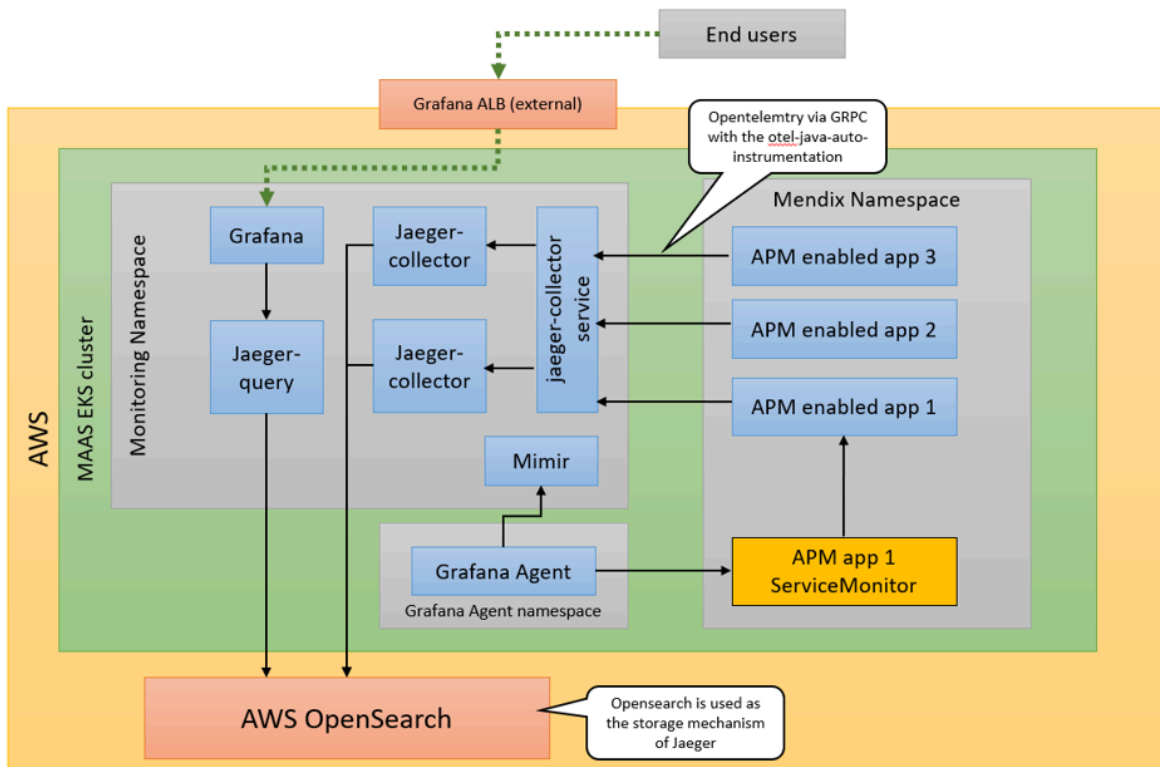


## Application Performance Monitoring (APM)

Application Performance Monitoring, or APM, is collecting data from the backend system to get performance data such as CPU, memory as well as trace data and logs (although we are not using APM for logs) amongst other things. Typically you add APM by attaching an instrumentation agent to the running process of the backend system. In our case, this means attaching a Java Agent to the JVM of the Mendix runtime. Dynatrace was able to do this generically by attaching itself to all running JVMs. In the Grafana setup, we are adding the OpenTelemetry Java Instrumentation and using the auto instrumentation capabilities of OpenTelemetry. OpenTelemetry is supporting a bunch of Java frameworks, but the Mendix Runtime is not a part of it. This means that any Mendix specific actions are not captured, but the JDBC driver is, so it is able to capture performance and query times. OpenTelemetry is also exporting JVM metrics as Prometheus metrics.

We are sending OpenTelemetry data to Jaeger. We are using Jaeger and not the Grafana Agent OpenTelemetry collector because the Grafana Agent does not connect trace data correctly. What that means is that every span becomes a unique trace id which renders the tracing feature useless. Jaeger does not have this problem and therefore connects spans with traces correctly. This allows you to trace the SQL queries and their times accordingly. Mendix application developers can add their own OpenTelemetry code in their codebases to add their own spans. This will then be added as part of the trace data which is visible in Grafana.



In this diagram an “APM enabled app” can be read as a Mendix application that has APM enabled, but this framework is generically applicable for other apps too.

## OpenTelemetry overview


OpenTelemetry (also known as otel or otlp) is a CNCF project and has seen broad adaptation amongst a wide variety of commercial and open-source offerings. This space used to be fragmented with vendors offering their own tracing standard, but these days OpenTelemetry is emerging to become a standard in APM / tracing. There are two parts to it; an agent and a collector. The latter in our case is done by Jaeger, but OpenTelemetry provides its own collectors as well. The agent is either attached to an existing application or embedded into the codebase directly, making it part of the application itself. We are using the attached mode so we don't have to manually instrument the application to get the out-of-the-box experience of OpenTelemetry. Regardless of embedding it or not, you always add your own tracing and metrics using the OpenTelemetry framework (of course, adding it to the codebase implies OpenTelemetry becoming a runtime dependency). As previously mentioned the agent in our case runs as a Java agent using the auto instrumentation capabilities. As Mendix is not a supported framework out of the box we will not see any Mendix related data and those must be manually instrumented.

OpenTelemetry supports traces, logs and metrics. Tracing data is being exposed as the Jaeger format, and Jaeger runs as a cluster service inside of the Mendix EKS cluster and serves as an endpoint for all APM enabled applications. The logging support is not used at this time of writing and does not have supporting infrastructure in place. This of course refers to the logging capabilities of OpenTelemetry itself and not to generic container-based logging which is captured by Loki. Metrics are exposed as a Prometheus endpoint and the Mendix Helm chart deploys a ServiceMonitor per application, which the Grafana Agent uses to determine scrape targets (this is part of the standard Prometheus Operator CRDs, which is what the Grafana Agent also uses to dynamically manage its scrape targets).

The Java agent options we use for Mendix applications are as follows (note that the OpenTelemetry jar is built into the container). This sets the format, or propagator, to Jaeger format, starts the Prometheus metrics exporter and sets the endpoint. The last line, the service name, is important to identify which traces belong to which application. As this is a snippet from the Helm chart, we use a Helm function to identify the Mendix application name.

```
1 - -javaagent:/opt/mendix/otel-java-agent.jar
2 - -Dotel.propagators=jaeger
3 - -Dotel.metrics.exporter=prometheus
4 - -Dotel.exporter.prometheus.port=9464
5 - -Dotel.exporter.otlp.endpoint=http://jaeger-collector.monitoring.svc.cluster.local:4317
6 - -Dotel.exporter.otlp.protocol=grpc
7 - -Dotel.service.name={{ include "mendix-app.fullname" . }}
```

Of course, the Mendix deployment creates a portmapping to expose port 9464 on a cluster level which is used by the ServiceMonitor to generate a valid scrape configuration for the Grafana Agent for this service. Finally, please note that the primary metrics that OpenTelemetry exposes are JVM based metrics.

 Note that by default the OpenTelemetry Agent exposes HTTP route metrics, which has a high cardinality by nature (although OpenTelemetry has a limit on the number of metrics it exposes). This may cause Mimir performance degradation over time, especially with a high number of APM enabled applications.

## Jaeger overview

Jaeger is a distributed tracing tool which is also a CNCF tool. This enables the generation of trace ID and spans and can capture the time it takes for certain parts of the codebase to execute. This is an inherently different kind of data than metric data. As mentioned earlier, Mendix is not a supported framework by OpenTelemetry but we can at least capture JDBC calls and the type of queries that certain HTTP requests are doing. Further instrumentation

(which the Mendix app developers will have to add) will make it possible to get more details. Jaeger has a query-ui pod, which serves as both a query engine as well as the Jaeger UI. The collectors are the ones that actually receive the data and write it into its storage layer.


With Jaeger, there are two types of storage mechanisms supported; Cassandra and Elasticsearch. We've settled on the latter as there is a little more knowledge of that system in the company and with HCL. We use the AWS managed version of it to abstract away the management of it. This is all deployed via CDK. Jaeger itself is deployed via its standard helm chart.

Note that the Jaeger UI does not show anything more than Grafana can display at this time of writing, therefore it adds no additional value to use it.

## Infrastructure-as-code

The code for APM is fairly straightforward but comes in two parts; one is the CDK code needed for the AWS Managed OpenSearch (their fork of Elasticsearch) and registering applications to enable APM. Both are found in the *monitoring* repository. Inside this repository there is a `configuration/apm.yaml` and this is the place where you enable APM apps. The only thing this will do is writing a parameter in the SSM Parameter Store, namely `/dsm/apps/mendix/<app_name>/config/apm` and set its value to `true`. The Helm chart of the respond to this by enabling the OpenTelemetry Java Agent as mentioned in the overview section above. As this a change to the app configuration, a redeployment is required.

OpenSearch runs as a 3 node cluster with no master nodes, nor any ultrawarm nodes. This is done to reduce costs. As we don't know at this time of writing how much load the APM enabled apps will produce on the OpenSearch backend, it is recommended to monitor the performance of it as more apps get onboarded onto it.

 Note that we are not using the Jaeger Operator but simply the standalone Helm chart of Jaeger. The Operator did not add any functionality to offset yet another layer of abstraction.

### Enabling a new APM app

To summarize what is mentioned above, this is how to enable a new APM app:

1. Add the name of app to `configuration/apm.yaml` in the *monitoring* repository.

2. Run the pipeline (select the feature/apm parameter)
3. Redeploy the application by running the release pipeline of the application you neabled.

Note that disabling APM implies the same procedure, except removing the app in step1 instead of adding it.

## Further reading

- [Observability Primer | OpenTelemetry](#)
- [Deployment — Jaeger documentation \(jaegertracing.io\)](#)