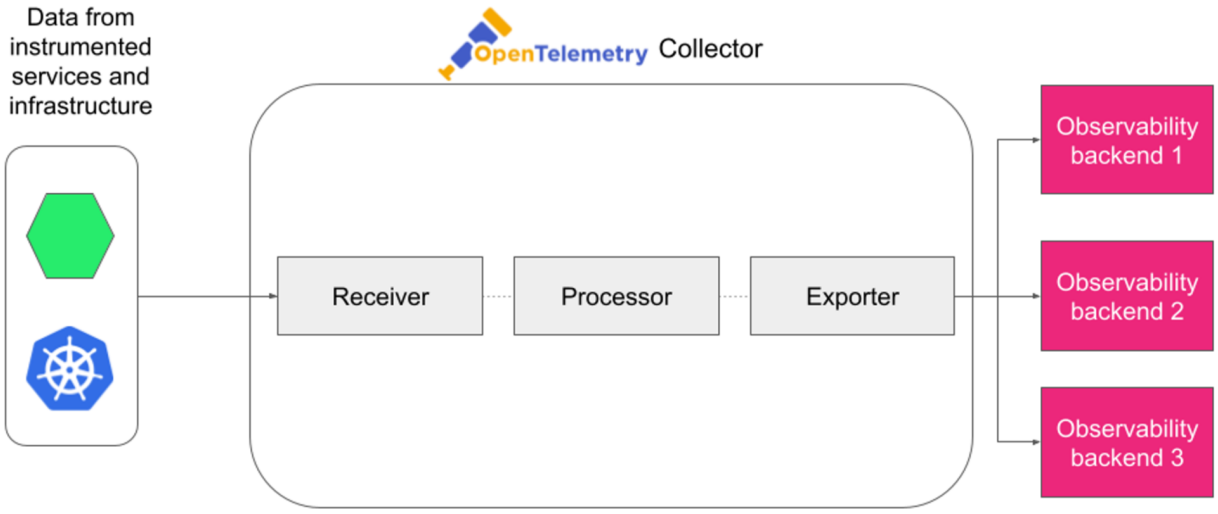
OTEL-Observability-Project

What is Open Telemetry (OTel)?

OpenTelemetry is an open-source project designed to provide a unified standard for collecting and managing telemetry data from software applications, including traces, metrics, and logs. It helps in observing and understanding the behavior of applications and systems by offering a set of APIs, libraries, and agents for instrumenting code and exporting telemetry data.



Architectural driagram:

https://opentelemetry.io/docs/demo/architecture/

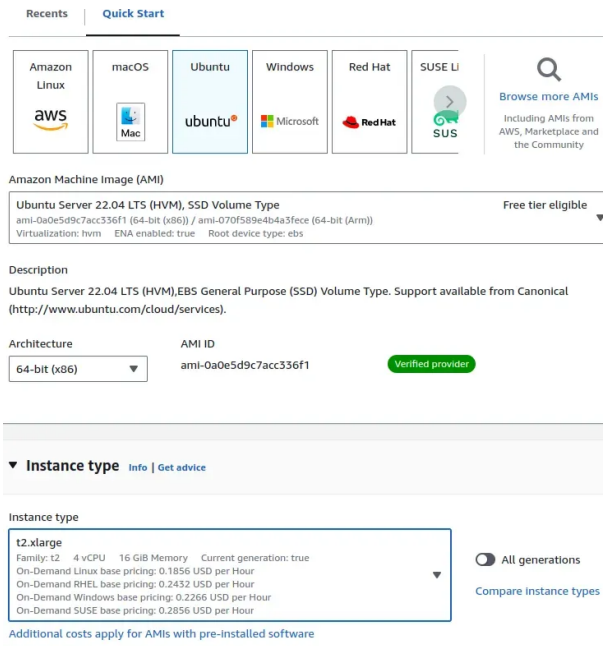
Prerequisites:

1. AWS Cloud

Steps:

Create EC2 instance with the following configurations:

- Ubuntu 22
- T2.xlarge (More than 6GB ram required)
- 15GB storage



Clone the repo:

```
git clone https://github.com/open-telemetry/opentelemetry-demo.g
cd opentelemetry-demo/
```

As a DevOps Engineer you should understand what this project is doing and how things are happening so let's understand the docker compose file responsible for

deploying the app.

Understanding the docker compose file structure!

This Docker Compose file is designed to set up an observability demo environment using various microservices. If you're new to observability, here's how you can understand this setup:

https://github.com/open-telemetry/opentelemetry-demo/blob/main/docker-compose.yml

Breakdown of the File:

- **Logging Configuration (`x-default-logging`)**:
  - This section defines how logs are handled. The logs are stored in JSON format with limits on file size (`5m`) and number of files (`2`). The `tag` option adds a name tag to each log entry, which is useful for identifying which service generated the log.
- **Networks**:
  - The `networks` section defines a custom network called `opentelemetry-demo` using the bridge driver, which allows containers to communicate with each other.
- **Services**:
  - Each service represents a different microservice in the application. These microservices work together to form a complete application.

Service Details:

Language	Automatic Instrumentation	Instrumentation Libraries	Manual Instrumentation
.NET	Accounting Service	Cart Service	Cart Service
C++			Currency Service
Go		Checkout Service, Product Catalog Service	Checkout Service, Product Catalog Service
Java	Ad Service		Ad Service
JavaScript		Frontend	Frontend, Payment Service
Kotlin		Fraud Detection Service	
PHP		Quote Service	Quote Service
Python	Recommendation Service		Recommendation Service
Ruby		Email Service	Email Service
Rust		Shipping Service	Shipping Service

Microservices in the app and the languages they are written in.

1. **Core Demo Services:** Application services written in different languages.
2. **Dependent Services:** Services that the application services depend on like *Redis, Kafka* etc.
3. **Telemetry Components:** Components that deal with the telemetry data generated by the above services like *Collector, Prometheus, Grafana, OpenSearch, Jaeger*.

- **Accounting Service (`accountingservice`)**:
  - **Image**: Specifies the Docker image used to run the service.
  - **Build**: Defines how the service is built, including the Dockerfile to use.
  - **Environment Variables**: Configures how the service interacts with other parts of the system, like setting the endpoint for sending telemetry data to an OpenTelemetry collector (`OTEL_EXPORTER_OTLP_ENDPOINT`).
  - **Dependencies**: `depends_on` ensures certain services like `otelcol` (OpenTelemetry Collector) and `kafka` are started before this service.
  - **Logging**: Uses the predefined logging configuration.
- **Ad Service (`adservice`)**:
  - Similar to the accounting service but with additional ports exposed and configured for sending logs and metrics to the observability system.

- **Cart Service (`cartservice`)**:
  - Handles shopping cart operations and interacts with other services like `checkoutservice`, all while sending telemetry data to OpenTelemetry.
- **Checkout Service (`checkoutservice`)**:
  - Manages the checkout process. It depends on multiple other services to ensure the whole checkout flow works properly. It also sends data for observability.
- **Other Services (e.g., `currencyservice`, `emailservice`, `frauddetectionservice`)**:
  - Each of these services plays a specific role in the application (like handling currency conversion, sending emails, or detecting fraud) and is configured similarly with dependencies, logging, and observability settings.
- **Frontend (`frontend`) and Frontend Proxy (`frontendproxy`)**:
  - The `frontend` service is the user-facing part of the application, while `frontendproxy` helps manage traffic between the frontend and backend services.
- **Image Provider (`imageprovider`) and Load Generator (`loadgenerator`)**:
  - The `imageprovider` supplies images to the frontend, and the `loadgenerator` simulates user traffic to test the system'ssrc/flagd/demo.flagd.json performance.

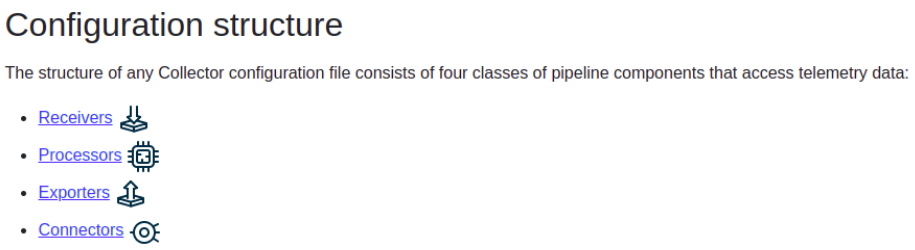
Observability in Action:

- **Telemetry Data**: Most services are configured to send data (logs, metrics, and traces) to an OpenTelemetry Collector (`otelcol`), which collects and processes this data, making it available for analysis.
- **Dependencies**: The `depends_on` condition ensures that services are started in the right order, crucial for a distributed system to function properly.

We can also check how the OTEL variables are being passed as environment variables for the core demo and dependent services. The config files and the code for all these are in the `/src` folder. You can go ahead and look at how each

service is instrumented considering the language and this documentation here helps us to better understand the instrumentation for each service in detail.

Otel Collector



Receivers: Collect telemetry data (traces, metrics, logs) from various sources (e.g., applications, services, or endpoints).

Exporters: Send the collected telemetry data to external systems or storage (e.g., logging systems, metrics platforms).

Processors: Transform or modify the telemetry data between collection and export (e.g., batch processing, filtering, or data enrichment).

Connectors (spanmetrics): Extracts metrics from trace data (span metrics) for further processing or export.

Service: Defines how telemetry data flows through the system, specifying which receivers, processors, and exporters are used for traces, metrics, and logs.

https://github.com/open-telemetry/opentelemetry-demo/blob/main/src/otelcollector/otelcol-config.yml

▼ **otelcol-config.yaml file explained!**

This file is a configuration for an OpenTelemetry Collector, which is used to collect, process, and export telemetry data (traces, metrics, logs) from various sources. Below is a breakdown of each section:

1. Receivers

- `otlp:`

  The OpenTelemetry Protocol (OTLP) receiver is configured to accept telemetry data over `gRPC` and `HTTP`. The `endpoint` values are determined by environment variables `OTEL_COLLECTOR_HOST`, `OTEL_COLLECTOR_PORT_GRPC`, and `OTEL_COLLECTOR_PORT_HTTP`. The `CORS` configuration allows requests from any HTTP or HTTPS origin.

- `httpcheck/frontendproxy:`

  This receiver checks the availability of the `frontendproxy` service by sending HTTP requests to the endpoint `http://frontendproxy:${env:ENVOY_PORT}`.

- `docker_stats:`

  This receiver collects metrics related to Docker containers by connecting to Docker through the UNIX socket `/var/run/docker.sock`.

- `redis:`

  This receiver collects metrics from a Redis instance running at the endpoint `valkey-cart:6379`, authenticating with the username `valkey`. It collects data every `10s`.

- `hostmetrics:`

  This receiver gathers various host-level metrics, such as CPU, disk, load, filesystem, memory, network, paging, and processes. Some metrics are filtered based on mount points and filesystem types.

- `prometheus:`

  This receiver scrapes metrics from the OpenTelemetry Collector itself using Prometheus. It scrapes every `10s` from the target `0.0.0.0:8888`.

2. Exporters

- `debug:`

This exporter is likely used for debugging purposes, exporting data to a local or testing environment.

- `otlp:`

  Exports traces to a Jaeger instance running at `jaeger:4317` with `TLS` but marked as `insecure`, meaning it doesn't enforce strict certificate checks.

- `otlphttp/prometheus:`

  Exports metrics to Prometheus using the OTLP over HTTP at the endpoint `http://prometheus:9090/api/v1/otlp`, also marked as `insecure`.

- `opensearch:`

  Exports logs to an OpenSearch instance at `http://opensearch:9200`. Logs are stored in an index named `otel`.

3. Processors

- `batch:`

  This processor batches data before exporting it, which helps in reducing the number of requests sent to the exporters.

- `transform:`

  This processor modifies trace data. It contains statements to replace certain parts of trace span names, such as removing query parameters (`\\\\?.*`) and standardizing API endpoint names (e.g., `GET /api/products/{productId}`).

4. Connectors

- `spanmetrics:`
  This connector is used to extract metrics from trace data, allowing for metrics such as request latency to be derived from traces.

5. Service

- `pipelines:`
  Defines the pipelines for processing and exporting telemetry data:
  - `traces:`

    Receivers: [otlp]

Processors: [transform, batch]

Exporters: [otlp, debug, spanmetrics]

This pipeline handles trace data, processes it with the `transform` and `batch` processors, and exports it using the `otlp`, `debug`, and `spanmetrics` exporters.

- `metrics:`

  Receivers: [hostmetrics, docker_stats, httpcheck/frontendproxy, otlp, prometheus, redis, spanmetrics]

  Processors: [batch]

  Exporters: [otlphttp/prometheus, debug]

  This pipeline handles metrics, processes them with the `batch` processor, and exports them to Prometheus and the debug endpoint.

- `logs:`

  Receivers: [otlp]

  Processors: [batch]

  Exporters: [opensearch, debug]

  This pipeline handles log data, processes it with the `batch` processor, and exports it to OpenSearch and the debug endpoint.

Summary

This file configures an OpenTelemetry Collector to gather telemetry data from various sources, process it, and export it to different backends like Jaeger, Prometheus, and OpenSearch. Each pipeline is responsible for a specific type of telemetry data (traces, metrics, logs), ensuring that the data is collected, processed, and exported according to the defined configuration.

Now that you understood everything, lets start with deploying the application and then OBSERVE it!

Install Docker

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" |
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io do
```

Now start the application:

```
sudo docker compose up --force-recreate --remove-orphans --de
```

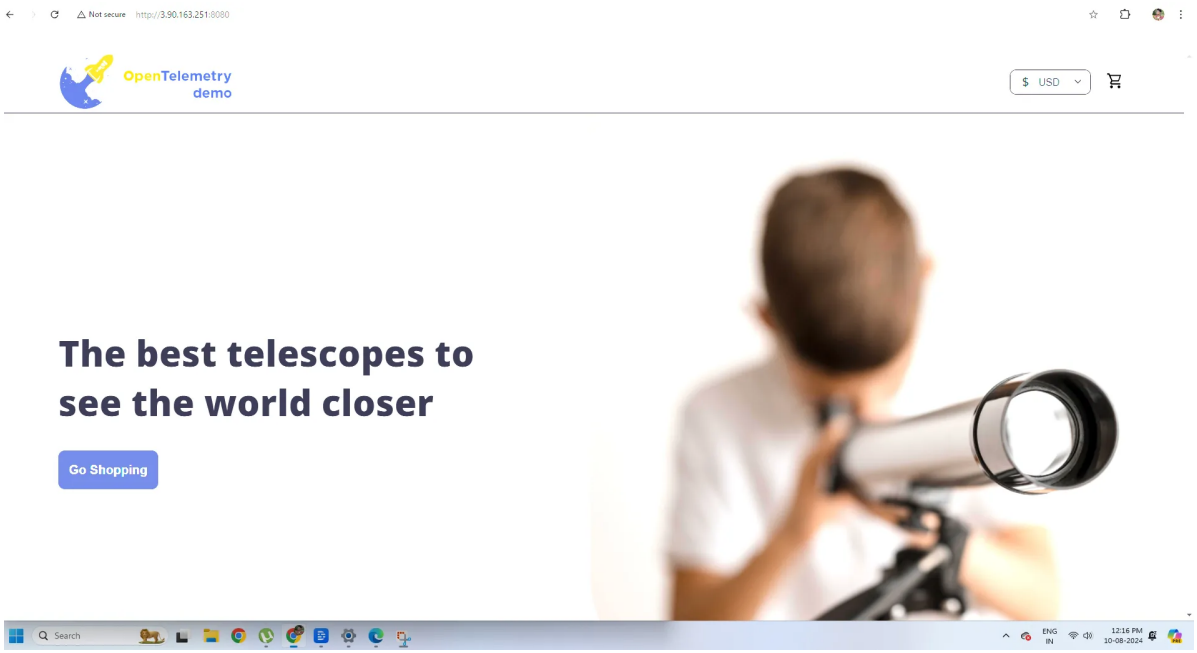
- After a while all your containers should have been created and started.

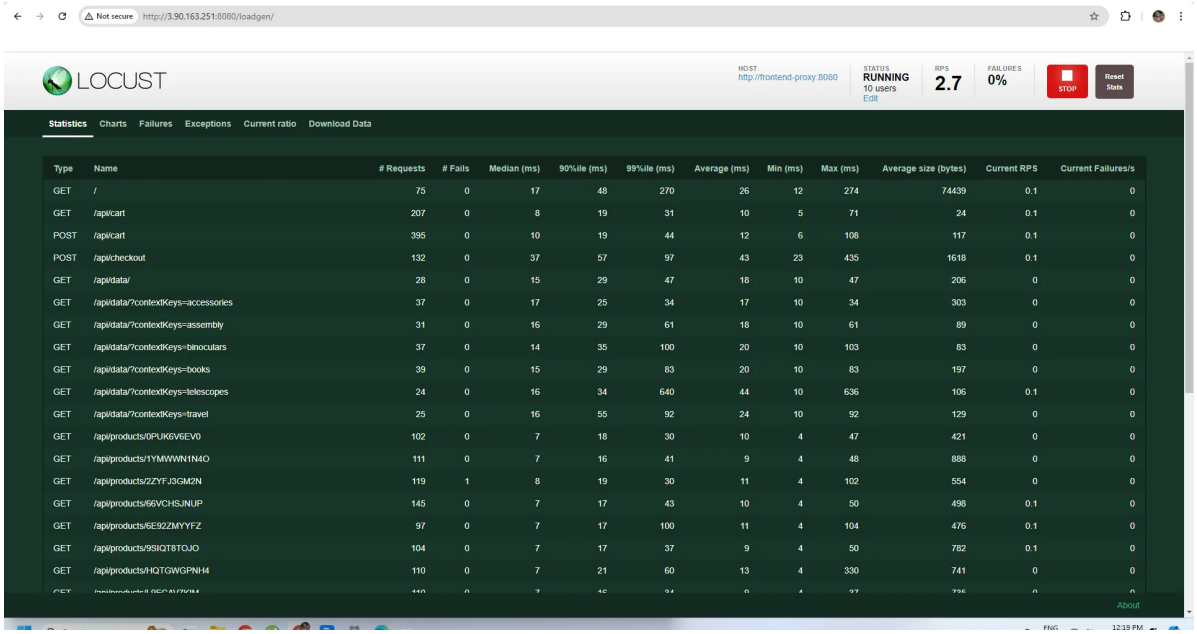
> ⚠ **NOTE**: If you **kafka** container is unhealthy, it means the RAM allocation wasn't adequate to run the containers well, and if you are using 8 GB ram, you might have to run it several times before it will become healthy, if you are using less than 8GB ram, then you have to allocate more memory to your machine.

Once all the containers are started, we can see the access them:

- Web store: http://IP:8080/

- Grafana: http://IP:8080/grafana/
- Load Generator UI: http://IP:8080/loadgen/
- Jaeger UI: http://IP:8080/jaeger/ui/



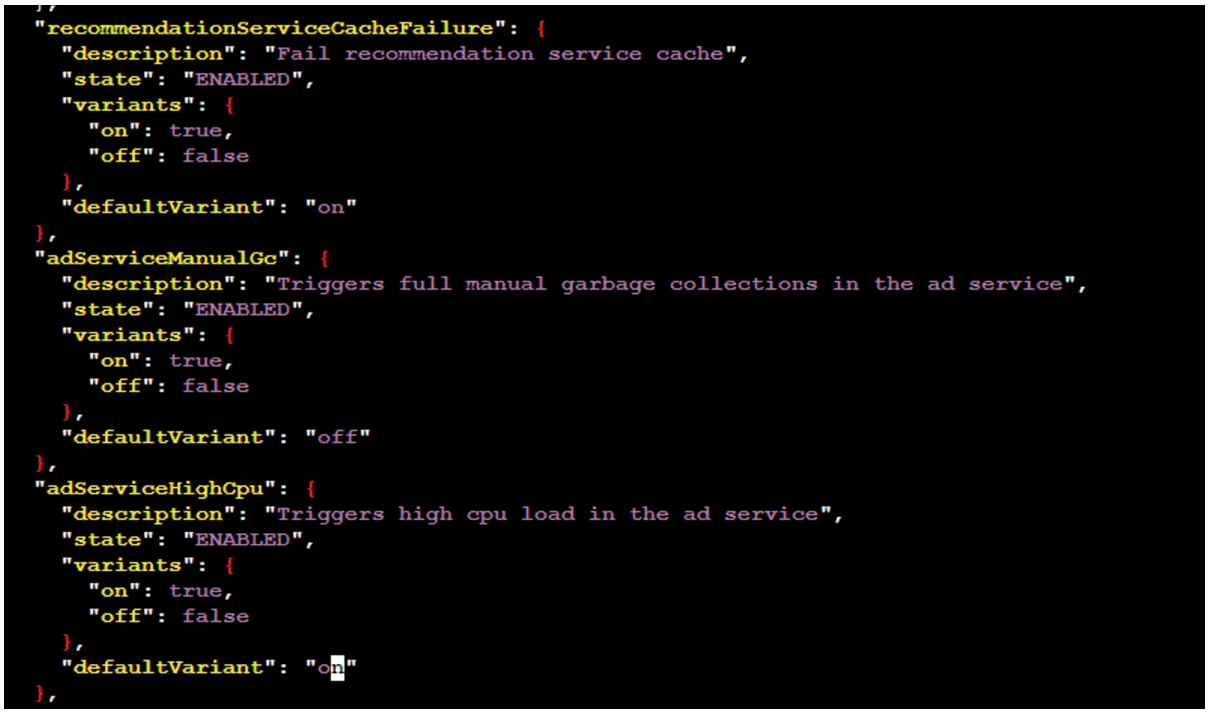


Feature Flags

The demo provides several feature flags that you can use to simulate different scenarios.

https://opentelemetry.io/docs/demo/feature-flags/

Flag values are stored in the `src/flagd/demo.flagd.json` file. To enable a flag, change the `defaultVariant` value in the config file for a given flag to “on”.

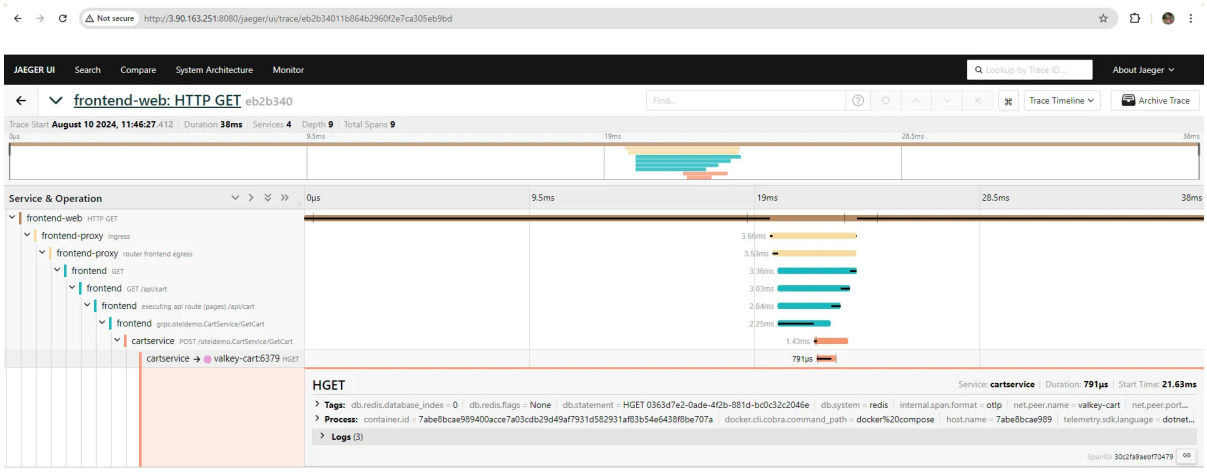


View and Analyse with the Jaeger UI

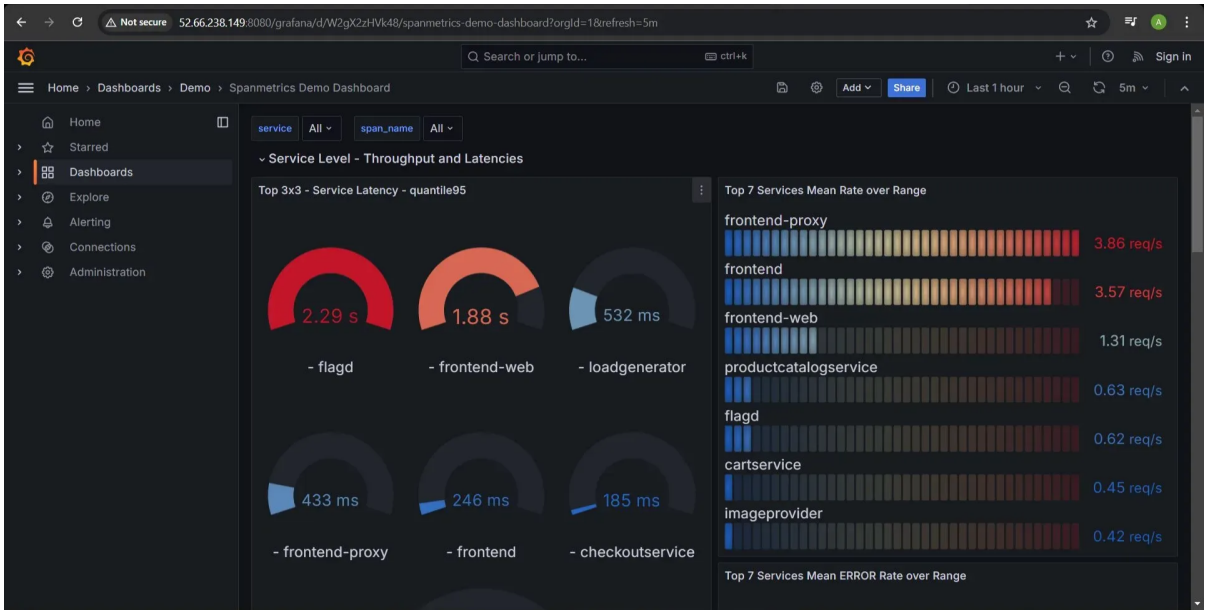
With the `adServcieFailure` feature flag enabled, let’s see how we can use **Jaeger** to diagnose the issue to determine the root cause. Remember, that the service will ***generate an error for GetAds 1/10th of the time.***

> Jaeger is usually the first tool you get in contact with when you start getting into the world of Distributed Tracing. With Jaeger, we can visualise the whole chain of events. With this

> visibility we can easier isolate the problem when something goes wrong.



Metrics on Grafana



By following these steps, you can set up and explore an observability demo environment using OpenTelemetry and Docker. This setup will help you

understand the intricacies of distributed tracing and how to monitor and diagnose issues in microservices-based applications.