**Low-Level Design: OpenTelemetry Observability in GKE**

**1. Introduction**

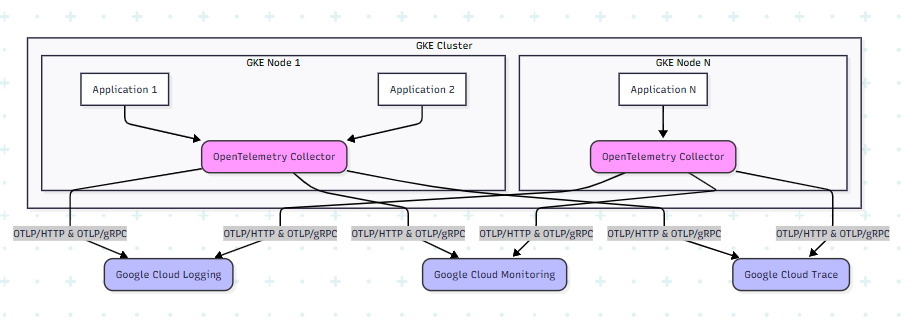
This document outlines the low-level design for implementing observability using OpenTelemetry within our Google Kubernetes Engine (GKE) environment. The primary goal is to standardize the collection of logs, metrics, and traces from our applications and centralize them within Google Cloud Observability (Cloud Logging, Cloud Monitoring, Cloud Trace) for enhanced visibility, debugging, and performance analysis.

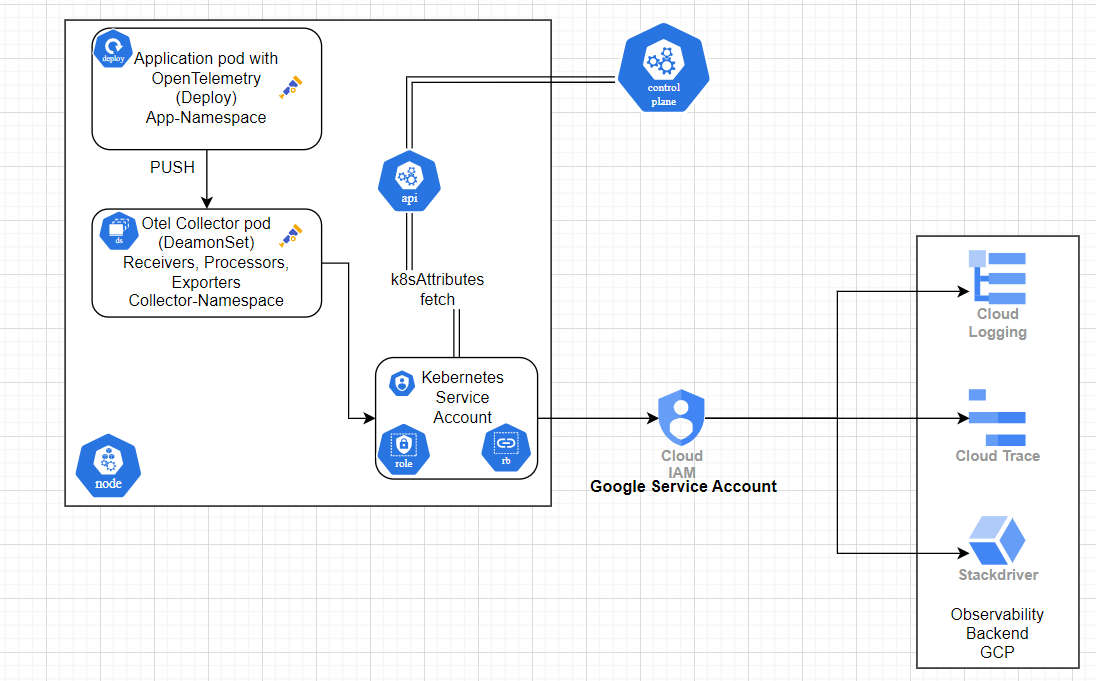
This design is intended for developers, SREs, and management as a Proof of Concept (POC) to understand the architecture and implementation details.

**2. Architecture Overview**

The observability architecture consists of three main components:

1. **Application Instrumentation**: Applications are instrumented using OpenTelemetry SDKs to generate logs, metrics, and traces.
2. **OpenTelemetry Collector DaemonSet**: A DaemonSet of OpenTelemetry Collectors runs on each GKE node, receiving telemetry data from applications, processing it, and exporting it to Google Cloud Observability.
3. **Google Cloud Observability**: The collected telemetry data is stored, visualized, and analyzed in Google Cloud Logging, Cloud Monitoring, and Cloud Trace.



****

**3. Detailed Design**

**3.1. Application Instrumentation (Frontend Service Example)**

Applications will be instrumented using OpenTelemetry SDKs, configured to export telemetry data via OTLP (OpenTelemetry Protocol) over HTTP to the OpenTelemetry Collector.

**Key aspects:**

* **Language-Specific SDKs**: Use appropriate OpenTelemetry SDKs for the application's programming language (e.g., Python opentelemetry-sdk for app.py).
* **Resource Attributes**: Define common resource attributes (e.g., service.name, environment) to enrich telemetry data with contextual information.
* **Traces**:
  + Initialize TracerProvider and BatchSpanProcessor.
  + Use OTLPSpanExporter to send traces to the OpenTelemetry Collector's OTLP/HTTP endpoint for traces (e.g., http://otel-collector-otel-collector-headless.opentelemetry.svc.cluster.local:4318/v1/traces).
  + Manually create spans for key operations and propagate context using inject(headers).
  + Set SpanKind (e.g., SERVER for incoming requests, CLIENT for outgoing requests).
  + Add relevant attributes (e.g., http.method, http.target, http.url, http.status\_code).
  + Set span status (e.g., trace.StatusCode.OK, trace.StatusCode.ERROR).
* **Metrics**:
  + Initialize MeterProvider and PeriodicExportingMetricReader.
  + Use OTLPMetricExporter to send metrics to the OpenTelemetry Collector's OTLP/HTTP endpoint for metrics (e.g., http://otel-collector-otel-collector-headless.opentelemetry.svc.cluster.local:4318/v1/metrics).
  + Define and record various metric types (e.g., Counter for request counts, Histogram for request latencies).
* **Logs**:
  + Initialize LoggerProvider and BatchLogRecordProcessor.
  + Use OTLPLogExporter to send logs to the OpenTelemetry Collector's OTLP/HTTP endpoint for logs (e.g., http://otel-collector-otel-collector-headless.opentelemetry.svc.cluster.local:4318/v1/logs).
  + Integrate with standard logging libraries (e.g., Python's logging) using LoggingHandler to automatically capture application logs.
  + Enrich logs with trace and span IDs for correlation.

**GITLAB-Repo:**

**Example (from app.py):**

# Set up tracing resource

resource = Resource(attributes={

"service.name": "frontend-service",

"environment": "dev"

})

# Set up tracer

trace.set\_tracer\_provider(TracerProvider(resource=resource))

tracer = trace.get\_tracer\_provider().get\_tracer(\_\_name\_\_)

trace.get\_tracer\_provider().add\_span\_processor(

BatchSpanProcessor(OTLPSpanExporter(endpoint= "http://otel-collector-otel-collector-headless.opentelemetry.svc.cluster.local:4318/v1/traces"))

)

# Set up metrics

metrics.set\_meter\_provider(

MeterProvider(

resource=resource,

metric\_readers=[PeriodicExportingMetricReader(OTLPMetricExporter(endpoint="http://otel-collector-otel-collector-headless.opentelemetry.svc.cluster.local:4318/v1/metrics"))]

)

)

meter = metrics.get\_meter(\_\_name\_\_)

request\_counter = meter.create\_counter(

name="http.server.request.count",

unit="1",

description="Counts number of incoming HTTP requests"

)

request\_latency = meter.create\_histogram(

name="http.server.request.duration",

unit="s",

description="Tracks request durations"

)

# Logging setup

LoggingInstrumentor().instrument(set\_logging\_format=True)

logger = logging.getLogger("frontend")

logging.basicConfig(level=logging.INFO)

# Set up OpenTelemetry logger provider

log\_provider = LoggerProvider(resource=resource)

set\_logger\_provider(log\_provider)

# Export logs to OTEL Collector

log\_exporter = OTLPLogExporter(endpoint= "http://otel-collector-otel-collector-headless.opentelemetry.svc.cluster.local:4318/v1/logs")

log\_processor = BatchLogRecordProcessor(log\_exporter)

log\_provider.add\_log\_record\_processor(log\_processor)

# Bridge Python logging -> OpenTelemetry logs

otel\_handler = LoggingHandler(level=logging.NOTSET, logger\_provider=log\_provider)

logging.getLogger().addHandler(otel\_handler)

**3.2. OpenTelemetry Collector DaemonSet**

The OpenTelemetry Collector will be deployed as a DaemonSet in the GKE cluster. This ensures that an instance of the collector runs on every node, allowing it to receive telemetry data directly from applications running on that node.

**Key components and configurations:**

* **DaemonSet (daemonset.yaml)**:
  + Ensures one collector pod per node.
  + Uses hostNetwork: true and dnsPolicy: ClusterFirstWithHostNet to allow the collector to see the real pod IPs and enable communication with applications via localhost or nodeIP.
  + Mounts a ConfigMap containing the collector configuration.
  + Sets KUBE\_NODE\_NAME environment variable from spec.nodeName for the k8sattributes processor.
  + Exposes OTLP HTTP (port 4318) and gRPC (port 4317) endpoints on hostPort for applications to send data.
* **Service Account (serviceaccount.yaml)**:
  + A dedicated Service Account (otel-collector-sa) is created for the DaemonSet.
  + This Service Account is annotated with iam.gke.io/gcp-service-account to enable Workload Identity, allowing the collector to authenticate to Google Cloud APIs using a GCP Service Account (gke-observability-sa@observability-project-466314.iam.gserviceaccount.com).
* **ClusterRole and ClusterRoleBinding (clusterrole.yaml, clusterrolebinding.yaml)**:
  + A ClusterRole is defined with permissions to get, list, and watch Kubernetes resources like pods, nodes, namespaces, replicasets, and deployments. These permissions are crucial for the k8sattributes processor to enrich telemetry data with Kubernetes metadata.
  + A ClusterRoleBinding links this ClusterRole to the otel-collector-sa Service Account, granting the necessary permissions cluster-wide.
* **Headless Service (service.yaml)**:
  + A headless service (otel-collector-otel-collector-headless) is created for the OpenTelemetry Collector. This service does not have a ClusterIP and allows direct communication with individual collector pods, which is useful when applications need to send data to a specific collector instance (though in this DaemonSet setup, localhost or nodeIP is often used). It exposes ports 4317 (gRPC) and 4318 (HTTP).
* **Collector Configuration (values.yaml - config section)**:
  + **Receivers**:
    - otlp: Configured to receive OTLP data over HTTP (port 4318) and gRPC (port 4317) on all interfaces (0.0.0.0).
  + **Processors**:
    - resourcedetection: Detects GKE resources and automatically adds relevant resource attributes (e.g., GKE cluster name, project ID).
    - k8sattributes: Enriches telemetry data with Kubernetes metadata such as k8s.namespace.name, k8s.pod.name, k8s.node.name, k8s.container.name, k8s.deployment.name, and k8s.pod.uid. It uses node\_from\_env\_var: KUBE\_NODE\_NAME for node filtering and serviceAccount as auth\_type.
    - resource: Adds static resource attributes like gcp.project.name.
    - batch: Batches telemetry data for efficient export.
  + **Exporters**:
    - googlecloud: Configured to export traces, metrics, and logs to Google Cloud Observability.
      * project: Specifies the Google Cloud project ID (observability-project-466314).
      * metric.prefix: Sets a custom prefix for metrics (custom.googleapis.com).
      * log.default\_log\_name: Defines the default log name for exported logs.
      * user\_agent: Identifies the collector as otel-collector.
  + **Service Pipelines**:
    - Defines pipelines for traces, metrics, and logs.
    - Each pipeline uses otlp as the receiver.
    - The processors are applied in a specific order: resourcedetection, k8sattributes, resource, batch. This order is crucial for proper enrichment and batching before export.
    - googlecloud is used as the exporter for all pipelines.

**Example (from values.yaml and daemonset.yaml):**

# values.yaml (config section)

config: |

receivers:

otlp:

protocols:

http:

endpoint: 0.0.0.0:4318

grpc:

endpoint: 0.0.0.0:4317

processors:

resourcedetection:

detectors: [gke]

timeout: 5s

resource:

attributes:

- action: insert

key: gcp.project.name

value: observability-project-466314

k8sattributes:

passthrough: false

filter:

node\_from\_env\_var: KUBE\_NODE\_NAME

auth\_type: serviceAccount

pod\_association:

- sources:

- from: connection

- sources:

- from: resource\_attribute

name: ip

- sources:

- from: resource\_attribute

name: k8s.pod.ip

extract:

metadata:

- k8s.namespace.name

- k8s.pod.name

- k8s.node.name

- k8s.container.name

- k8s.deployment.name

- k8s.pod.uid

exporters:

googlecloud:

project: "observability-project-466314"

metric:

prefix: "custom.googleapis.com"

log:

default\_log\_name: opentelemetry.io/collector-exported-log

trace:

attribute\_mappings: []

user\_agent: "otel-collector"

service:

pipelines:

traces:

receivers: [otlp]

processors: [resourcedetection, k8sattributes, resource, batch]

exporters: [googlecloud]

metrics:

receivers: [otlp]

processors: [resourcedetection, k8sattributes, resource, batch]

exporters: [googlecloud]

logs:

receivers: [otlp]

processors: [resourcedetection, k8sattributes, resource, batch]

exporters: [googlecloud]

# daemonset.yaml (relevant section)

spec:

template:

spec:

serviceAccountName: {{ .Values.serviceAccount.name }}

hostNetwork: true

dnsPolicy: ClusterFirstWithHostNet

containers:

- name: otel-collector

env:

- name: KUBE\_NODE\_NAME

valueFrom:

fieldRef:

fieldPath: spec.nodeName

ports:

- containerPort: 4317

name: otlp-grpc

hostPort: 4317

protocol: TCP

- containerPort: 4318

name: otlp-http

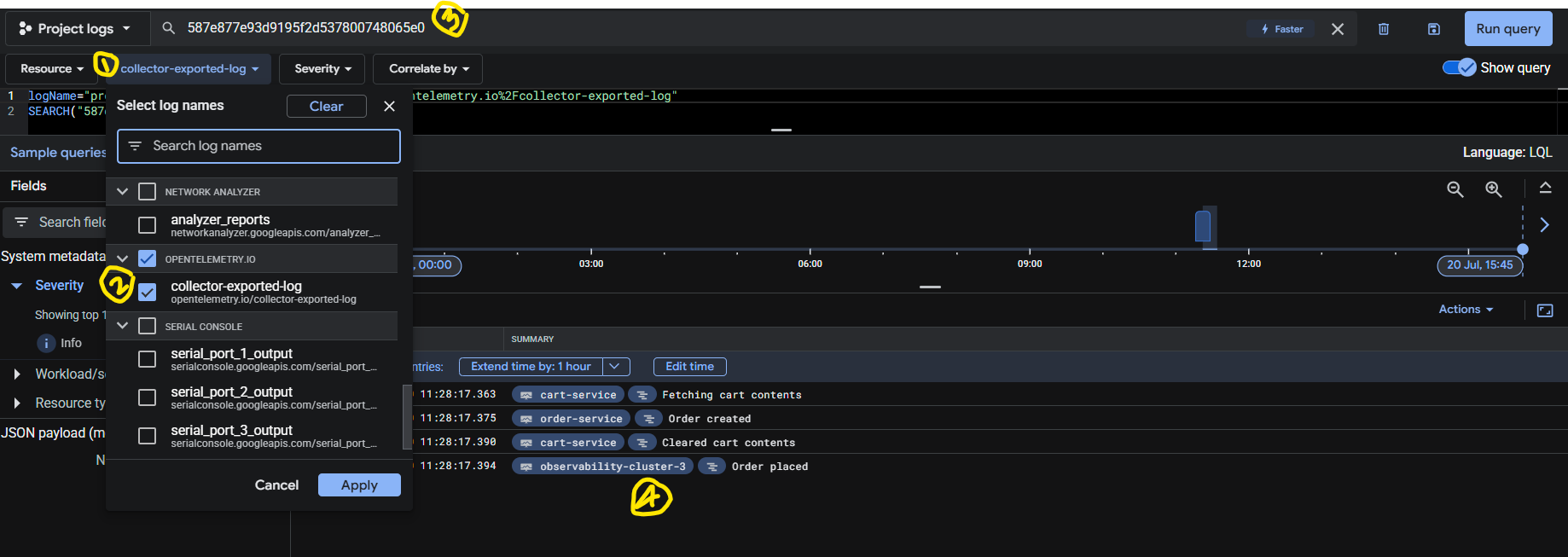
hostPort: 4318

protocol: TCP

**3.3. Google Cloud Observability**

Once exported by the OpenTelemetry Collector, the telemetry data will be available in the following Google Cloud Observability services:

* **Cloud Logging**: All logs emitted by the applications and processed by the collector will appear in Cloud Logging. The k8sattributes processor ensures that logs are enriched with Kubernetes metadata, making them easily searchable and filterable by pod name, namespace, deployment, etc.



Sample Logs:

{

insertId: "qfb0l2fgp3vuj"

**labels: {**

**code.file.path: "/app/app.py"**

**code.function.name: "get\_cart"**

**code.line.number: "104"**

**environment: "dev"**

**http.method: "GET"**

**http.status\_code: "200"**

**instrumentation\_source: "cart-service"**

**otelServiceName: "cart-service"**

**otelSpanID: "8df107df928fd2cb"**

**otelTraceID: "587e877e93d9195f2d537800748065e0"**

**otelTraceSampled: "true"**

**service.name: "cart-service"**

**}**

logName: "projects/observability-project-466314/logs/opentelemetry.io%2Fcollector-exported-log"

receiveTimestamp: "2025-07-20T05:58:22.167931894Z"

resource: {

labels: {

cluster\_name: "observability-cluster-3"

container\_name: "cart-service"

location: "asia-south1-a"

namespace\_name: "internal-app"

pod\_name: "cart-service-6874d4c876-8rfzj"

project\_id: "observability-project-466314"

}

type: "k8s\_container"

}

severity: "INFO"

spanId: "8df107df928fd2cb"

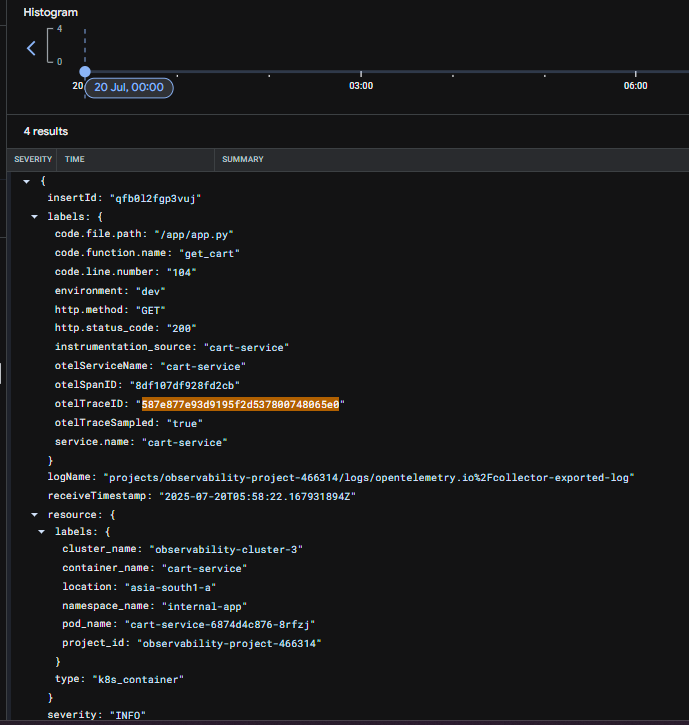
textPayload: "Fetching cart contents"

timestamp: "2025-07-20T05:58:17.363082752Z"

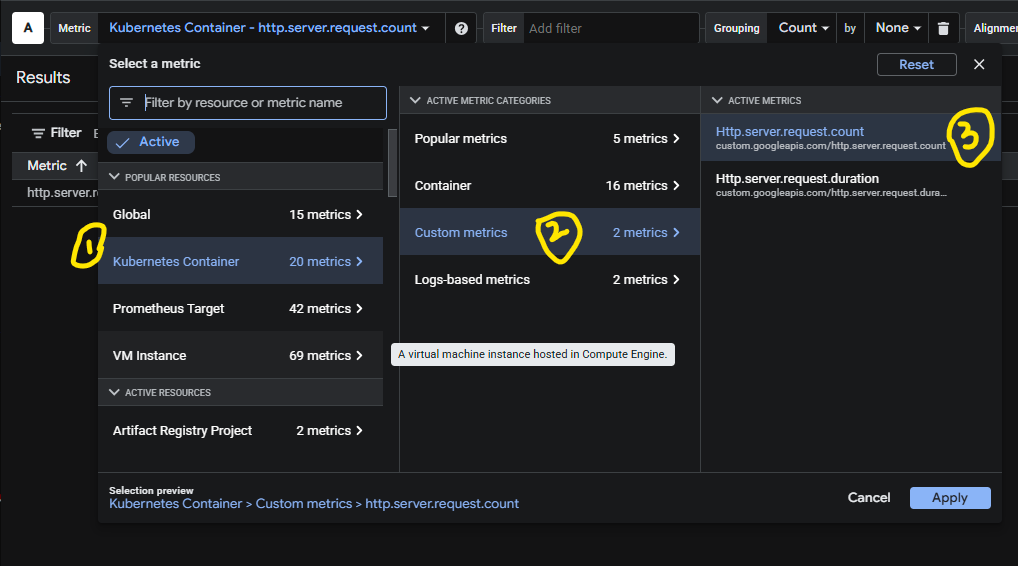
trace: "projects/observability-project-466314/traces/587e877e93d9195f2d537800748065e0"

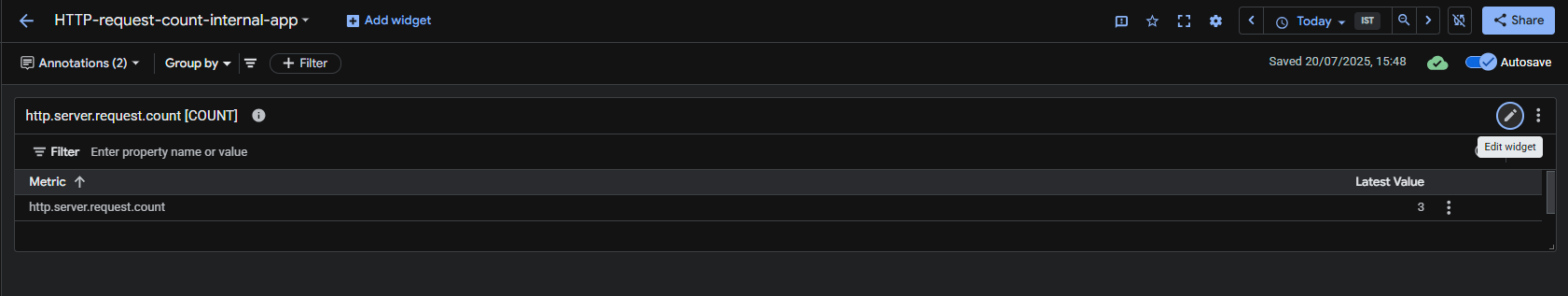
traceSampled: true

}

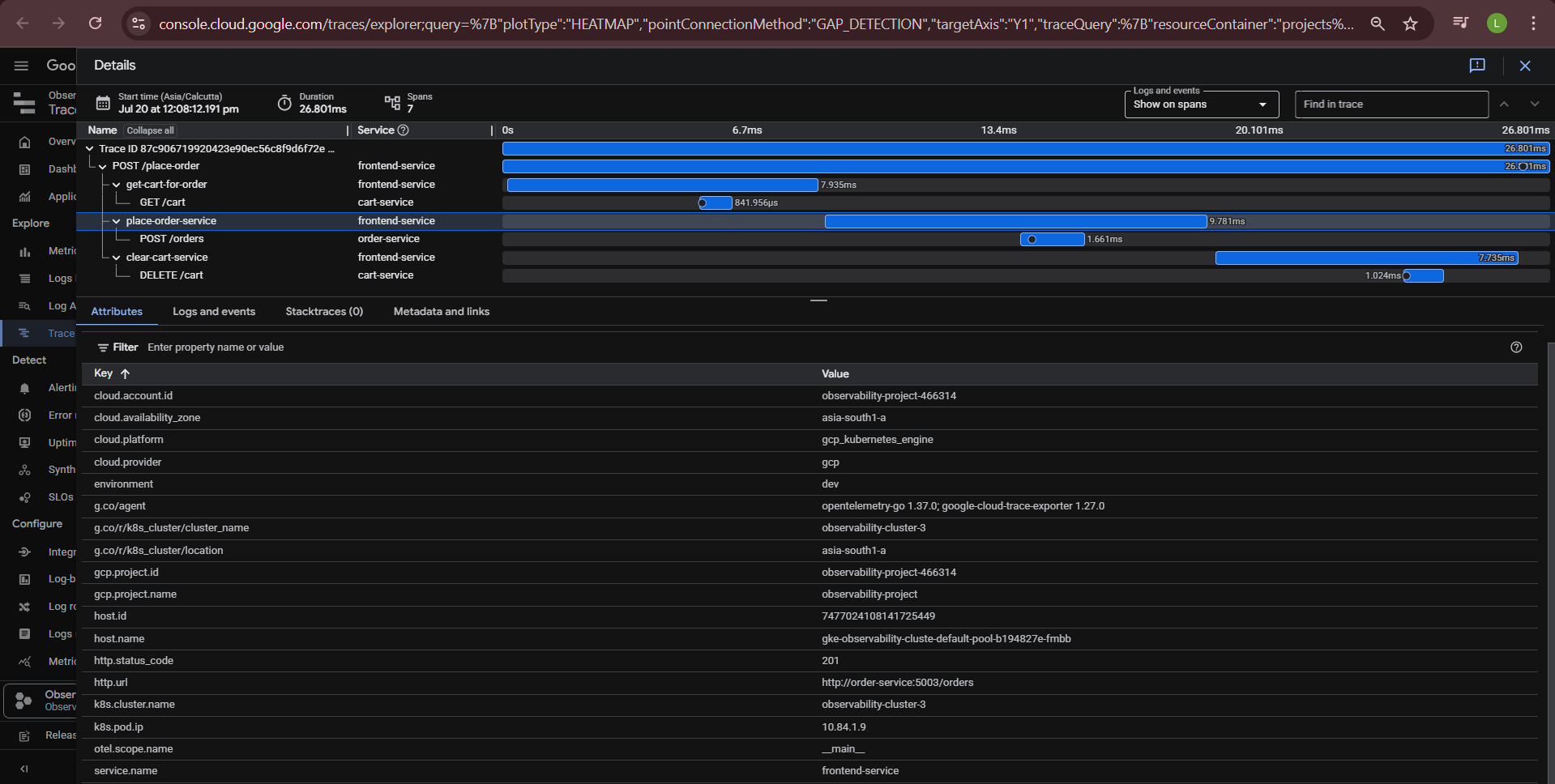


* **Cloud Monitoring**: Custom metrics defined in the applications (e.g., http.server.request.count, http.server.request.duration) will be visible in Cloud Monitoring under the custom.googleapis.com prefix. This allows for creating dashboards, alerts, and analyzing application performance.





* **Cloud Trace**: Distributed traces, including spans from the frontend service and its interactions with other services (product-service, cart-service, order-service), will be available in Cloud Trace. This enables end-to-end visibility of requests, identifying bottlenecks, and understanding service dependencies.



**4. Deployment Steps (High-Level)**

1. **GCP Project Setup**: Ensure the Google Cloud project observability-project-466314 is set up with necessary APIs enabled (Cloud Logging, Cloud Monitoring, Cloud Trace, IAM).
2. **Service Account and Workload Identity**:
   * Create a GCP Service Account (gke-observability-sa) with permissions to write logs, metrics, and traces to Google Cloud Observability.
   * Enable Workload Identity on your GKE cluster.
   * Bind the Kubernetes Service Account (otel-collector-sa) to the GCP Service Account.
3. **Helm Chart Deployment**:
   * Use the provided Helm chart to deploy the OpenTelemetry Collector DaemonSet.
   * The values.yaml file contains the configuration for the collector, including receivers, processors, and exporters.
   * Example deployment command (assuming Helm is configured):
   * helm install otel-collector . -f values.yaml --namespace opentelemetry --create-namespace
4. **Application Deployment**:
   * Deploy your instrumented applications (e.g., frontend.yaml, app.py) to the GKE cluster.
   * Ensure applications are configured to send OTLP data to the OpenTelemetry Collector's headless service (e.g., http://otel-collector-otel-collector-headless.opentelemetry.svc.cluster.local:4318).

**5. Benefits**

* **Unified Observability**: Collects logs, metrics, and traces through a single agent (OpenTelemetry Collector).
* **Vendor-Neutral Instrumentation**: OpenTelemetry provides a standardized, vendor-agnostic way to instrument applications, reducing vendor lock-in.
* **Rich Contextual Data**: The k8sattributes and resourcedetection processors automatically enrich telemetry data with valuable Kubernetes and GKE metadata, making analysis easier.
* **Scalability**: DaemonSet deployment ensures that the collector scales with your GKE cluster, providing efficient data collection from all nodes.
* **Centralized Monitoring**: All observability data is sent to Google Cloud Observability, providing a single pane of glass for monitoring and troubleshooting.
* **Reduced Overhead**: Batching and efficient export mechanisms minimize the performance impact on applications and network traffic.

**6. Future Enhancements**

* **Auto-Instrumentation**: Explore OpenTelemetry auto-instrumentation agents for supported languages to reduce manual instrumentation effort.
* **Advanced Processing**: Implement additional processors in the collector for data filtering, sampling, or aggregation based on specific requirements.
* **Alerting and Dashboards**: Develop comprehensive dashboards and alerting rules in Google Cloud Monitoring based on the collected metrics.
* **Cost Optimization**: Monitor data ingestion volumes and implement sampling strategies to optimize observability costs.

**7. Current Observations and Clarifications**

**7.1. Kubernetes Attributes in Cloud Logging**

**Observation**: It has been observed that Kubernetes attributes such as k8s.namespace.name, k8s.pod.name, and k8s.node.name are not appearing directly under the labels field in Google Cloud Logging log entries as might be initially expected.

Context and Clarification: This behavior is actually as intended and reflects the structured nature of Google Cloud Logging's LogEntry format, particularly when integrating with OpenTelemetry Collector's googlecloud exporter and the k8sattributes processor.

****

**8. Commands Used and Roles:**

**8.1 Commands:**

helm install otel-collector ./otel-collector-helm -n opentelemetry --create-namespace -f otel-collector-helm/values.yaml

helm uninstall otel-collector -n opentelemetry

helm upgrade otel-collector ./otel-collector-helm -f otel-collector-helm/values.yaml -n opentelemetry

gcloud iam service-accounts add-iam-policy-binding \

gke-observability-sa@observability-project-466314.iam.gserviceaccount.com \

--role roles/iam.workloadIdentityUser \

--member "serviceAccount:observability-project-466314.svc.id.goog[opentelemetry/otel-collector-sa]"

gcloud iam service-accounts add-iam-policy-binding \

gke-observability-sa@observability-project-466314.iam.gserviceaccount.com \

--role roles/iam.serviceAccountTokenCreator \

--member "serviceAccount:observability-project-466314.svc.id.goog[opentelemetry/otel-collector-sa]"

8.2 Roles:

1. Artifact Registry Reader
2. Artifact Registry Writer
3. Cloud Trace Agent
4. Logs Writer
5. Monitoring Metric Writer
6. Service Account Token Creator
7. Workload Identity User