Operational Observability

Operational observability is focused on how the system performs. In general, it consists of two pillars:

- *Measuring and Monitoring.* Measure how the system performs according to the system consumer's expectations and monitor to detect and predict failures and violations of those expectations.
- *Troubleshooting.* Find and eliminate a cause of the detected system failure or unexpected behavior.

Measuring and Monitoring

Measuring and monitoring lets us observe and predict the system behavior, compare it to the defined expectations and ultimately detect existing or potential issues.

We measure with ***Metrics*** and monitor with dashboards and alerts.

We are building the monitoring part of the observability system focused on SLOs/SLIs. It draws out what is most important for the consumer of the system. Formal system expectations from the perspective of different personas will let us identify what key performance indicators we want to measure and what metrics we want to collect. This analysis will help us design targeting dashboards and alerts for different observers at various levels.

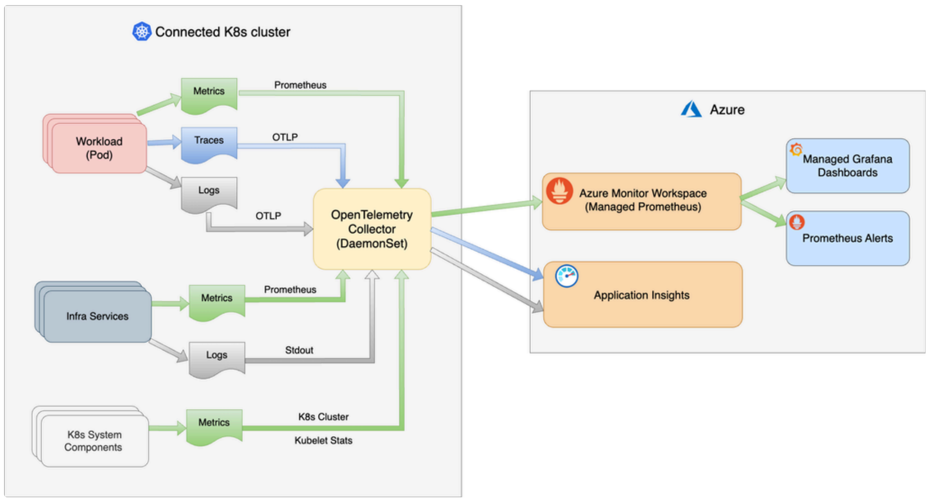
Troubleshooting

When the monitoring system signals a problem or a tendency towards a problem, the troubleshooting lets us identify and locate the cause of the problem.

There are two observability instruments serving the troubleshooting purposes: ***Traces*** and ***Logs***.

Traces and logs are most valuable when they are correlated. It gives the ability to analyze what was happening at every point in the flow when a specific piece of data was processed.

Data Collection



The data collection solution is based on OpenTelemetry. Workloads expose metrics via Prometheus endpoints and send traces and correlated logs to an OTLP endpoint. An OpenTelemetry Collector on the cluster collects the metrics, logs and traces and sends the data to Azure Monitor. It is configured to collect both application metrics and standard Kubernetes metrics such as CPU, memory, disc, nodes statistics, etc. (see the full list of standard Kubernetes cluster and kubelet system metrics). This will allow us to send detailed information about the performance and health of the solution, including resource usage, network activity, and more. Metrics are landed in Azure Monitor Workspace (Managed Prometheus) to be consumed by Grafana dashboards and Prometheus alerts. Traces and correlated logs are forwarded to Application Insights.

Provisioning of observability resources

Provisioning of the cloud services, such as Azure Monitor Workspace, Managed Grafana and Application Insights is performed with a bicep IaC pipeline.

Deployment and configuring of the OpenTelemetry Collector on the clusters is accomplished via GitOps.

OpenTelemetry Collector deployment plan

OpenTelemetry Collector is deployed on each cluster as a daemon set, so there is a collector pod on each cluster node. There is a separate daemon set instance for each environment (stage, prod).

The amount of resources used by each collector pod depends on how many metrics we are collecting, workloads logging level and traces ratio. At this point, basing on the previous experience, we can estimate the resource usage as the following:

```yaml
resources:
  requests:
    cpu: 100m
    memory: 512Mi
  limits:
    cpu: 500m
    memory: 1Gi

```

OpenTelemetry Collector Receivers

OpenTelemetry Collector collects data with *receivers* (plugins). There is a great variety or different receivers for different types of observability data coming from different sources. In this setup we are using the following receivers:

- OTLP to collect traces and logs from the OpenTelemetry instrumented workloads (control-plane)
- Prometheus to scrape metrics from all workloads and infra services (e.g. Flux) on the cluster
- Filelog Receiver to collect logs from *stdout* produced by non-instrumented workloads and infra services
- K8s Cluster Receiver to collect cluster-level metrics
- Kubelet Stats Receiver to collect node, pod, container and volume metrics

OpenTelemetry Collector Exporters

OpenTelemetry Collector sends observability data to different observability backends with *exporters.* In this setup we are using the following exporters:

- Azure Monitor Exporter to send all collected traces and logs to Application Insights service in Aure
- One the following (evaluating the best option):
  - Prometheus Remote Write exporter to push collected metrics directly to Azure Monitor Workspace (Managed Prometheus).
  - Prometheus Exporter to expose collected metrics via Prometheus endpoint on Otel Collector. Azure Managed Prometheus agent will pick up the metrics from the endpoint and send them to Azure Monitor Workspace (Managed Prometheus).

OpenTelemetry Collector Distribution

Following the crawl->walk->run paradigm we can start with the open source OpenTelemetry Collector Contrib Distro that contains a full list of available receivers and exporters. Moving forward, it is recommended to build our own distribution containing only the receivers and exporters that we use. The reasons:

- reduce the size of the collector
- reduce deployment times for the collector
- improve the security of the collector by reducing the available attack surface area
- eliminate dependency on the open source code that we never use

Sending Queue

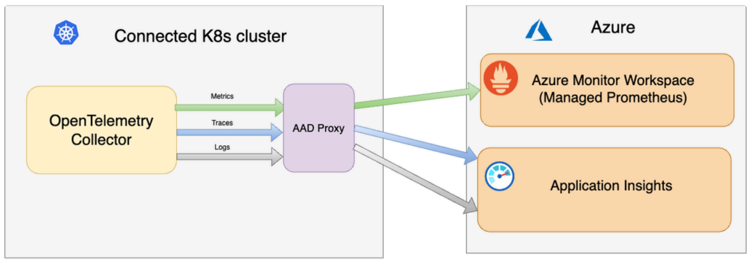
OpenTelemetry Collector is configured to collect data in batches (default=1024 items) and send them to the backend via an in-memory queue. The sending queue allows to survive loss of connection and save the data. The size of the queue is configured depending on the amount of data and desired survival time. The default queue size for traces and logs is 1000 batches (each batch is 1024 items) and for the metrics the default queue size is 10000 metrics.

The in-memory sending queue approach allows for short time connectivity issues (e.g. minutes). For the longer survival times (e.g. days) the sending queue can be configured with a persistent storage to save the data in a persistent volume outside of the cluster. In this case the data will survive not only connectivity loss but the cluster restart as well. However, this feature is available for traces and logs only.

Authentication with Azure Monitor

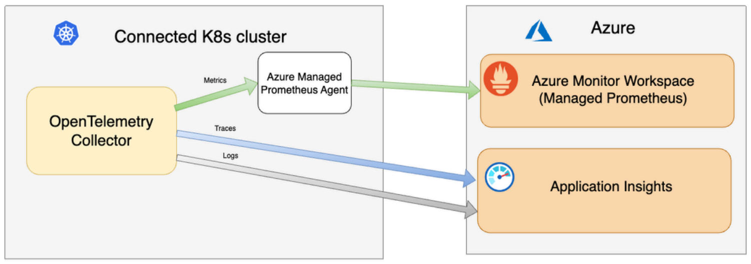
To send data to Azure Monitor the Otel Collector should be able to authenticate with Azure Monitor services. There are two options that we are considering at this point:

Managed Identity



This approach requires an additional component on the cluster - Azure Active Directory authentication proxy. It serves as a proxy between Otel Collector and Azure Monitor handling authentication via managed identity. The proxy can be deployed to a cluster as a separate deployment or as a sidecar to the Otel Collector.

Connection String



The Otel Collector uses *connection string* to authenticate with Application Insights. This approach also requires an additional component on the cluster - Azure Monitor Managed Prometheus agent. It can be installed in the cluster as an extension. This agent scrapes metrics from the Otel Collector Prometheus endpoint and sends them to Azure Monitor Workspace.

Application Instrumentation

To emit observability data, the application code should be instrumented with OpenTelemetry SDK either manually or automatically. There is a comprehensive guidance on instrumenting Python code. Here are some samples of manual instrumentation :

Traces

```python
def do_work():
    with tracer.start_as_current_span("foo") as foo:
        # do some work that 'foo' tracks
        print("doing some work...")
        # Create a nested span to track nested work
        with tracer.start_as_current_span("bar") as bar:
            # do some work that 'bar' tracks
            print("doing some nested work...")
            # the nested span is closed when it's out of scope

        # This span is also closed when it goes out of scope

```

Metrics

```python
work_counter = meter.create_counter(
    "work.counter", unit="1", description="Counts the amount of work done"
)

def do_work(work_item):
    # count the work being doing
    work_counter.add(1, {"work.type": work_item.work_type})
    print("doing some work...")
```

TODO (user stories to create):

- Define SLOs/SLIs
- Identify metrics to collect. Metrics that contribute to defined SLOs/SLIs
- Design dashboards and alerts that monitor SLOs/SLIs
- Observability data collection pipeline (configure Otel Collector on Edge cluster)
- Instrument Control Tower code with Otel SDK
- Implement Grafana dashboards
- Implement Prometheus alerts
- Provision cloud Azure Monitor services (Azure Monitor Workspace, Managed Grafana and Application Insights)
- Deploy Otel Collector with GitOps

Why OpenTelemetry

- Support of all three pillars: Distributed Tracing, Logs, Metrics. OpenTelemetry is currently the recommended way to expose distributed tracing data in the application and send it to application insights. Legacy App Insights SDKs are being deprecated.
- Ability to aggregate/process/filter data on the edge before sending it to the cloud
- Ability to split and send the collected observability data to multiple backends (e.g. Azure Monitor, ELK, New Relic, Loki, Tempo, etc). Applications are abstracted away from the observability backend.
- Ability to work and save observability data in disconnected mode for an extended period of time
- Support of multi-layered networks on the edge
- Integration with ST-One observability data is happening via OpenTelemetry
- AIO implements observability on top of OpenTelemetry and it comes with OpenTelemetry Collector
- OpenTelemetry is Microsoft's strategy regarding collecting and distributing observability data