WAM IMM OBSERVABILITY

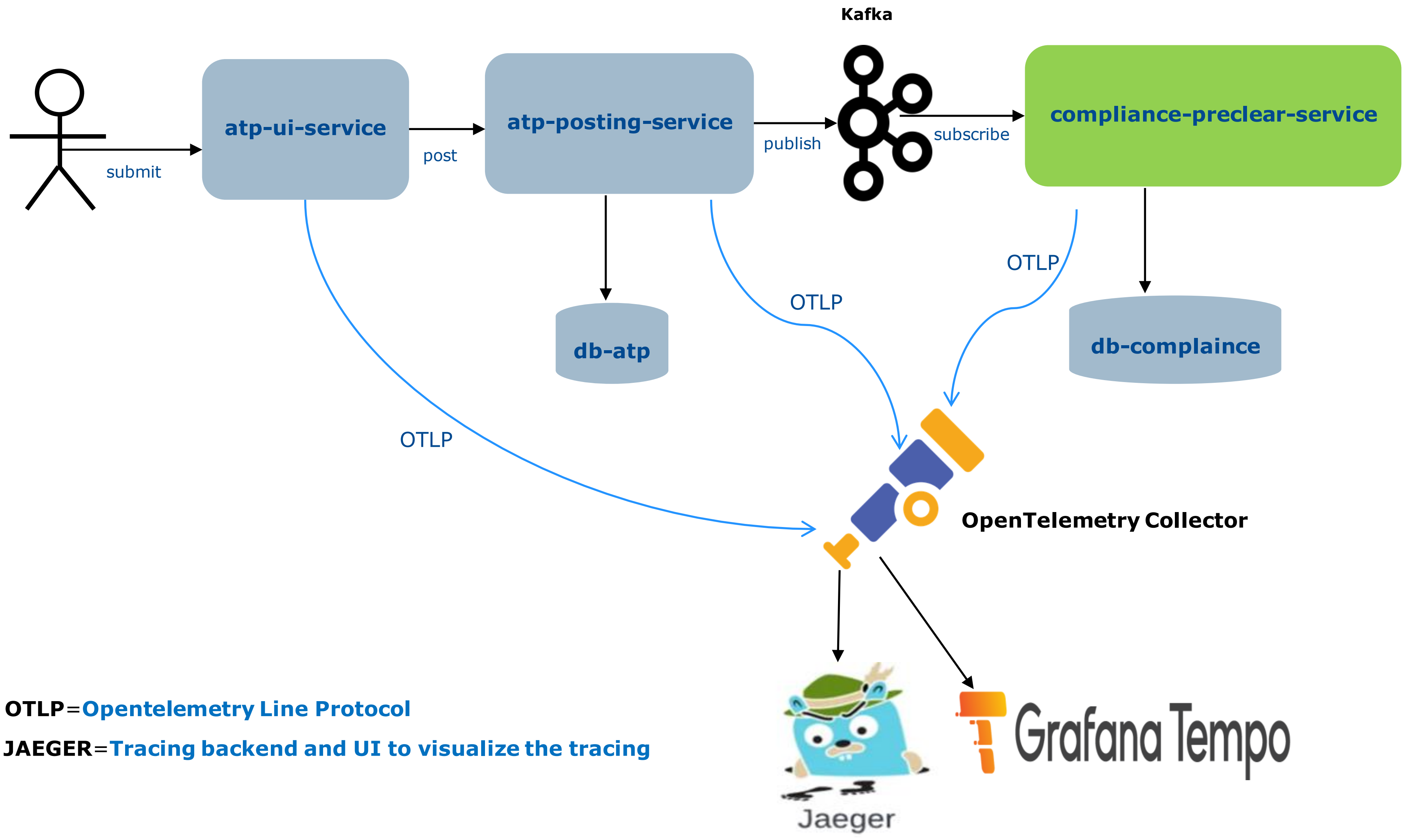
Jan 2023

Objective:

End-to-End request tracing across all micro services to pre-clear trades

WESTERN ASSET

System Architecture



OTLP=**Opentelemetry Line Protocol**

JAEGER=**Tracing backend and UI to visualize the tracing**

Western Asset

System Design

The system in this POC has 3 Spring Boot Microservices:

1. atp-ui-service: This service sits between the UI and the backend. It is called by a UI web app, which in turn, calls the back end **atp-posting-service** via REST API calls.

2. **atp-posting-service**: Provides CRUD service for **Trade Transaction**. In addition to persisting data to its own **database postgres-atp** upon CRUD operations, it also publishes **events/messages to Kafka** when creating, updating, or deleting a **Trade Transaction** record.

3. **compliance-preclear-service**: Listens on the Kafka topic, consumes **Trade Transaction** created/updated/deleted events and persist to its own database **postgres-compliance**

These microservices communicate as:

1. **REST API** (**atp-ui-service** **and** **atp-posting-service**)

1. **Event-driven pub/sub through Kafka** (**atp-posting-service** **and** **compliance-preclear-service**)

Distributed Tracing using OpenTelemetry

OpenTelemetry allows to trace a request passing through a number of independent micro services without being bound to a vendor-locking implementation.

OTEL libraries are available for various languages, frameworks and libraries with different level of maturity

OpenTelemetry + Spring Cloud Sleuth

OpenTelemetry provides an agent (JAR) to attach with Java applications for generating traces.

But, in this Demo POC We will combine OpenTelemetry with Spring Cloud Sleuth to handle auto instrumentation of the code, generates and transmits the tracing data.

Spring Cloud Sleuth abstraction delegates the instrumentation to OpenTelemetry and allows us to get up and running quickly without attaching the **opentelemetry-javaagent.jar** to your application as

```
java -javaagent:path/to/opentelemetry-javaagent.jar  -jar myapp.jar
```

Maven Dependencies for Auto Instrumentation

Our goal is to auto instrument the application code to generate the **TraceID**, **SpanID** and export the **tracing data** to **OpenTelemetry Collector**

```
<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId> spring-cloud-dependencies </artifactId>
         <version>${spring-cloud.version}</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId> spring-cloud-sleuth-otel-dependencies </artifactId>
          <version>${spring-cloud-sleuth-otel.version}</version>
         <scope>import</scope>
         <type>pom</type>
      </dependency>
   </dependencies>
</dependencyManagement>
```

1. **spring-cloud-dependencies**: To cloud enable / ready Spring Boot Application

2. **spring-cloud-sleuth-otel-dependencies**: This will integrate Spring Cloud Sleuth with OpenTelemetry

Maven Dependencies for Auto Instrumentation

```
<dependencies>
   <dependency>
      <groupId>org.springframework.cloud</groupId>

      <artifactId> spring-cloud-starter-sleuth </artifactId>

      <exclusions>
         <exclusion>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-sleuth-brave</artifactId>
         </exclusion>
      </exclusions>
   </dependency>
   <dependency>
      <groupId>org.springframework.cloud</groupId>

      <artifactId> spring-cloud-sleuth-otel-autoconfigure </artifactId>

</dependency>
   <dependency>
      <groupId>io.opentelemetry</groupId>

      <artifactId> opentelemetry-exporter-otlp-trace </artifactId>

   </dependency>
</dependencies>
```

3. **spring-cloud-starter-sleuth**: By default Sleuth integrates with the OpenZipkin Brave tracer available in the spring-cloud-sleuth-brave module. Since we are going to use OpenTelemetry tracer in this POC, we will exclude **spring-cloud-sleuth-brave** from the spring-cloud-starter-sleuth dependency and add **spring-cloud-sleuth-otel-autoconfigure** dependency. This will replace the default tracing implementation based on Brave with the implementation based on **OpenTelemetry**

4. **opentelemetry-exporter-otlp-trace:**: This will send the tracing data to **OpenTelemetry Collector**

Microservice Configuration

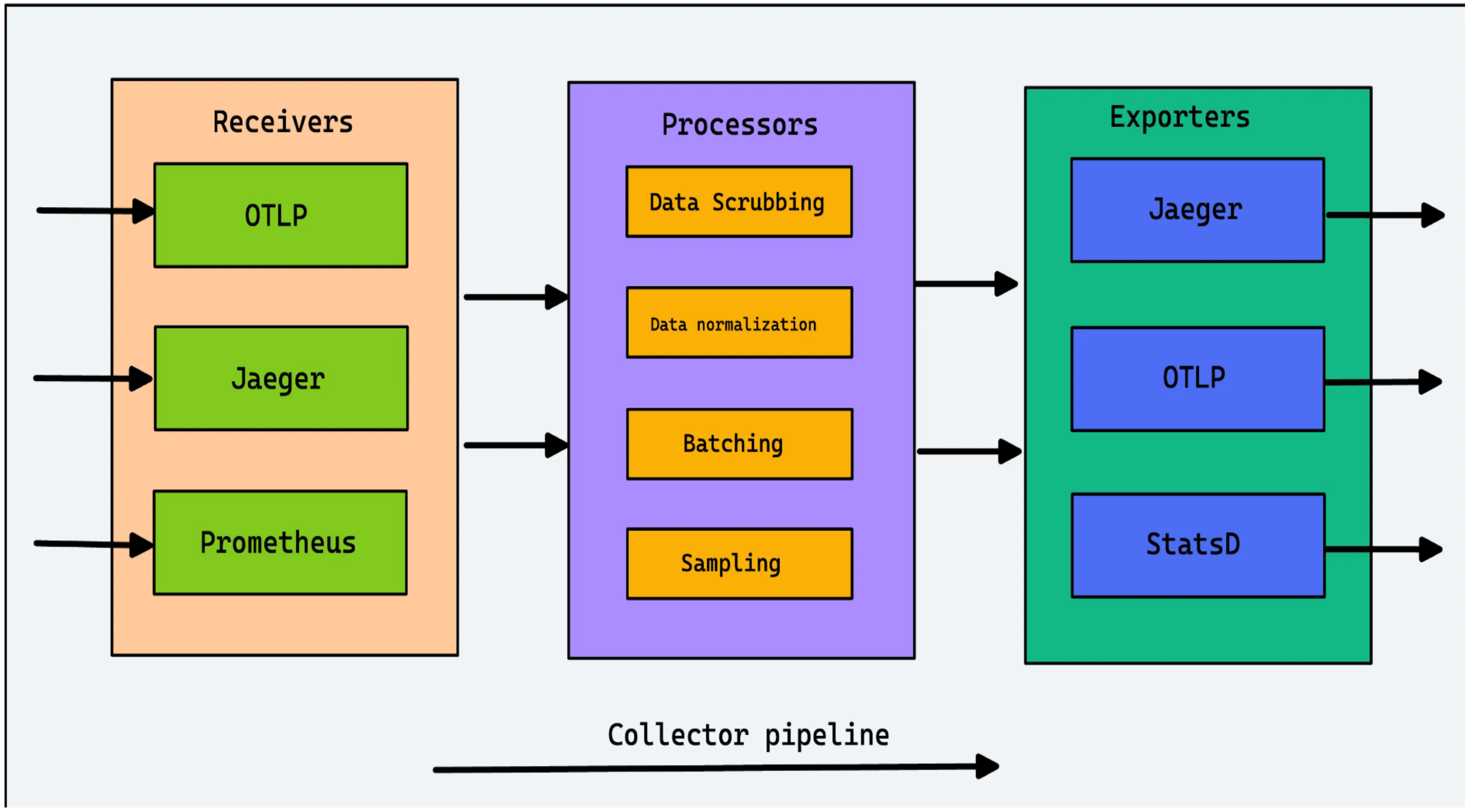
OpenTelemetry Collector endpoint: The **application.yaml** of each microservice should have **spring.sleuth.otel.exporter.otlp.endpoint** to configure the OpenTelemetry Collector endpoint.

```yaml
spring:
  application:
    name: atp-posting-service
  sleuth:
    otel:
      config:
        trace-id-ratio-based: 1.0
      exporter:
        otlp:
          endpoint: http://otel-collector:4317
```

WESTERN ASSET

OpenTelemetry Collector Configuration

OpenTelemetry Collector has 3 components **Receiver, Processor and Exporter** to receive, process and export telemetry data in **Vendor-agnostic way**



OpenTelemetry Collector Configuration

One configuration **otel-config.yaml** file is required to define the behaviors of the OpenTelemetry **receivers, processors, and exporters**. In this POC our receivers to listen on gRPC and HTTP, processors using batch and exporters as jaeger and logging.

```yaml
receivers:
  otlp:
    protocols:
      grpc:
      http:

processors:
  batch:

exporters:
  logging:
    logLevel: debug
  jaeger:
    endpoint: jaeger-all-in-one:14250
    tls:
      insecure: true

service:
  pipelines:
    traces:
      receivers: [ otlp ]
      processors: [ batch ]
      exporters: [ logging, jaeger ]
```

WESTERN ASSET

Infrastructure Setup

We will use docker compose to bring up all **3 microservices, their own databases, kafka and jaeger-all-in-one**

1. **atp-ui-service**
2. **atp-posting-service**
3. **compliance-preclear-service**
4. **postgres-atp: database for atp-posting-service**
5. **postgres-compliance: database for compliance-preclear-service**
6. **jaeger-all-in-one: single image that runs all Jaeger backend components and UI**
7. **otel-collector: the engine of OpenTelemetry tracing, it receives, processes, and exports the tracing data to the backend**
8. **zookeeper: track the status of nodes in the Kafka cluster and maintain a list of Kafka topics and messages**
9. **kafka: pub/sub event streaming platform**

Build and Run

1. Project root folder: ./mvnw clean package

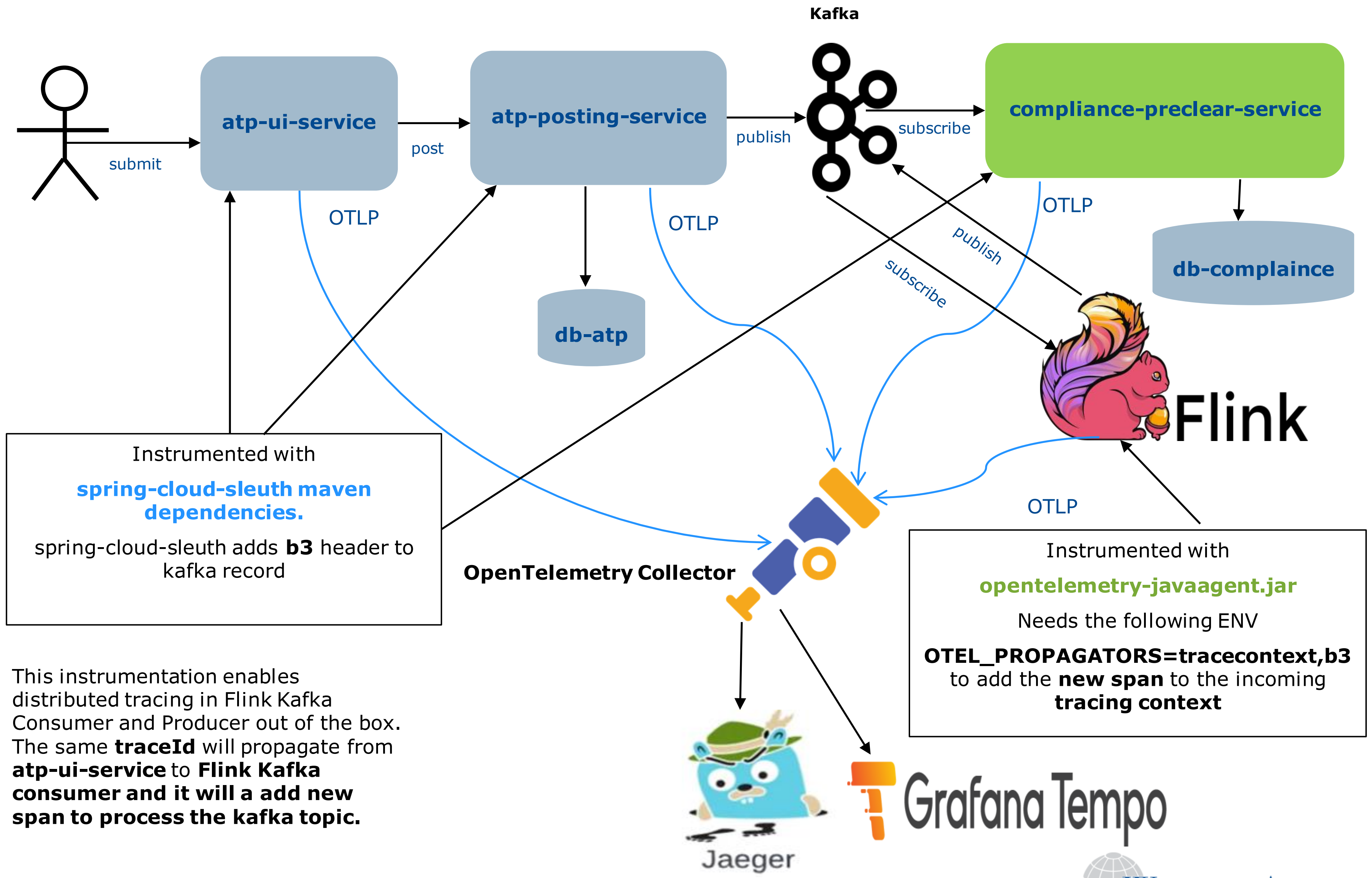
2. Go to each sub folder of the microservice and execute:
docker build –t <service-name>:0.0.1-SNAPSHOT .

3. docker-compose up

4. Entry point for atp-ui-service : http://localhost:8080/swagger-ui/index.html

5. Jaeger UI: http://localhost:16686/

Next Step to Add Flink Job in System Architecture



Instrumented with

spring-cloud-sleuth maven dependencies.

spring-cloud-sleuth adds **b3** header to kafka record

Instrumented with

opentelemetry-javaagent.jar

Needs the following ENV

OTEL_PROPAGATORS=tracecontext,b3 to add the **new span** to the incoming **tracing context**

This instrumentation enables distributed tracing in Flink Kafka Consumer and Producer out of the box. The same **traceId** will propagate from **atp-ui-service** to **Flink Kafka consumer and it will a add new span to process the kafka topic.**

Instrumentation - spring-cloud-sleuth + opentelemetry-javaagent

PROS: **opentelementry-javaagent** enables distributed tracing in **Flink Kafka Consumer and Producer out of the box**. The same **traceId** will propagate from **atp-ui-service** to **Flink Kafka consumer** and it will **a add new** span to process the kafka topic.

CONS: But the Flink Kafka Producer will create **a new traceId and spanId**, because it runs in a separate thread.

OpenTelemetry uses metadata to collect traces across micro services and method call within a single service.
These metadata passed in

1. HTTP Headers when the micro services communicate over REST and HTTP.
2. ThreadLocal method calls within a single service.
3. Message Headers when the micro services communicate asynchronously using some messaging system Kafka, RabbitMQ, Active MQ and others.

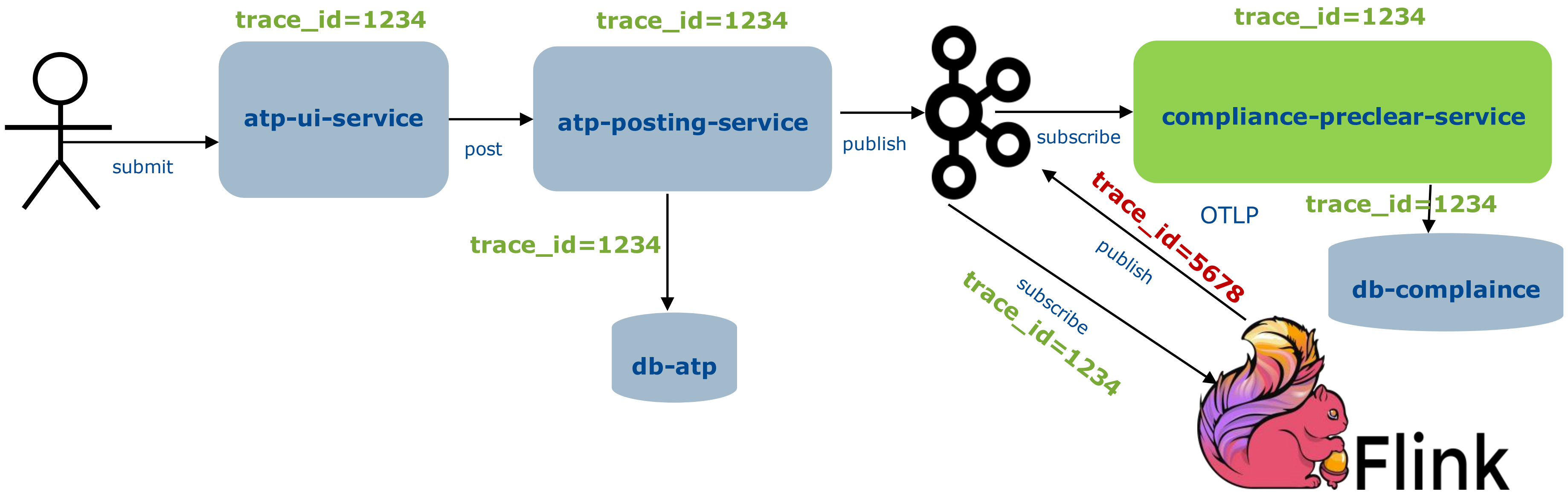
spring-clould-seluth extends the capability of OpenTelemetry library by propagating the same tracing context across different threads within the same jvm / process.

Challenges with Flink and AWS KDA

Components (**source/consumer, operator and sink/producer**) of Flink Application works in different threads. OpenTelemetry library is not well integrated with Fink. Flink itself does not extends the capability of OpenTelemetry library like spring-cloud-seulth to propagate the tracing context from **source/consumer -> operator -> sink/producer**

Strategy for Flink Application:

W3c tracing context can be propagated from non-flink producer applications(Spring Boot, Traditional Web App, Stand Alone and Kafka Source Connectors) to up to Flink Kafka Source/Consumer. **No tracing context is established across the operators. A new tracing context is established within Flink Sink/Producer Component.**



Flink Operators communicate over DataStream API. Producer Applications instrumented with OpenTelemetry will add tracing metadata to the Header of Kafka Record. To propagate distributed tracing context in Flink App (**source/consumer -> operator -> sink/producer) we have to take hybrid or manual instrumentation approach.**

WESTERN ASSET

Propagate Tracing Context using Enriched Data Model

In this approach we will enrich the business data model(**TradeVO**) by adding tracing metadata from Kafka Record Header as shown below

```
public class TradeVO {
   private Long tradeId;
   private String ticketType;
   private String assetId;
   private String pfNumber;
   private BigDecimal parAmount;
   private String status;
}
```

```
public class EnrichedTradeVO {
   public TradeVO tradeVO;
   public Metadata metadata; // added by Kafka Broker
   public Headers headers; // added by OpenTelemetry Library
}
```

We will manually
Create a new span in each Flink Component
Add it to the incoming tracing context
Pass the new span_id in Header.parentid field

How to instrument Flink App in AWS KDA

AWS has confirmed as of now **there is no configuration parameter in** KDA which will allow us to pass **opentelemetry-javaagent.jar** as JVM argument as

- FLINK_ENV_JAVA_OPTS=-javaagent:/tmp/opentelemetry-javaagent.jar

So we have to do manual instrumentation. The first step is to get a handle to a **Global instance of the OpenTelemetry interface**.

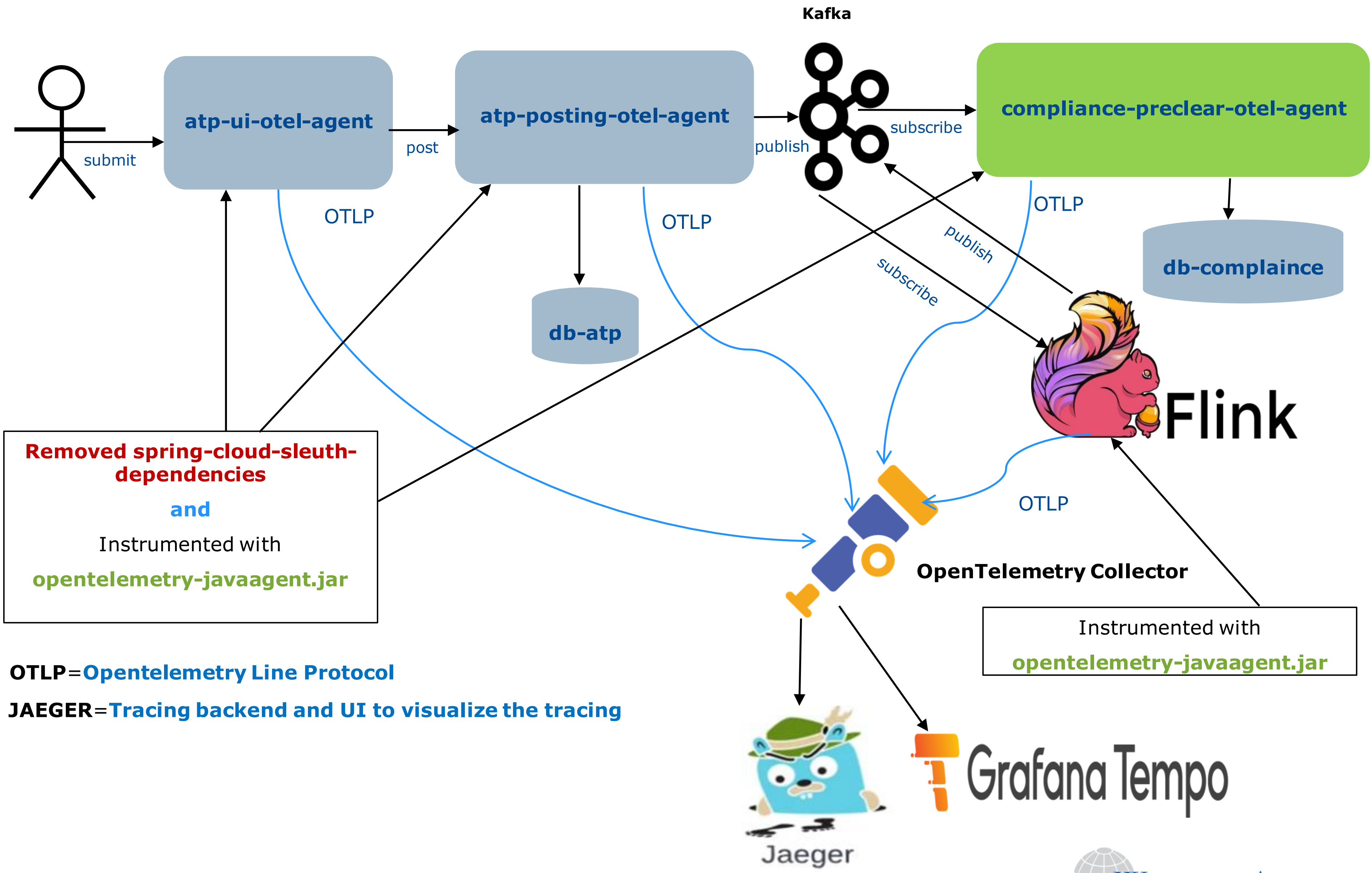
```
Resource resource = Resource.getDefault()
  .merge(Resource.create(Attributes.of(ResourceAttributes.SERVICE_NAME, "logical-service-name")));

SdkTracerProvider sdkTracerProvider = SdkTracerProvider.builder()
  .addSpanProcessor(BatchSpanProcessor.builder(OtlpGrpcSpanExporter.builder().build()).build())
  .setResource(resource)
  .build();

OpenTelemetry openTelemetry = OpenTelemetrySdk.builder()
  .setTracerProvider(sdkTracerProvider)
  .setPropagators(ContextPropagators.create(W3CTraceContextPropagator.getInstance()))
  .buildAndRegisterGlobal();
```

More info on Manual Instrumentation: https://opentelemetry.io/docs/instrumentation/java/manual/

Next Step to Add Flink Job in System Architecture



Removed spring-cloud-sleuth-dependencies

and

Instrumented with

opentelemetry-javaagent.jar

Instrumented with

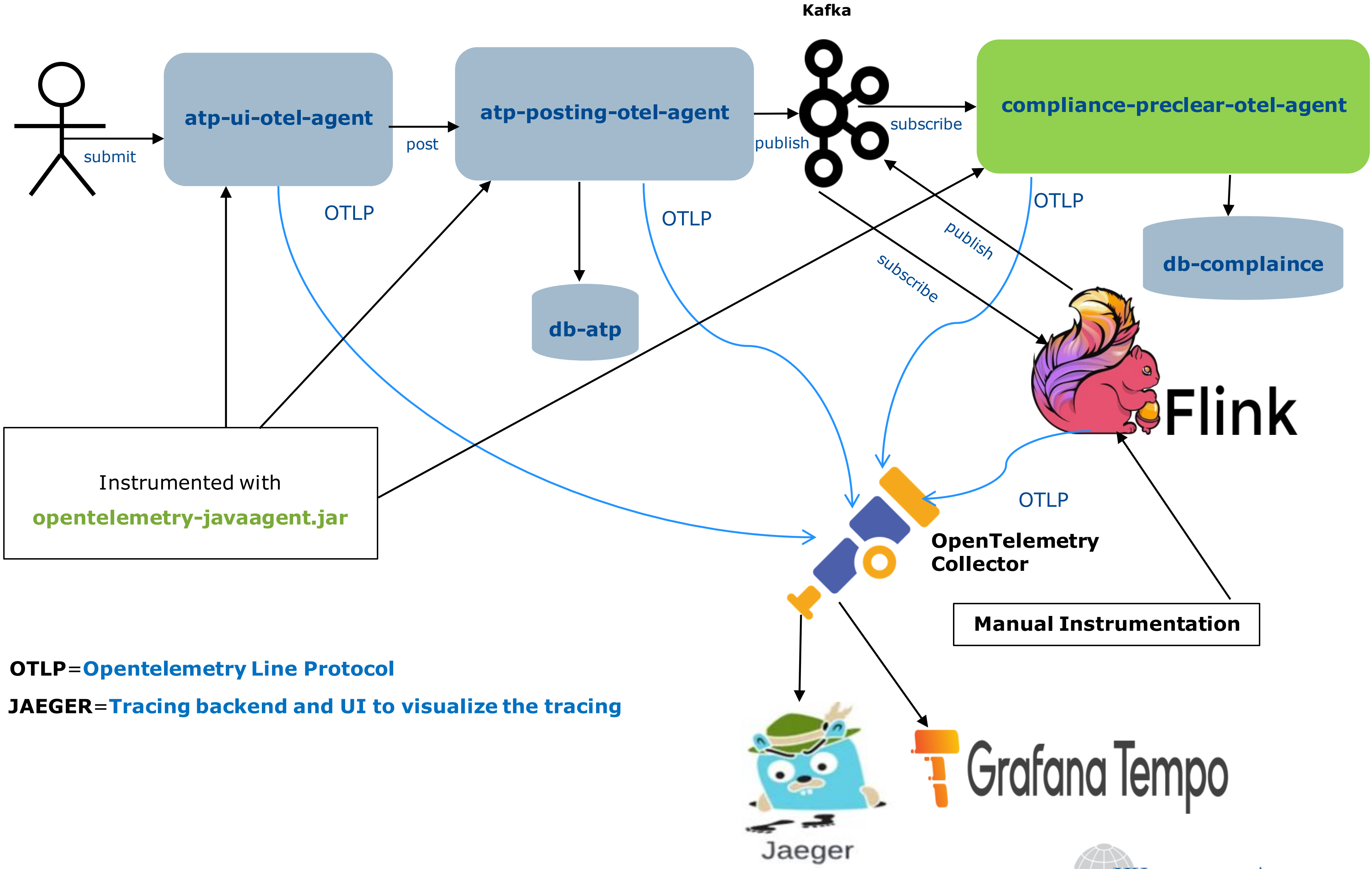
opentelemetry-javaagent.jar

OTLP=**Opentelemetry Line Protocol**

JAEGER=**Tracing backend and UI to visualize the tracing**

WESTERN ASSET

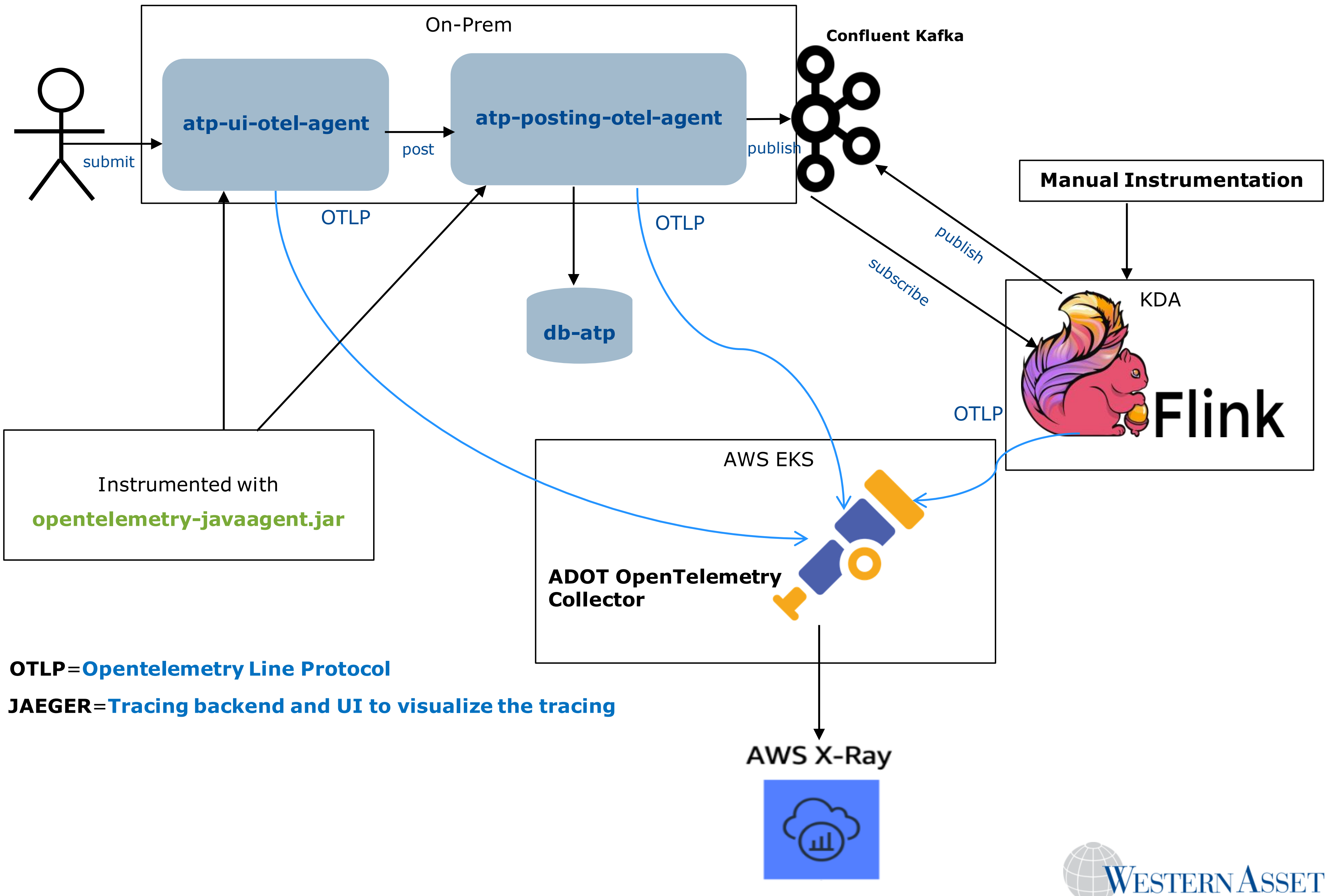
Next Step to Add Flink Job in System Architecture



OTLP=**Opentelemetry Line Protocol**

JAEGER=**Tracing backend and UI to visualize the tracing**

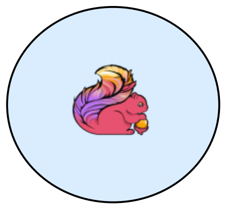
Run the demo app in AWS Infrastructure

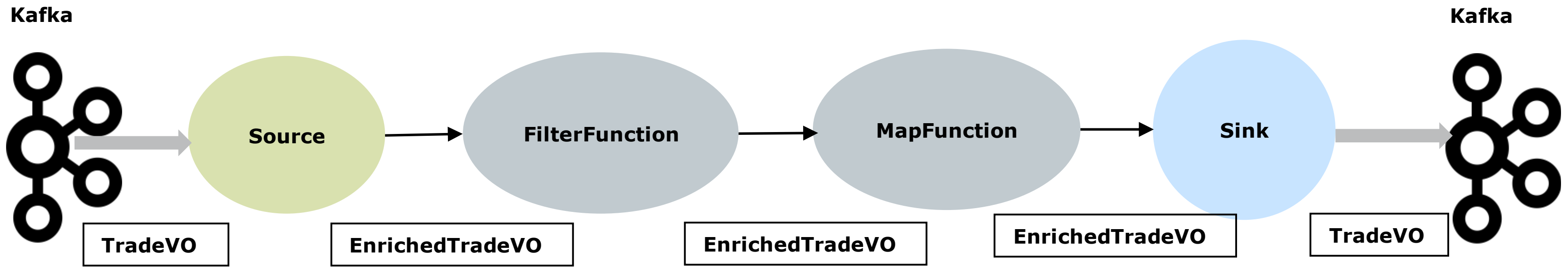


OTLP=**Opentelemetry Line Protocol**

JAEGER=**Tracing backend and UI to visualize the tracing**

TradePipeLineJob





Next Step to manually instrument Flink App

"headers":["**traceparent**","00-adbb381384d6a52cf03517e1fac1784a-aba6325bfca20584-01","**b3**","adbb381384d6a52cf03517e1fac1784a-aba6325bfca20584-1"]