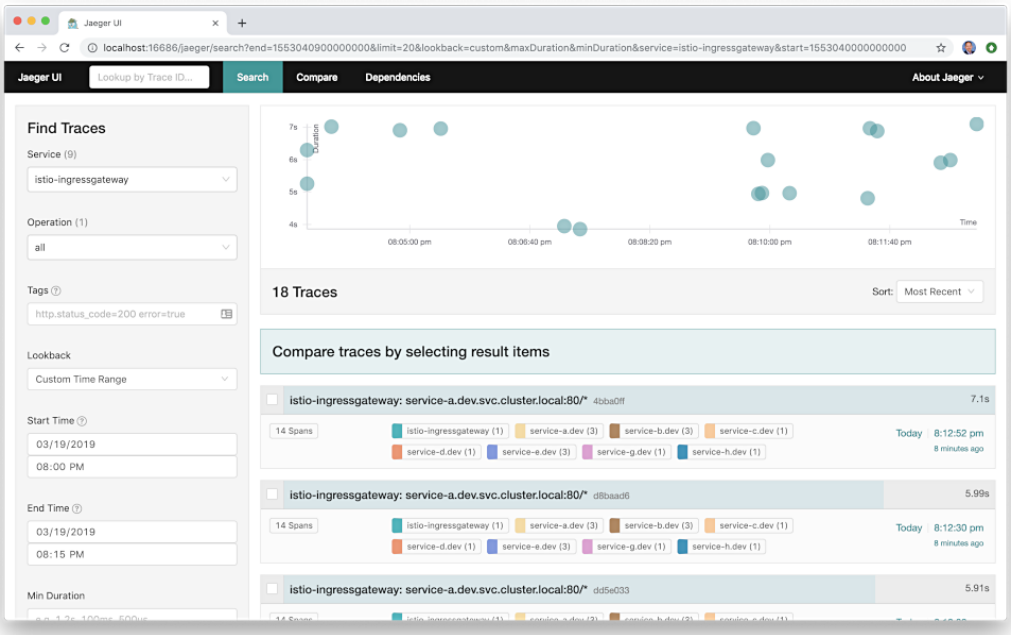


(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-19_at_11_01_26_pm.png)

Jaeger

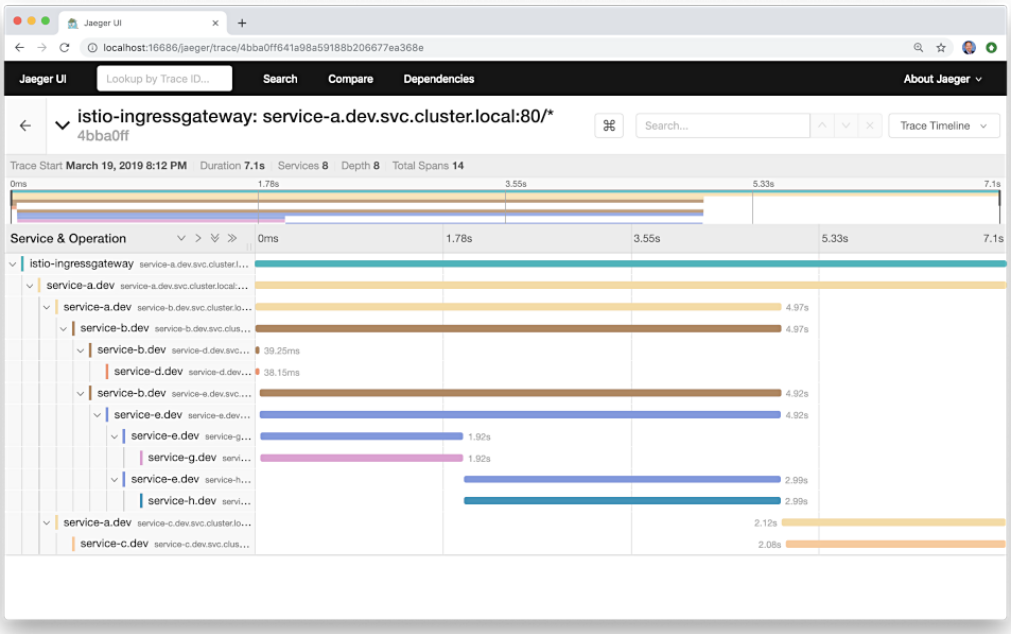
According to their website, Jaeger (https://www.jaegertracing.io/docs/1.10/), inspired by Dapper (https://research.google.com/pubs/pub36356.html) and OpenZipkin (http://zipkin.io/), is a distributed tracing system released as open source by Uber Technologies (http://uber.github.io/). It is used for monitoring and troubleshooting microservices-based distributed systems, including distributed context propagation, distributed transaction monitoring, root cause analysis, service dependency analysis, and performance and latency optimization. The Jaeger website contains a good overview of Jaeger's architecture and general tracing-related terminology.

Below we see the Jaeger UI Traces View. The UI shows the results of a search for the Istio Ingress Gateway service over a period of about forty minutes. We see a timeline of traces across the top with a list of trace results below. As discussed on the Jaeger website (https://www.jaegertracing.io/docs/1.10/architecture/), a trace is composed of spans.
A span represents a logical unit of work in Jaeger that has an operation name. A trace is an execution path through the system and can be thought of as a directed acyclic graph (https://en.wikipedia.org/wiki/Directed_acyclic_graph) (DAG) of spans (https://www.jaegertracing.io/docs/1.10/architecture#span). If you have worked with systems like Apache Spark, you are probably already familiar with DAGs.



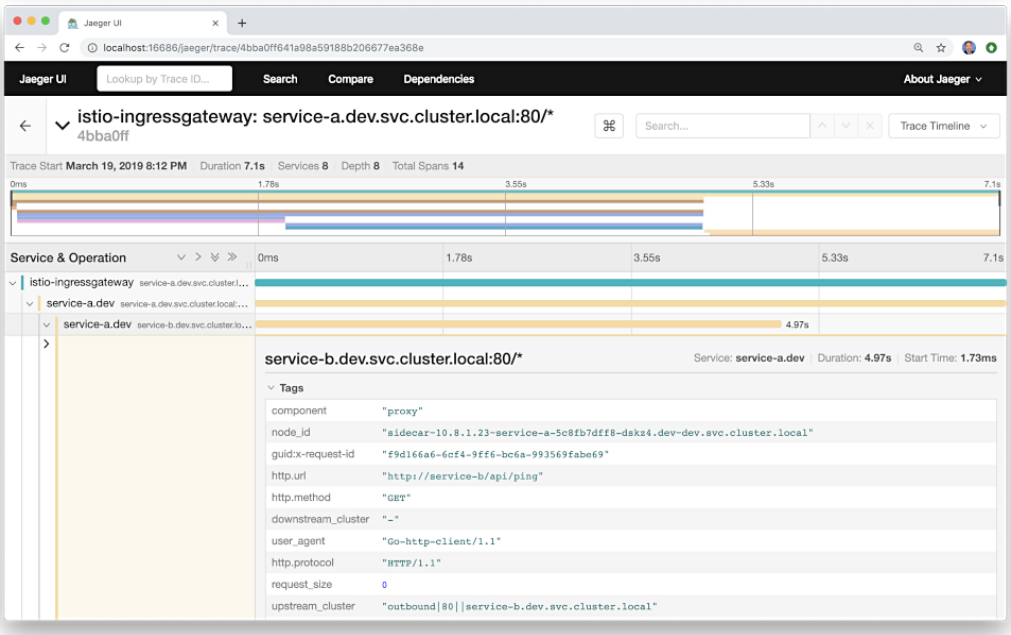
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-19_at_8_21_14_pm.png)

Below we see the Jaeger UI Trace Detail View. The example trace contains 16 spans, which encompasses eight services – seven of the eight Go-based services and the Istio Ingress Gateway. The trace and the spans each have timings. The root span in the trace is the Istio Ingress Gateway. The Angular UI, loaded in the end user's web browser, calls the mesh's edge service, Service A, through the Istio Ingress Gateway. From there, we see the expected flow of our service-to-service IPC. Service A calls Services B and C. Service B calls Service E, which calls Service G and Service H. In this demo, traces do not span the RabbitMQ message queues. This means you would not see a trace which includes a call from Service D to Service F, via the RabbitMQ.



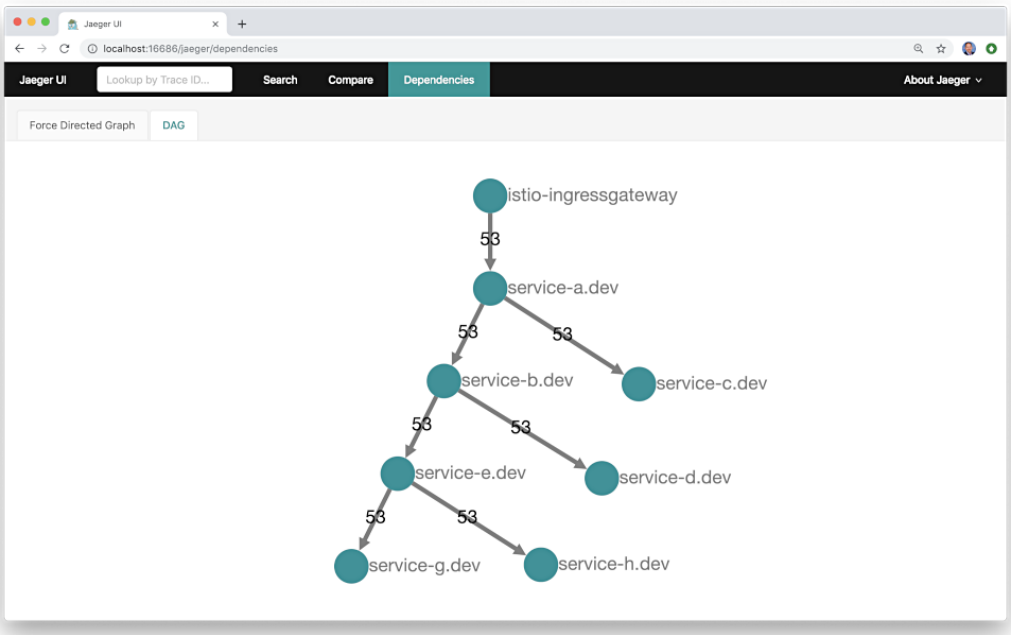
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-19_at_8_21_31_pm.png)

Within the Jaeger UI Trace Detail View, you also have the ability to drill into a single span, which contains additional metadata. Metadata includes the URL being called, HTTP method, response status, and several other headers.



(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-19_at_8_22_16_pm.png)

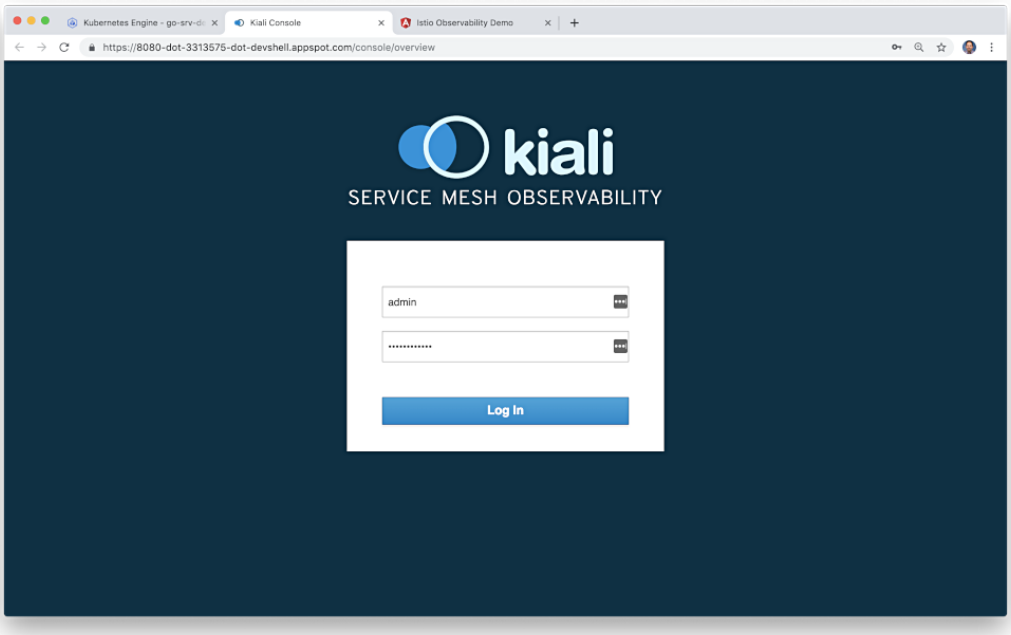
The latest version of Jaeger also includes a Compare feature and two Dependencies views, Force-Directed Graph, and DAG. I find both views rather primitive compared to Kiali, and more similar to Service Graph. Lacking access to Kiali, the views are marginally useful as a dependency graph.



(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-19_at_8_23_03_pm.png)

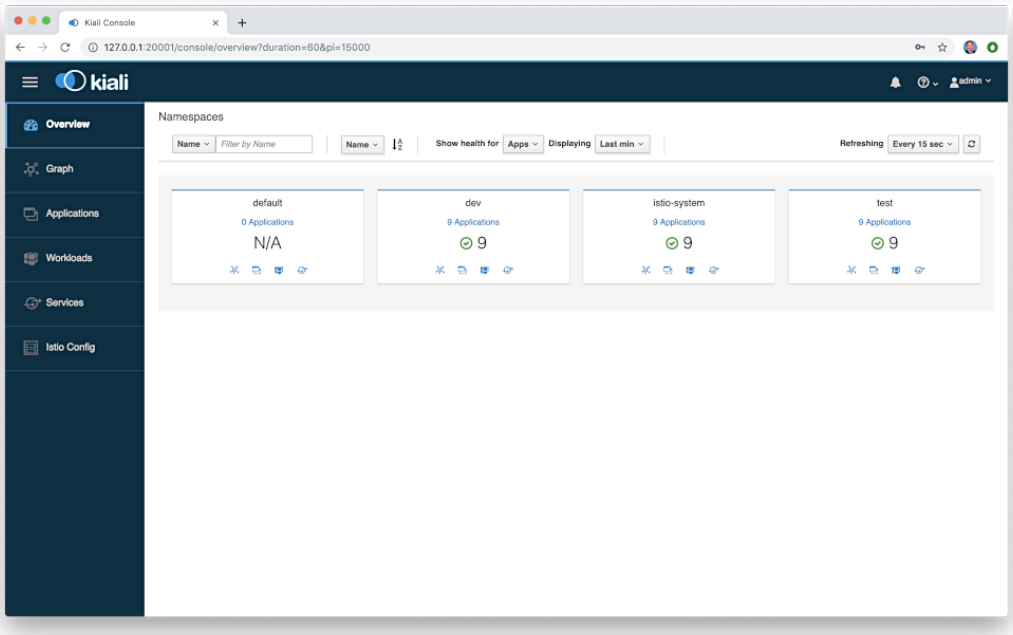
Kiali: Microservice Observability

According to their website (https://www.kiali.io/documentation/overview/), Kiali provides answers to the questions: What are the microservices in my Istio service mesh, and how are they connected? There is a common Kubernetes Secret (https://istio.io/docs/tasks/telemetry/kiali/#before-you-begin) that controls access to the Kiali API and UI. The default login is `admin`, the password is `1f2d1e2e67df`.



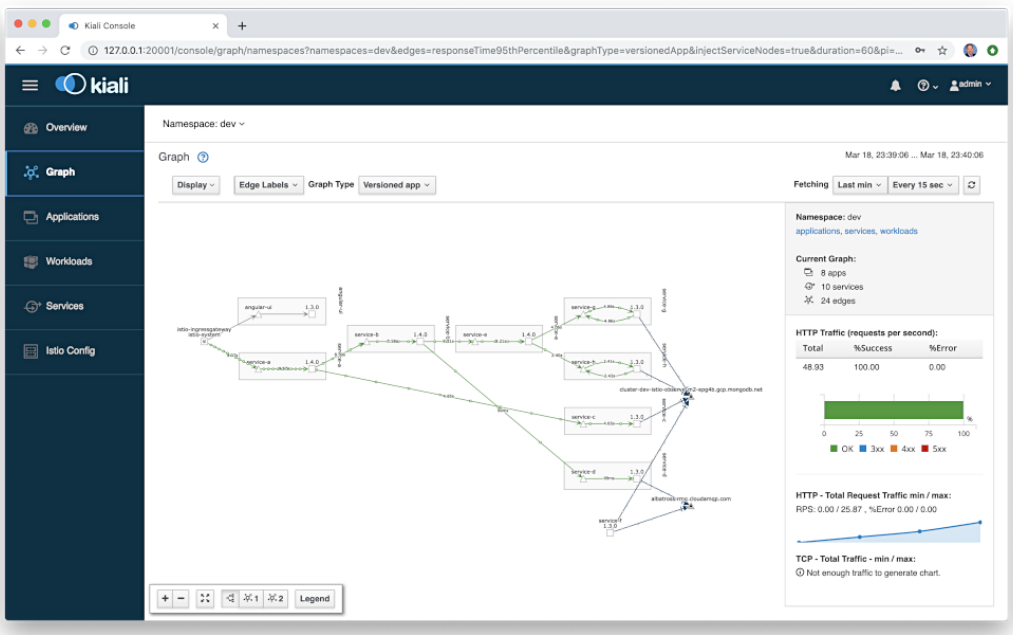
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-13_at_8_33_35_pm.png)

Logging into Kiali, we see the Overview menu entry, which provides a global view of all namespaces within the Istio service mesh and the number of applications within each namespace.



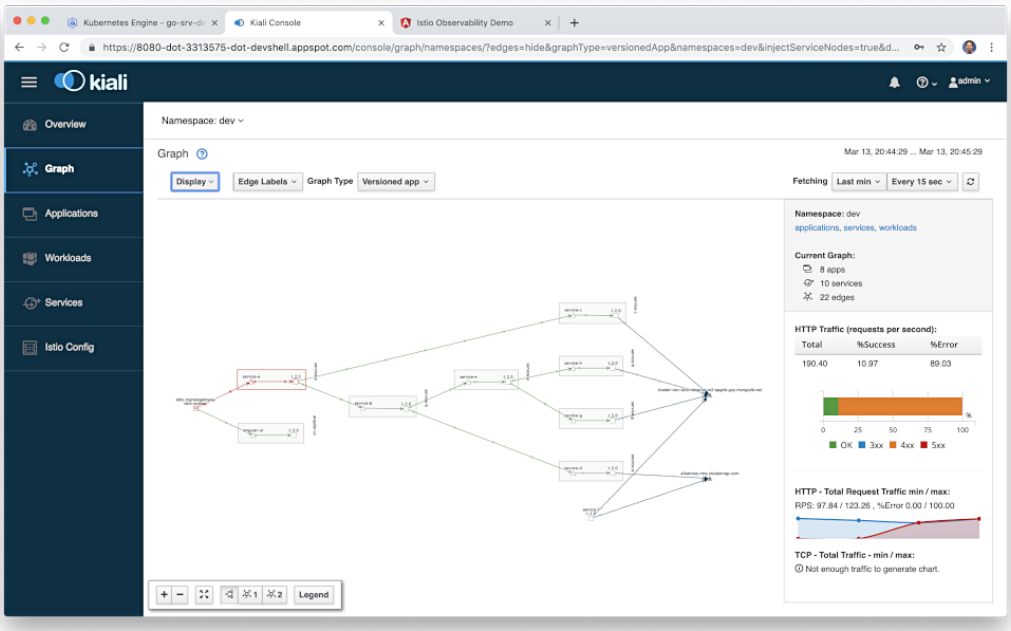
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-18_at_11_38_36_pm.png)

The Graph View in the Kiali UI is a visual representation of the components running in the Istio service mesh. Below, filtering on the cluster's `dev` Namespace, we can observe that Kiali has mapped 8 applications (workloads), 10 services, and 24 edges (a graph term). Specifically, we see the Istio Ingres Proxy at the edge of the service mesh, the Angular UI, the eight Go-based microservices and their Envoy proxy sidecars that are taking traffic (Service F did not take any direct traffic from another service in this example), the external MongoDB Atlas cluster, and the external CloudAMQP cluster. Note how service-to-service traffic flows, with Istio, from the service to its sidecar proxy, to the other service's sidecar proxy, and finally to the service.



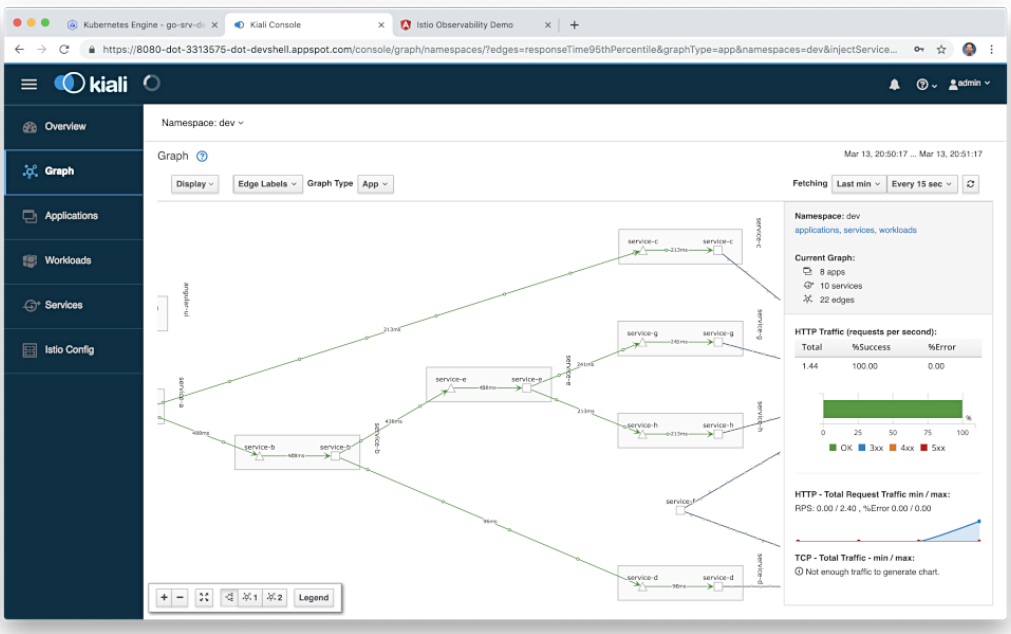
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-18_at_11_40_16_pm.png)

Below, we see a similar view of the service mesh, but this time, there are failures between the Istio Ingress Gateway and the Service A, shown in red. We can also observe overall metrics for the HTTP traffic, such as total requests/minute, errors, and status codes.



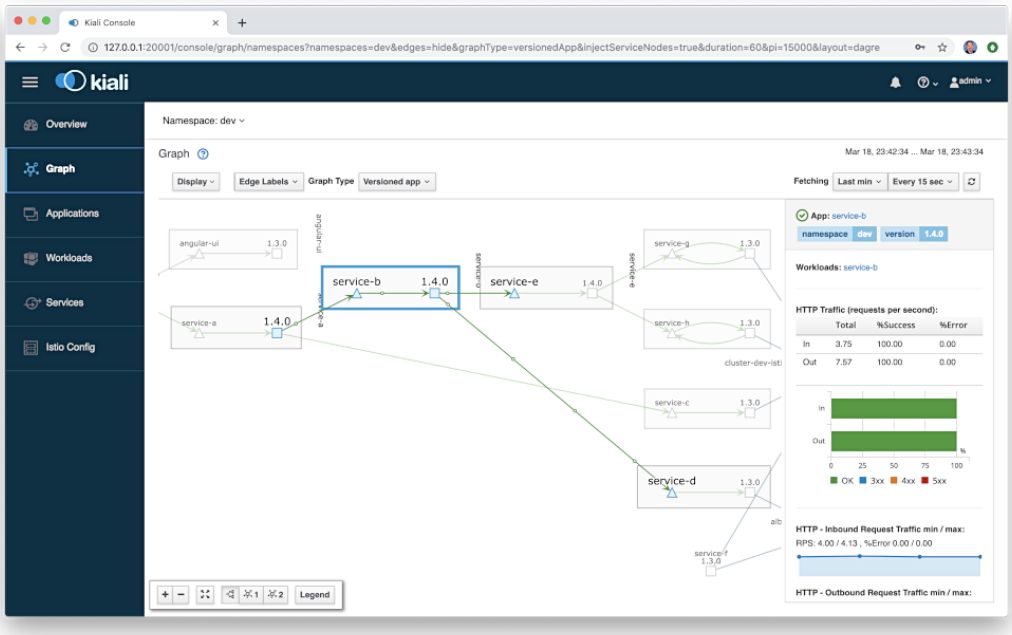
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-13_at_8_45_36_pm.png)

Kiali can also display average requests times and other metrics for each edge in the graph (the communication between two components).



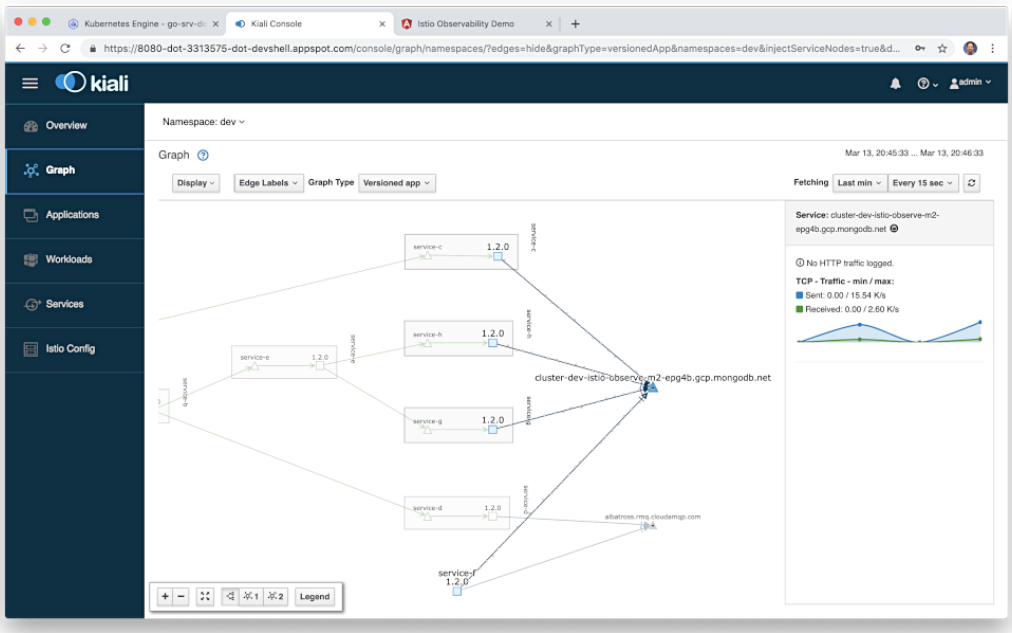
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-13_at_8_51_18_pm.png)

Kiali can also show application versions deployed, as shown below, the microservices are a combination of versions 1.3 and 1.4.



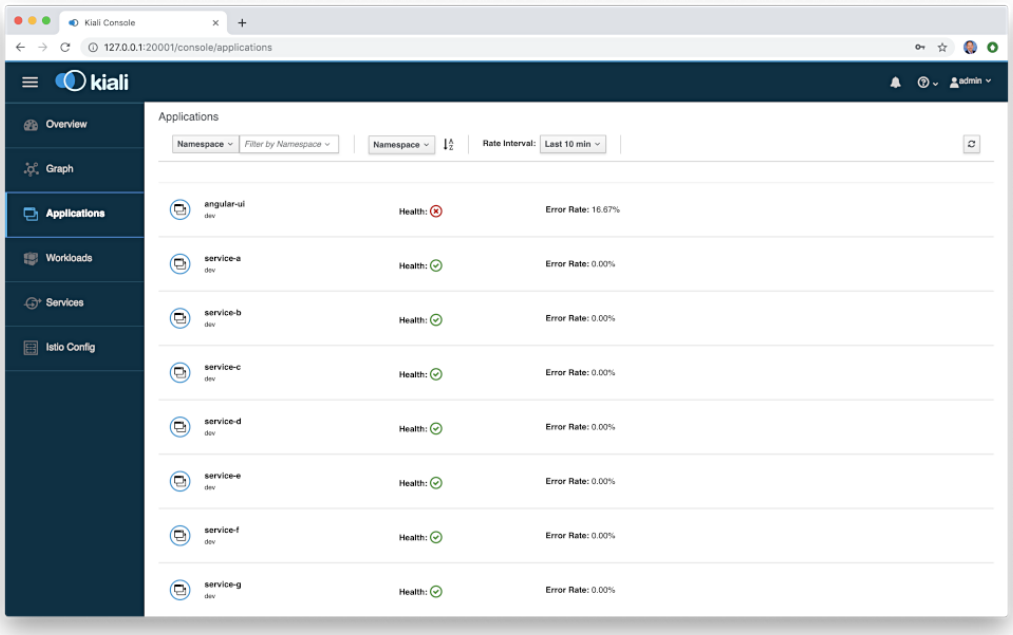
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-18_at_11_43_41_pm.png)

Focusing on the external MongoDB Atlas cluster, Kiali also allows us to view TCP traffic between the four services within the service mesh and the external cluster.



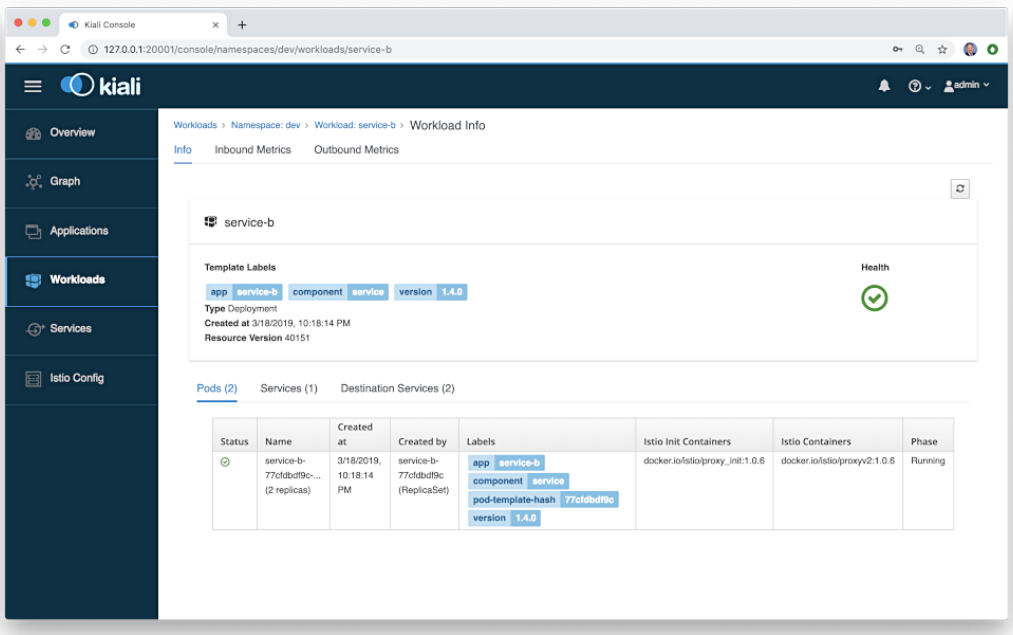
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-13_at_8_46_46_pm.png)

The Applications menu entry lists all the applications and their error rates, which can be filtered by Namespace and time interval. Here we see that the Angular UI was producing errors at the rate of 16.67%.



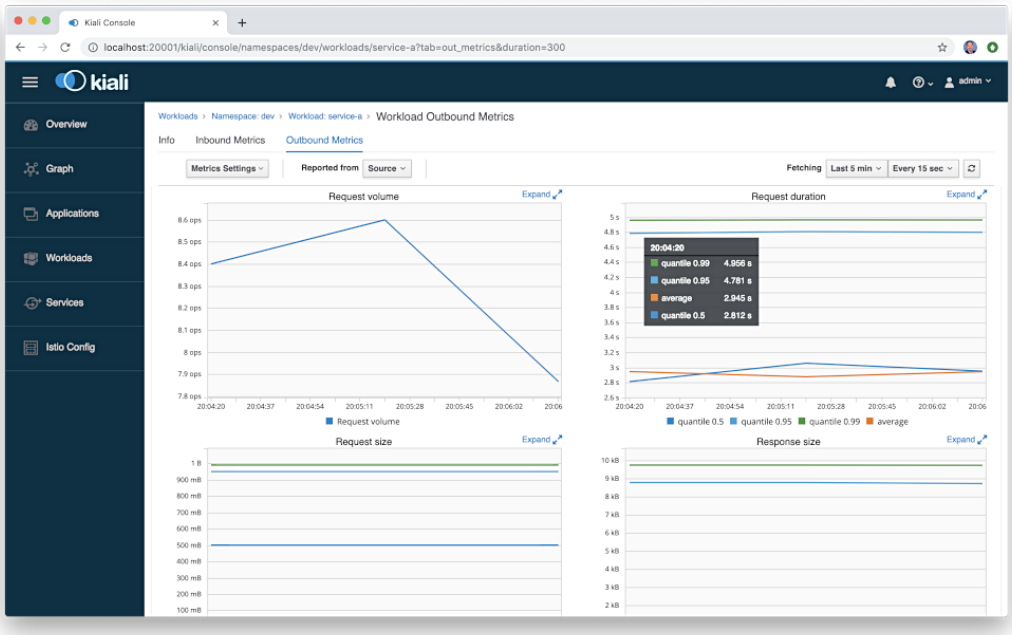
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-18_at_11_43_48_pm.png)

On both the Applications and Workloads menu entry, we can drill into a component to view additional details, including the overall health, number of Pods, Services, and Destination Services. Below, we see details for Service B in the `dev` Namespace.



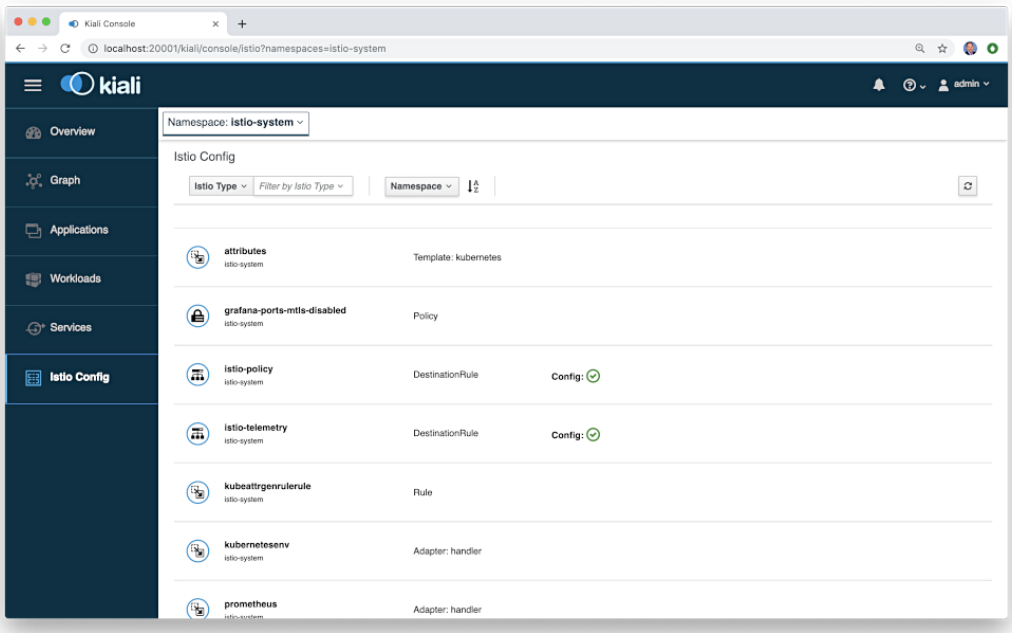
(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-18_at_11_44_37_pm.png)

The Workloads detailed view also includes inbound and outbound metrics. Below, the outbound volume, duration, and size metrics, for Service A in the `dev` Namespace.



(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-19_at_8_06_50_pm.png)

Finally, Kiali presents an Istio Config menu entry. The Istio Config menu entry displays a list of all of the available Istio configuration objects that exist in the user's environment.



(https://programmaticponderings.files.wordpress.com/2019/03/screen_shot_2019-03-19_at_8_38_08_pm.png)

Oftentimes, I find Kiali to be my first stop when troubleshooting platform issues. Once I identify the specific components or communication paths having issues, I can search the Stackdriver logs and the Prometheus metrics, through the Grafana dashboard.

Conclusion

In this two-part post, we have explored the current set of observability tools, which are part of the latest version of Istio Service Mesh. These tools included Prometheus and Grafana for metric collection, monitoring, and alerting, Jaeger for distributed tracing, and Kiali for Istio service-mesh-based microservice visualization. Combined with cloud platform-native monitoring and logging services, such as Stackdriver for Google Kubernetes Engine (GKE) on Google Cloud Platform (GCP), we have a complete observability solution for modern, distributed applications.

All opinions expressed in this post are my own and not necessarily the views of my current or past employers or their clients.

GCP , GKE , Google Cloud Platform , Grafana , Istio , Jaeger , Kiali , Logging , Metrics , Monitoring , Observability , Prometheus , Service Mesh , Stackdriver , Tracing

This entry was posted on March 21, 2019, 6:38 am and is filed under Cloud, DevOps, GCP, Go, JavaScript, Kubernetes, Software Development. You can follow any responses to this entry through RSS 2.0. You can leave a response, or trackback from your own site.

COMMENTS (2) TRACKBACKS (1)

#1 by Heiko W. Rupp (@pilhuhn) on March 21, 2019 - 12:13 pm

Thank you for your detailed article. For Kiali you write " I find Kiali to be my first stop […] , I can search the Stackdriver logs and the Prometheus metrics, through the Grafana dashboard."

How would you think Kiali could make this process more easy? For example what Prometheus metrics are missing in Kiali so that you need to go to Grafana? Likewise for logs?

#2 by Gary A. Stafford on March 24, 2019 - 9:27 pm

Once I've found the root of the issue, a service, route, or external route, I usually head to the logs to look for a specific issue.

1. Kubernetes-based Microservice Observability with Istio Service Mesh: Part 1 | Programmatic Ponderings