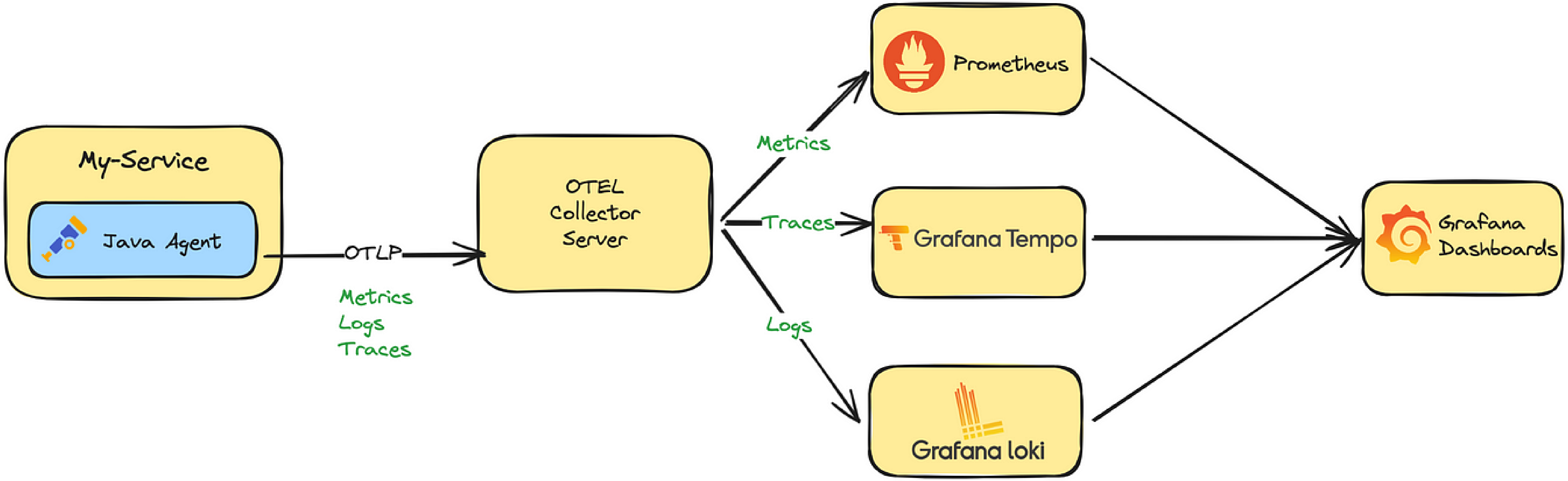
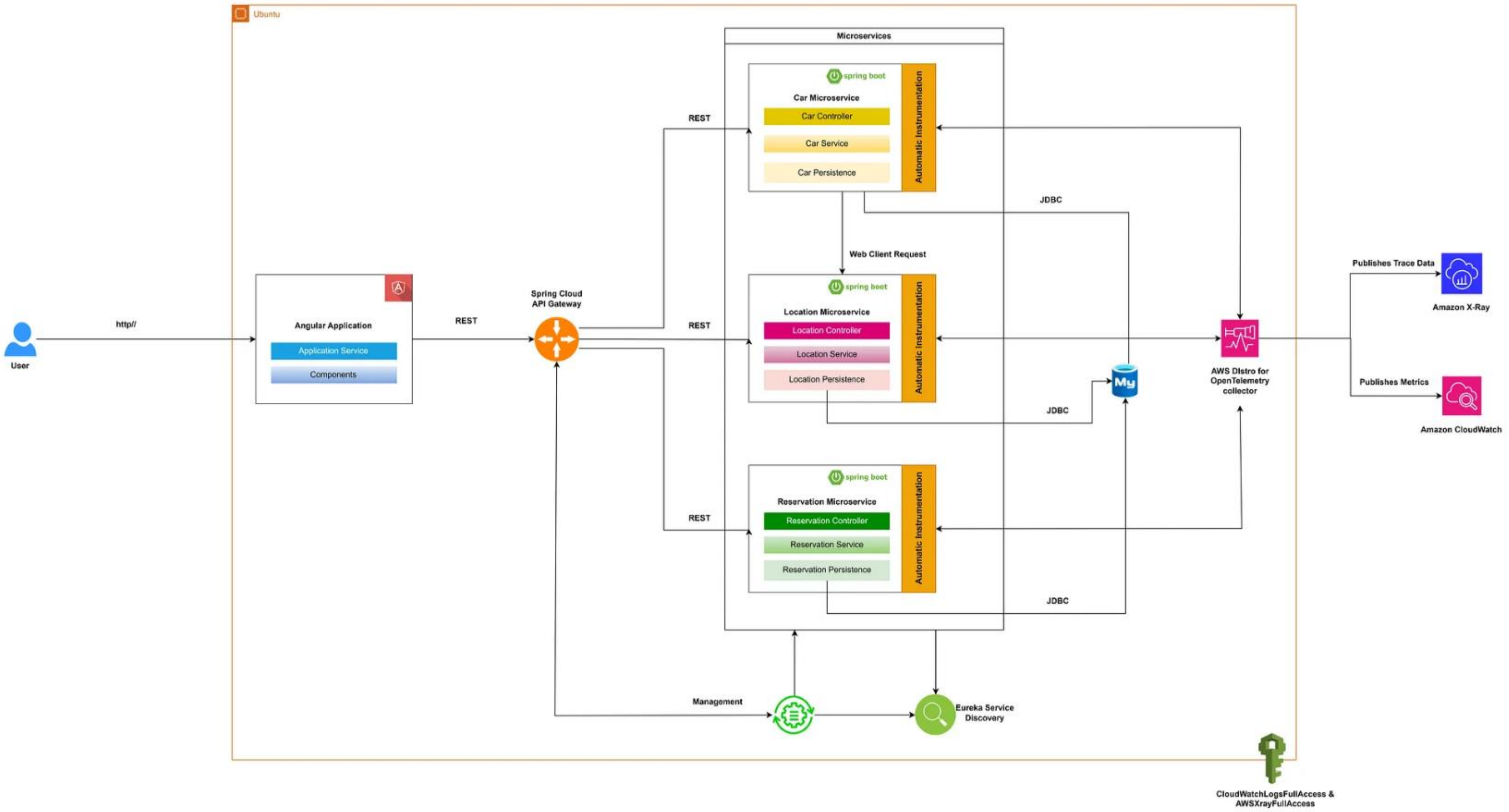
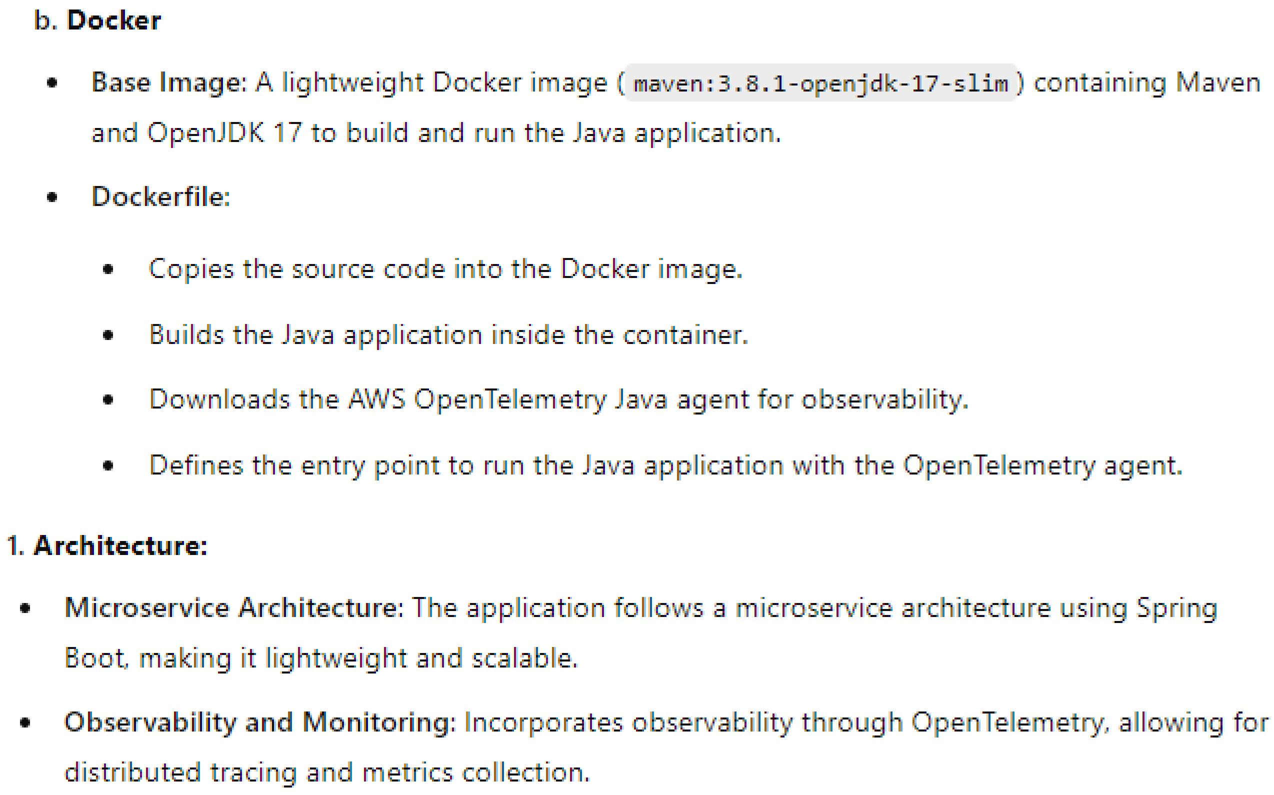
Enterprise Essentials

Observability - 2

Ensar


My-Service
Java Agent
OTLP
Metrics
Logs
Traces
OTEL
Collector
Server
Metrics
Traces
Logs
Prometheus
Grafana Tempo
Grafana loki
Grafana
Dashboards


Ubuntu
Microservices
spring boot
Car Microservice
Car Controller
Car Service
Car Persistence
Automatic Instrumentation
REST
JDBC
Web Client Request
spring boot
Location Microservice
Location Controller
Location Service
Location Persistence
Automatic Instrumentation
REST
JDBC
spring boot
Reservation Microservice
Reservation Controller
Reservation Service
Reservation Persistence
Automatic Instrumentation
REST
JDBC
User
http//
Angular Application
Application Service
Components
REST
Spring Cloud API Gateway
AWS DIstro for OpenTelemetry collector
Publishes Trace Data
Amazon X-Ray
Publishes Metrics
Amazon CloudWatch
Management
Eureka Service Discovery
CloudWatchLogsFullAccess & AWSXrayFullAccess

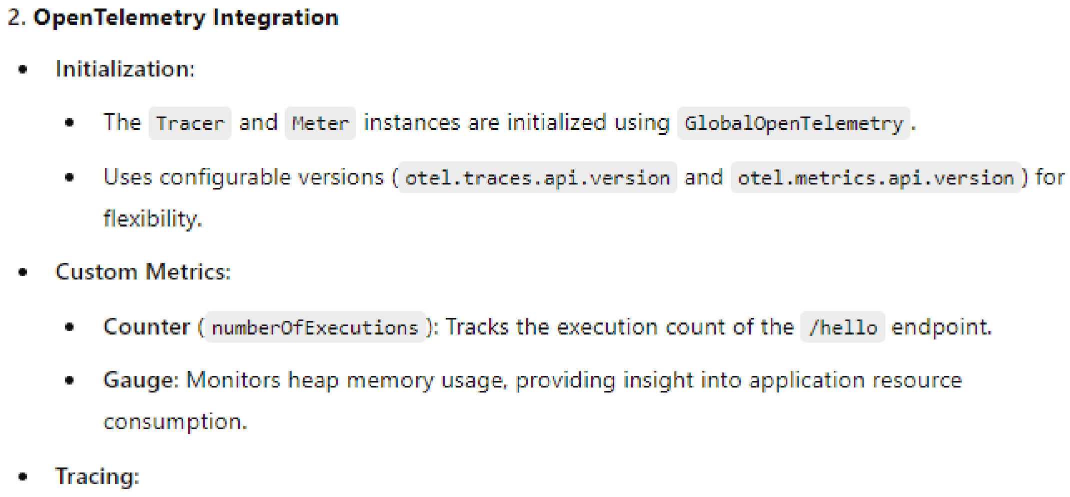


b. **Docker**

- **Base Image:** A lightweight Docker image (`maven:3.8.1-openjdk-17-slim`) containing Maven and OpenJDK 17 to build and run the Java application.
- **Dockerfile:**
  - Copies the source code into the Docker image.
  - Builds the Java application inside the container.
  - Downloads the AWS OpenTelemetry Java agent for observability.
  - Defines the entry point to run the Java application with the OpenTelemetry agent.

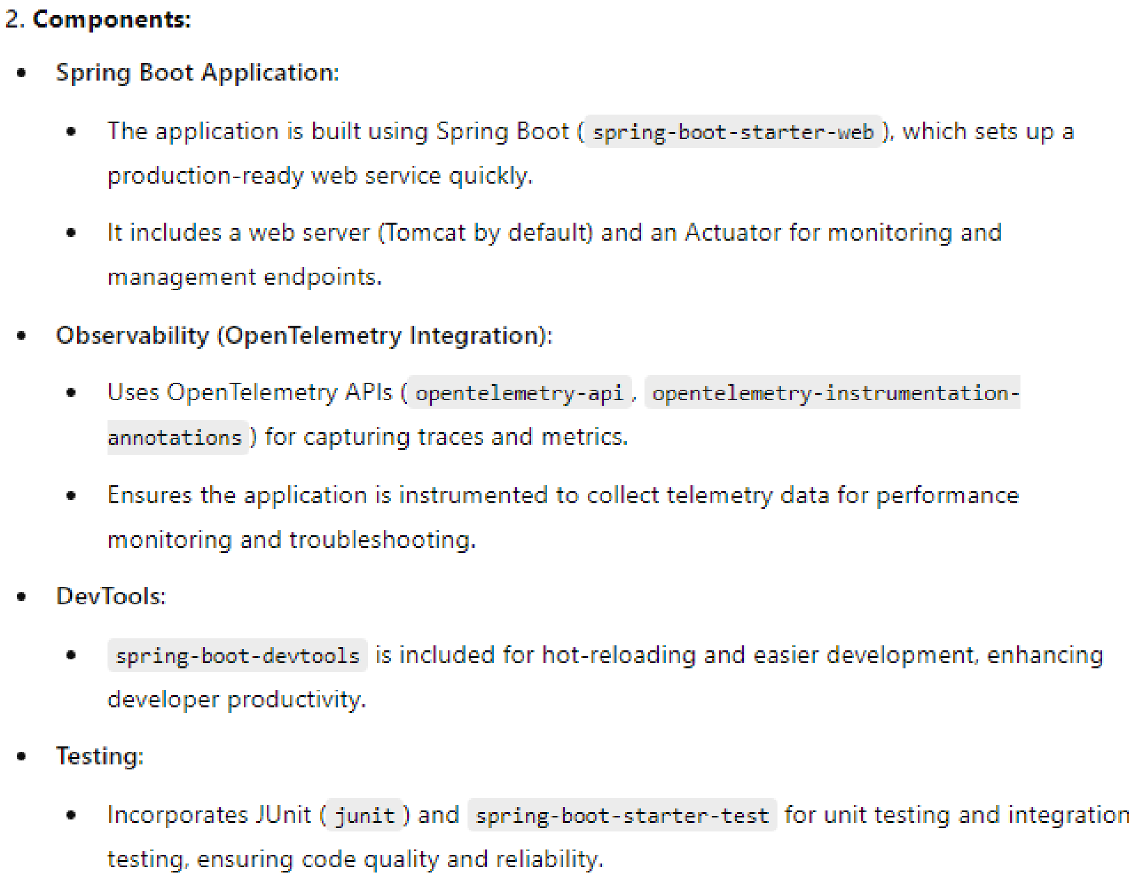
1. **Architecture:**

- **Microservice Architecture:** The application follows a microservice architecture using Spring Boot, making it lightweight and scalable.
- **Observability and Monitoring:** Incorporates observability through OpenTelemetry, allowing for distributed tracing and metrics collection.



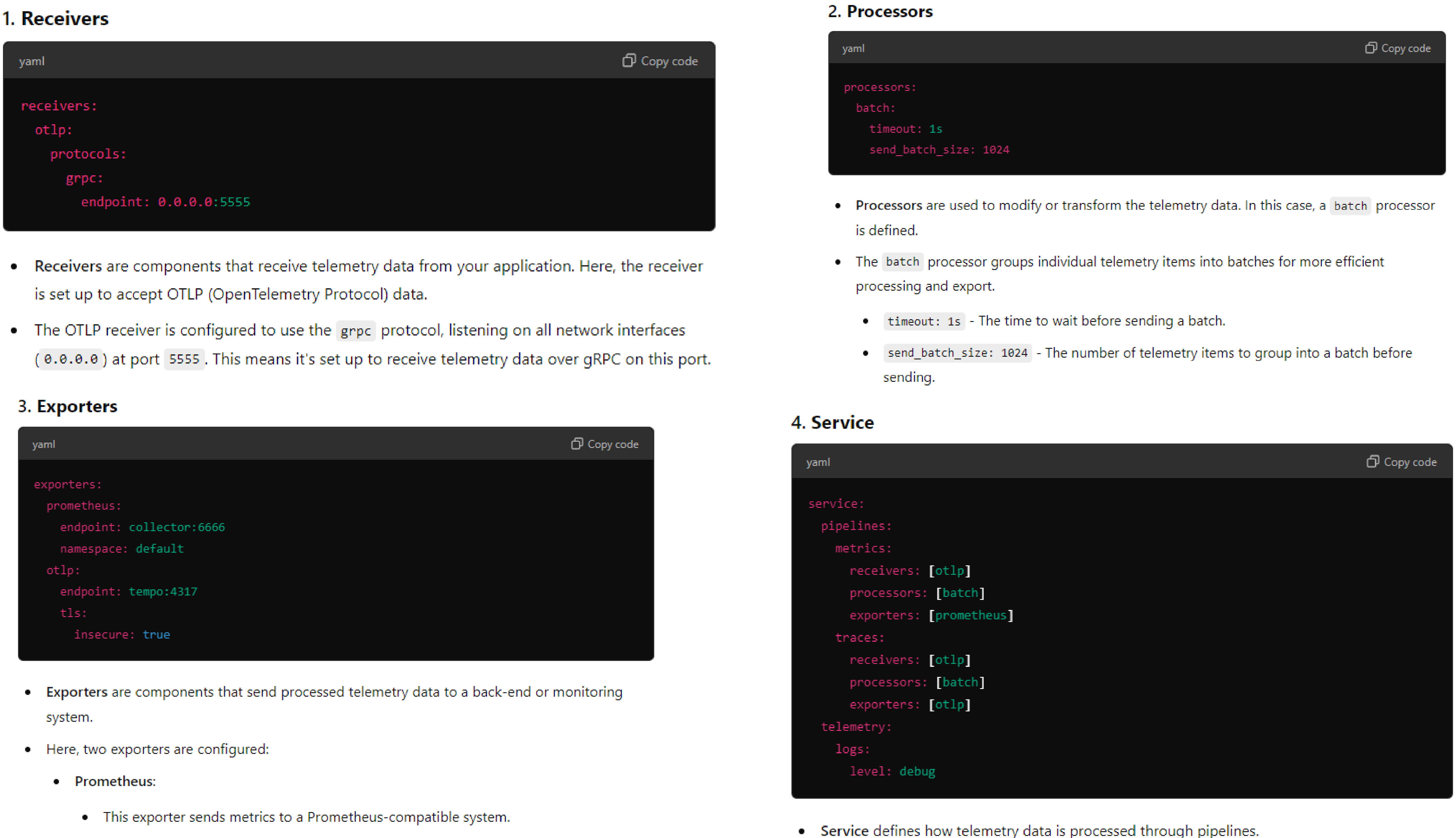
2. **OpenTelemetry Integration**

- Initialization:
  - The `Tracer` and `Meter` instances are initialized using `GlobalOpenTelemetry`.
  - Uses configurable versions (`otel.traces.api.version` and `otel.metrics.api.version`) for flexibility.
- Custom Metrics:
  - **Counter** (`numberOfExecutions`): Tracks the execution count of the `/hello` endpoint.
  - **Gauge**: Monitors heap memory usage, providing insight into application resource consumption.
- Tracing:



2. **Components:**

- **Spring Boot Application:**
  - The application is built using Spring Boot (`spring-boot-starter-web`), which sets up a production-ready web service quickly.
  - It includes a web server (Tomcat by default) and an Actuator for monitoring and management endpoints.
- **Observability (OpenTelemetry Integration):**
  - Uses OpenTelemetry APIs (`opentelemetry-api`, `opentelemetry-instrumentation-annotations`) for capturing traces and metrics.
  - Ensures the application is instrumented to collect telemetry data for performance monitoring and troubleshooting.
- **DevTools:**
  - `spring-boot-devtools` is included for hot-reloading and easier development, enhancing developer productivity.
- **Testing:**
  - Incorporates JUnit (`junit`) and `spring-boot-starter-test` for unit testing and integration testing, ensuring code quality and reliability.



1. Receivers

```yaml
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:5555
```

- **Receivers** are components that receive telemetry data from your application. Here, the receiver is set up to accept OTLP (OpenTelemetry Protocol) data.
- The OTLP receiver is configured to use the `grpc` protocol, listening on all network interfaces (`0.0.0.0`) at port `5555`. This means it's set up to receive telemetry data over gRPC on this port.

2. Processors

```yaml
processors:
  batch:
    timeout: 1s
    send_batch_size: 1024
```

- **Processors** are used to modify or transform the telemetry data. In this case, a `batch` processor is defined.
- The `batch` processor groups individual telemetry items into batches for more efficient processing and export.
  - `timeout: 1s` - The time to wait before sending a batch.
  - `send_batch_size: 1024` - The number of telemetry items to group into a batch before sending.

3. Exporters

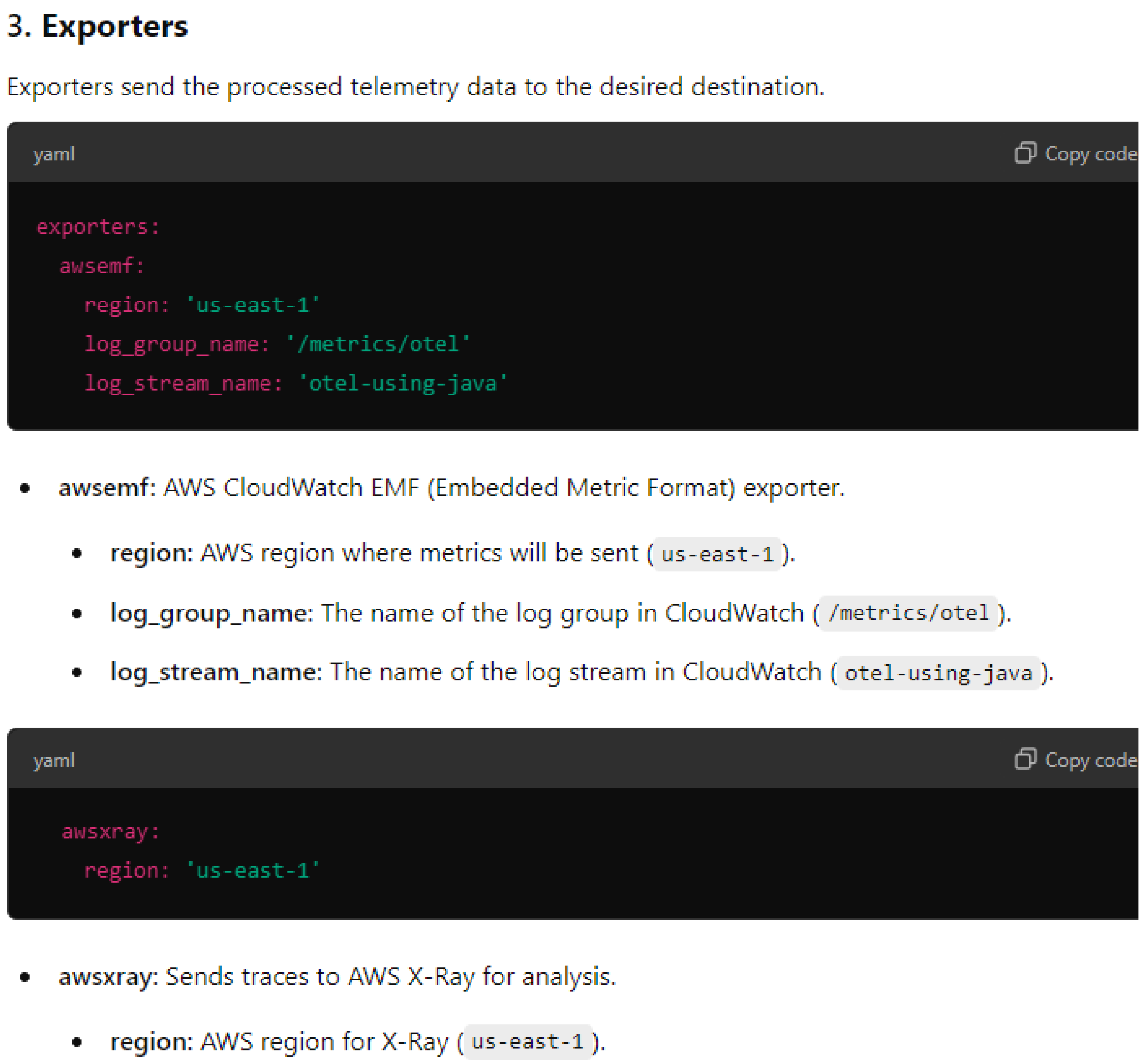
```yaml
exporters:
  prometheus:
    endpoint: collector:6666
    namespace: default
  otlp:
    endpoint: tempo:4317
    tls:
      insecure: true
```

- **Exporters** are components that send processed telemetry data to a back-end or monitoring system.
- Here, two exporters are configured:
  - **Prometheus:**
    - This exporter sends metrics to a Prometheus-compatible system.

4. Service

```yaml
service:
  pipelines:
    metrics:
      receivers: [otlp]
      processors: [batch]
      exporters: [prometheus]
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [otlp]
  telemetry:
    logs:
      level: debug
```

- **Service** defines how telemetry data is processed through pipelines.



3. Exporters

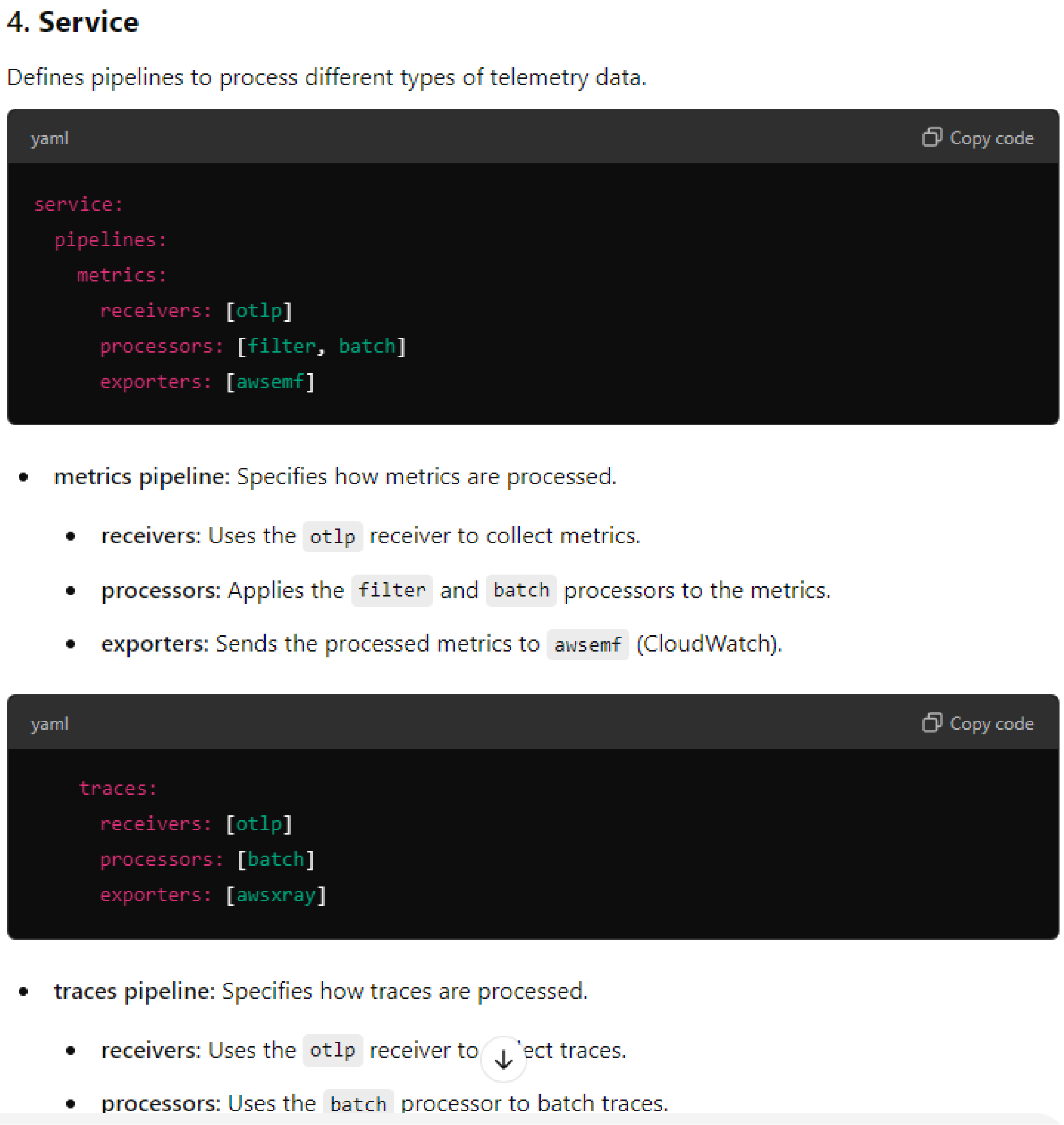
Exporters send the processed telemetry data to the desired destination.

```yaml
exporters:
  awsemf:
    region: 'us-east-1'
    log_group_name: '/metrics/otel'
    log_stream_name: 'otel-using-java'
```

- **awsemf:** AWS CloudWatch EMF (Embedded Metric Format) exporter.
  - **region:** AWS region where metrics will be sent (`us-east-1`).
  - **log_group_name:** The name of the log group in CloudWatch (`/metrics/otel`).
  - **log_stream_name:** The name of the log stream in CloudWatch (`otel-using-java`).

```yaml
  awsxray:
    region: 'us-east-1'
```

- **awsxray:** Sends traces to AWS X-Ray for analysis.
  - **region:** AWS region for X-Ray (`us-east-1`).



4. Service

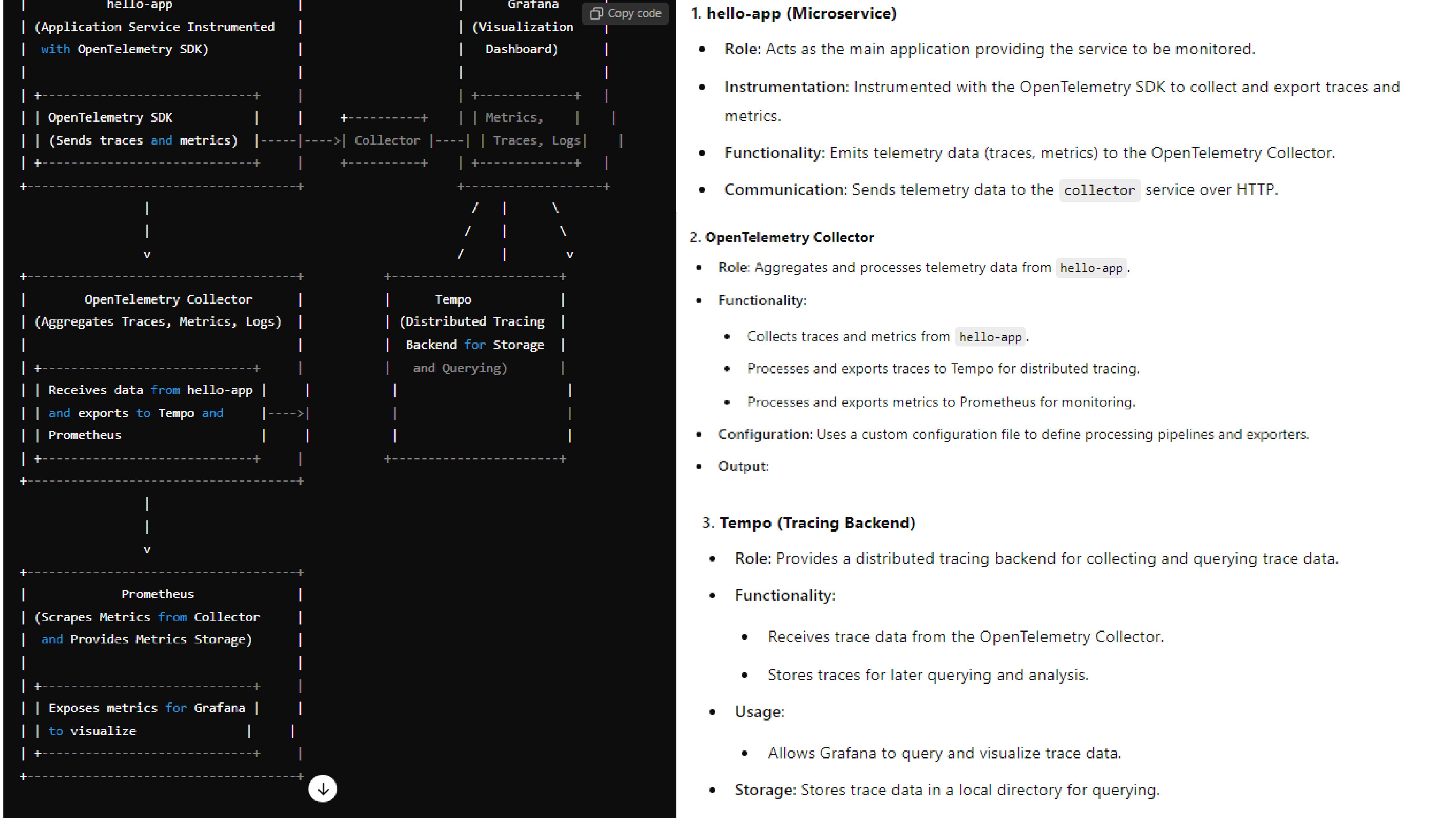
Defines pipelines to process different types of telemetry data.

```yaml
service:
  pipelines:
    metrics:
      receivers: [otlp]
      processors: [filter, batch]
      exporters: [awsemf]
```

- **metrics pipeline:** Specifies how metrics are processed.
  - **receivers:** Uses the `otlp` receiver to collect metrics.
  - **processors:** Applies the `filter` and `batch` processors to the metrics.
  - **exporters:** Sends the processed metrics to `awsemf` (CloudWatch).

```yaml
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [awsxray]
```

- **traces pipeline:** Specifies how traces are processed.
  - **receivers:** Uses the `otlp` receiver to 'ect traces.
  - **processors:** Uses the `batch` processor to batch traces.



```
|         hello-app                |                    |   Grafana         |
| (Application Service Instrumented |                    | (Visualization    |
|  with OpenTelemetry SDK)          |                    |   Dashboard)      |
|                                   |                    |                   |
| +---------------------------+     |                    | +------------+    |
| | OpenTelemetry SDK         |     |    +----------+    | | Metrics,   |    |
| | (Sends traces and metrics) |-----|---->| Collector |----| | Traces, Logs|   |
| +---------------------------+     |    +----------+    | +------------+    |
+-----------------------------------+                    +-------------------+
                  |                                        /   |      \
                  |                                       /    |       \
                  v                                      /     |        v
+-----------------------------------+          +----------------------+
|       OpenTelemetry Collector     |          |       Tempo          |
| (Aggregates Traces, Metrics, Logs) |          | (Distributed Tracing |
|                                   |          |  Backend for Storage |
| +---------------------------+     |          |   and Querying)      |
| | Receives data from hello-app |     |          |                      |
| | and exports to Tempo and     |---->|          |                      |
| | Prometheus                   |     |          |                      |
| +---------------------------+     |          +----------------------+
+-----------------------------------+
                  |
                  |
                  v
+-----------------------------------+
|             Prometheus            |
| (Scrapes Metrics from Collector   |
|  and Provides Metrics Storage)    |
|                                   |
| +---------------------------+     |
| | Exposes metrics for Grafana |     |
| | to visualize               |     |
| +---------------------------+     |
+-----------------------------------+
```

1. **hello-app (Microservice)**
   - **Role:** Acts as the main application providing the service to be monitored.
   - **Instrumentation:** Instrumented with the OpenTelemetry SDK to collect and export traces and metrics.
   - **Functionality:** Emits telemetry data (traces, metrics) to the OpenTelemetry Collector.
   - **Communication:** Sends telemetry data to the `collector` service over HTTP.

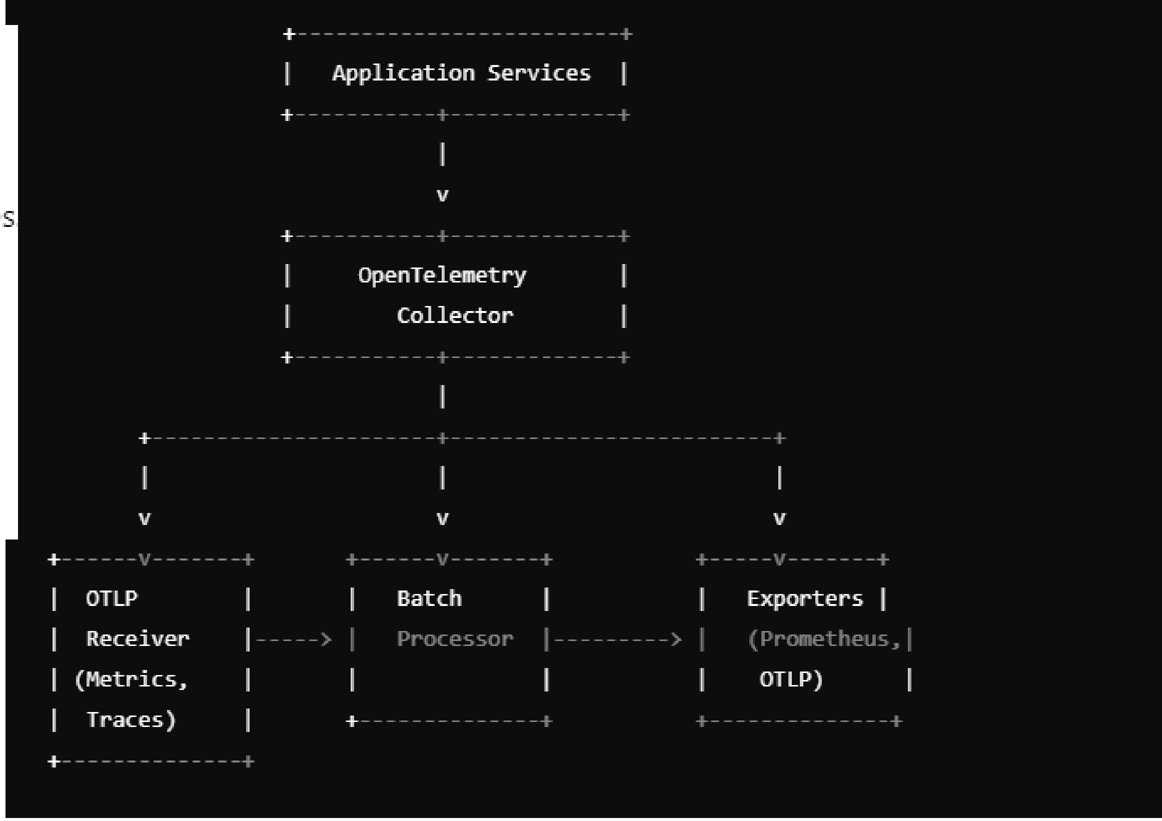
2. **OpenTelemetry Collector**
   - **Role:** Aggregates and processes telemetry data from `hello-app`.
   - **Functionality:**
     - Collects traces and metrics from `hello-app`.
     - Processes and exports traces to Tempo for distributed tracing.
     - Processes and exports metrics to Prometheus for monitoring.
   - **Configuration:** Uses a custom configuration file to define processing pipelines and exporters.
   - **Output:**

3. **Tempo (Tracing Backend)**
   - **Role:** Provides a distributed tracing backend for collecting and querying trace data.
   - **Functionality:**
     - Receives trace data from the OpenTelemetry Collector.
     - Stores traces for later querying and analysis.
   - **Usage:**
     - Allows Grafana to query and visualize trace data.
   - **Storage:** Stores trace data in a local directory for querying.

2. **Components**

The design consists of the following key components:

- **Receivers**: Entry points for collecting telemetry data (metrics and traces) from various sources.
- **Processors**: Middleware that processes, transforms, and batches telemetry data for more efficient handling.
- **Exporters**: Exit points that send telemetry data to various backends or monitoring systems.
- **Service Pipelines**: Define the flow of data from receivers through processors to exporters.

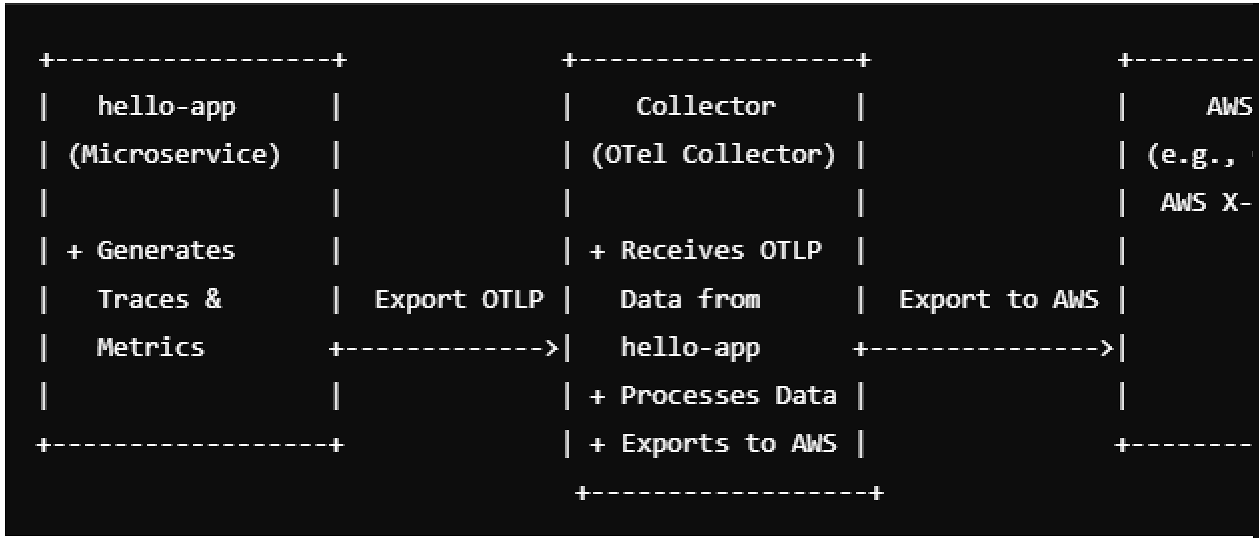


```
                 +------------------------+
                 |   Application Services  |
                 +-----------+------------+
                             |
                             v
                 +-----------+------------+
                 |     OpenTelemetry      |
                 |       Collector        |
                 +-----------+------------+
                             |
       +---------------------+------------------------+
       |                     |                        |
       v                     v                        v
+------v------+       +------v------+           +-----v-------+
|  OTLP       |       |  Batch      |           |  Exporters |
|  Receiver   |-----> |  Processor  |---------> |  (Prometheus,|
| (Metrics,   |       |             |           |   OTLP)      |
|  Traces)    |       +-------------+           +--------------+
+-------------+
```

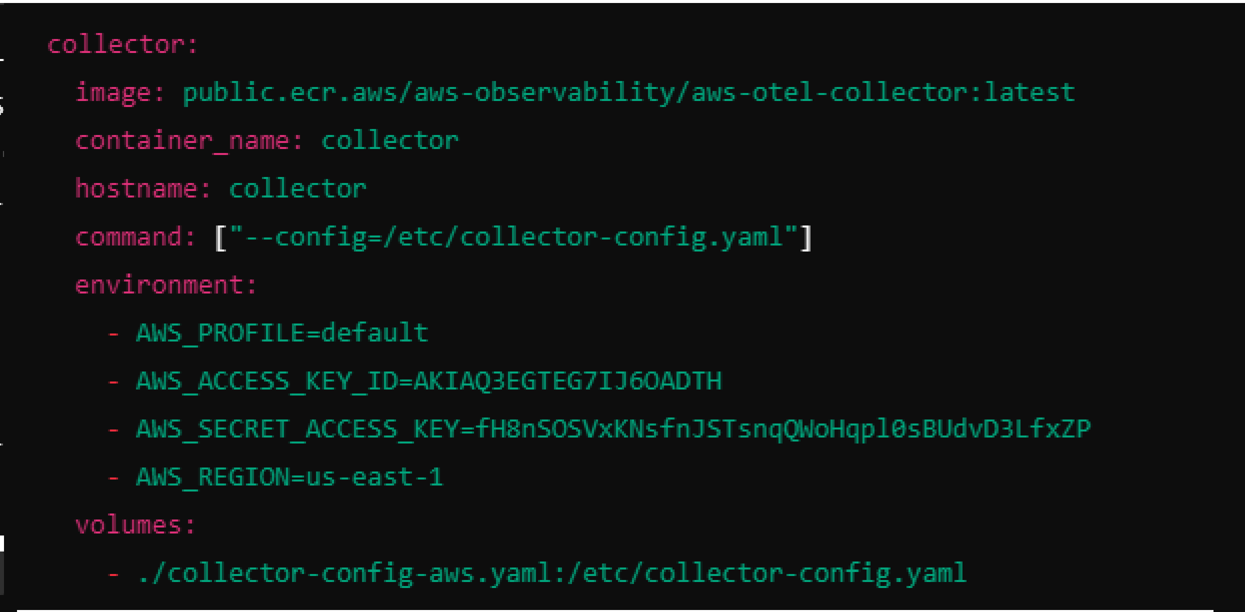
3. **Data Flow**

The data flow can be visualized as a pipeline system:

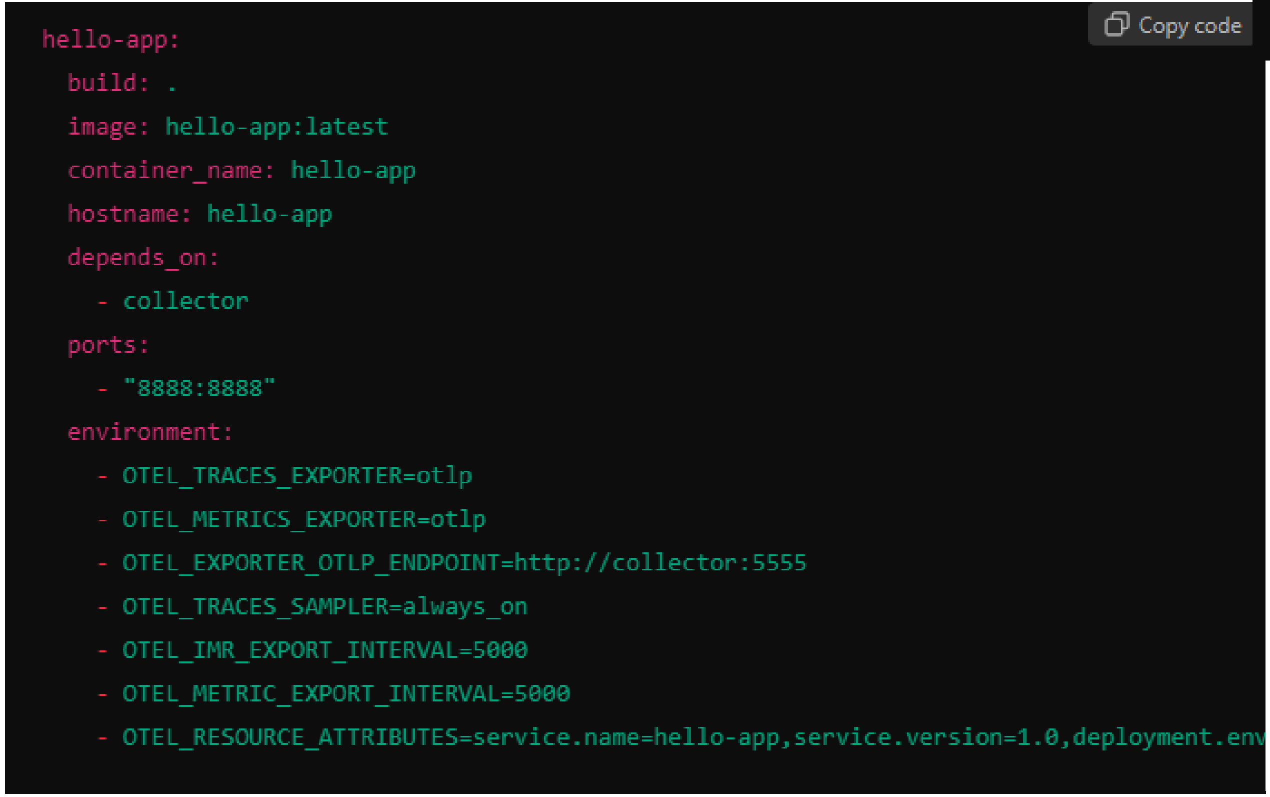
1. **Telemetry Data Collection**: Applications generate telemetry data (metrics and traces) and send it to the OpenTelemetry Collector.
2. **Reception**: The OTLP receiver listens for incoming telemetry data on a specified gRPC endpoint.
3. **Processing**: Data passes through a batch processor, which groups data into batches to optimize processing and exporting.
4. **Exporting**: Processed data is sent to configured backends:
   - Metrics are exported to a Prometheus-compatible endpoint for scraping and visualization.
   - Traces are sent to an OTLP endpoint, such as Tempo, for distributed tracing.



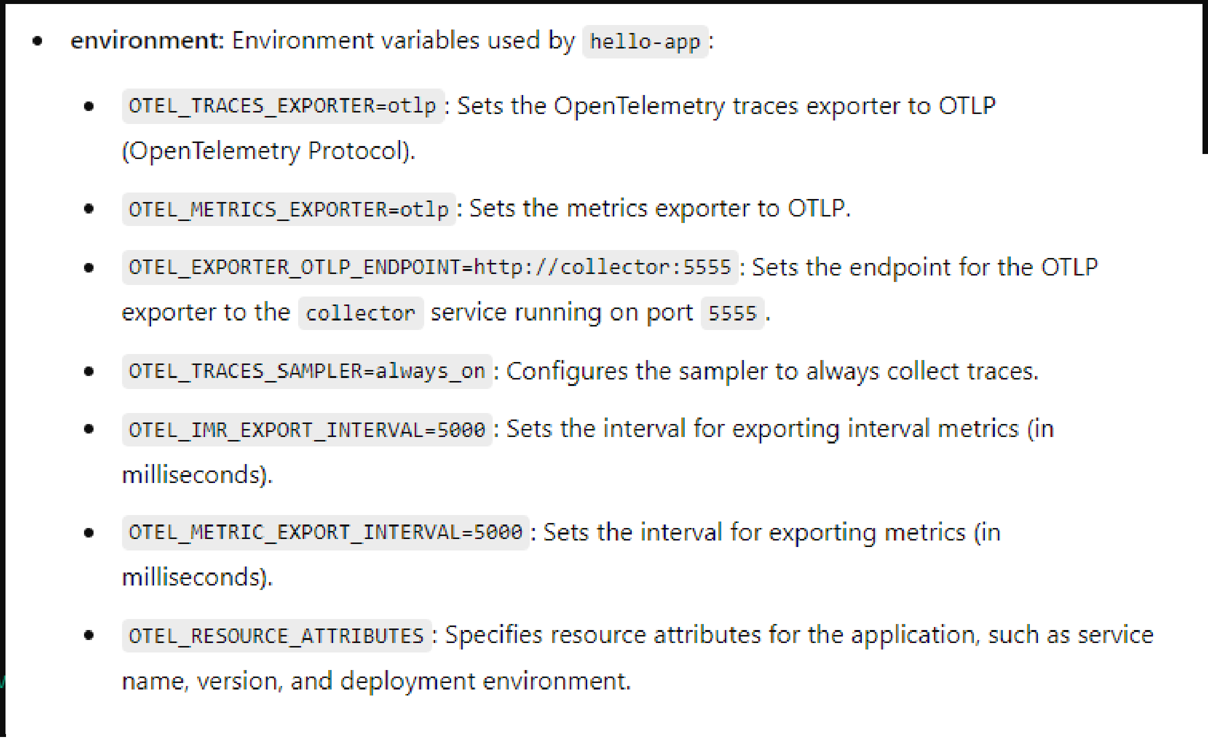
```
+-----------------+                 +-----------------+                 +--------
|   hello-app     |                 |    Collector    |                 |     AWS
| (Microservice)  |                 | (OTel Collector) |                | (e.g., 
|                 |                 |                 |                 |  AWS X-
| + Generates     |                 | + Receives OTLP |                 |
|   Traces &      | Export OTLP     |   Data from     | Export to AWS   |
|   Metrics       +-------------->|   hello-app     +--------------->|
|                 |                 | + Processes Data |                |
+-----------------+                 | + Exports to AWS |                +--------
                                     +-----------------+
```



```
collector:
  image: public.ecr.aws/aws-observability/aws-otel-collector:latest
  container_name: collector
  hostname: collector
  command: ["--config=/etc/collector-config.yaml"]
  environment:
    - AWS_PROFILE=default
    - AWS_ACCESS_KEY_ID=AKIAQ3EGTEG7IJ6OADTH
    - AWS_SECRET_ACCESS_KEY=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
    - AWS_REGION=us-east-1
  volumes:
    - ./collector-config-aws.yaml:/etc/collector-config.yaml
```



```
hello-app:
  build: .
  image: hello-app:latest
  container_name: hello-app
  hostname: hello-app
  depends_on:
    - collector
  ports:
    - "8888:8888"
  environment:
    - OTEL_TRACES_EXPORTER=otlp
    - OTEL_METRICS_EXPORTER=otlp
    - OTEL_EXPORTER_OTLP_ENDPOINT=http://collector:5555
    - OTEL_TRACES_SAMPLER=always_on
    - OTEL_IMR_EXPORT_INTERVAL=5000
    - OTEL_METRIC_EXPORT_INTERVAL=5000
    - OTEL_RESOURCE_ATTRIBUTES=service.name=hello-app,service.version=1.0,deployment.env
```



- **environment:** Environment variables used by `hello-app`:
  - `OTEL_TRACES_EXPORTER=otlp`: Sets the OpenTelemetry traces exporter to OTLP (OpenTelemetry Protocol).
  - `OTEL_METRICS_EXPORTER=otlp`: Sets the metrics exporter to OTLP.
  - `OTEL_EXPORTER_OTLP_ENDPOINT=http://collector:5555`: Sets the endpoint for the OTLP exporter to the `collector` service running on port `5555`.
  - `OTEL_TRACES_SAMPLER=always_on`: Configures the sampler to always collect traces.
  - `OTEL_IMR_EXPORT_INTERVAL=5000`: Sets the interval for exporting interval metrics (in milliseconds).
  - `OTEL_METRIC_EXPORT_INTERVAL=5000`: Sets the interval for exporting metrics (in milliseconds).
  - `OTEL_RESOURCE_ATTRIBUTES`: Specifies resource attributes for the application, such as service name, version, and deployment environment.