# OpenTelemetry PoC Plan on Google Cloud Platform

The goal is to prove that OpenTelemetry (OTel) can provide end-to-end observability for our microservices on GCP. We will demonstrate distributed tracing, metrics, and structured logging from a sample application into Google Cloud’s observability tools (Cloud Trace, Cloud Monitoring, Cloud Logging). OpenTelemetry is an open-source, vendor-neutral framework for collecting telemetry data[[1]](https://opentelemetry.io/#:~:text=), so the PoC will also highlight its flexibility (e.g. the ability to export to GCP and/or third-party backends).

## Objectives and Use Cases

* **Use Case 1 – Distributed Tracing:** Instrument a simple multi-tier service so that a request can be tracked across components. For example, a FastAPI app calls a backend function or another endpoint. Using OTel, each span is correlated so Cloud Trace shows an end-to-end waterfall (latency breakdown and bottlenecks). This validates that spans carry trace IDs across services.
* **Use Case 2 – Metrics & Logging:** Collect application metrics such as request count, latency histograms, error rates, throughput. Ensure OTel exports metrics to Cloud Monitoring (via Managed Prometheus) or a custom backend. Also structure logs with trace IDs so logs in Cloud Logging can be filtered by trace. We capture key logs/errors in JSON format and verify trace correlation.

These use cases prove that OTel can capture traces, metrics, and logs for our services, and export them to Google Cloud’s native observability tools (or alternative backends, since OTel is vendor-neutral[[1]](https://opentelemetry.io/#:~:text=)).

## Sample Microservice Implementation

We use a simple **Python FastAPI** microservice with two endpoints (/ping, /process) to generate test traffic. Key steps:

* **Set up FastAPI app:** Create a FastAPI application in Python. Define at least two routes (e.g. GET /ping returns “pong”, GET /process simulates work). In /process, include a call to another function or simulated downstream service (e.g. a time delay or HTTP call) to test distributed tracing. Inject some latency or occasional errors (e.g. time.sleep(), raising HTTPException(500)) to test error spans.
* **Instrument with OpenTelemetry:** Install OTel libraries (pip install opentelemetry-instrumentation-fastapi opentelemetry-exporter-otlp). In the code, auto-instrument FastAPI by calling:
* from opentelemetry.instrumentation.fastapi import FastAPIInstrumentor  
  app = FastAPI()  
  FastAPIInstrumentor.instrument\_app(app)
* This automatically creates spans for incoming HTTP requests[[2]](https://opentelemetry-python-contrib.readthedocs.io/en/latest/instrumentation/fastapi/fastapi.html#:~:text=import%20fastapi%20from%20opentelemetry,FastAPIInstrumentor). You can also add manual spans around key logic if needed.
* **Metrics Recording:** Use the OTel Python metrics SDK to record custom metrics. For example, create a MeterProvider with a CloudMonitoringMetricsExporter to send metrics to Cloud Monitoring. Google’s example shows:
* from opentelemetry import metrics  
  from opentelemetry.exporter.cloud\_monitoring import CloudMonitoringMetricsExporter  
  from opentelemetry.sdk.metrics import MeterProvider  
  from opentelemetry.sdk.metrics.export import PeriodicExportingMetricReader  
    
  metrics.set\_meter\_provider(  
   MeterProvider(  
   metric\_readers=[  
   PeriodicExportingMetricReader(  
   CloudMonitoringMetricsExporter(),  
   export\_interval\_millis=5000  
   )  
   ],  
   resource=Resource.create({  
   "service.name": "myservice",  
   "service.instance.id": "instance1",  
   })  
   )  
  )  
  meter = metrics.get\_meter(\_\_name\_\_)  
  request\_counter = meter.create\_counter("request\_count", unit="1")  
  request\_counter.add(1, {"endpoint": "/ping"})
* This sends a workload.googleapis.com/request\_count metric to Cloud Monitoring[[3]](https://google-cloud-opentelemetry.readthedocs.io/en/latest/examples/cloud_monitoring/README.html#:~:text=from%20opentelemetry%20import%20metrics%20from,resources%20import%20Resource)[[4]](https://google-cloud-opentelemetry.readthedocs.io/en/latest/examples/cloud_monitoring/README.html#:~:text=,). Similar constructs (histograms or gauges) can track latency or errors.
* **Logging with Trace Context:** Ensure logs include trace and span IDs. Use the OTel logging integration by enabling trace-context injection. For example, set the environment variable OTEL\_PYTHON\_LOG\_CORRELATION=true or use LoggingInstrumentor(set\_logging\_format=True). This will prepend each log record with trace\_id and span\_id fields[[5]](https://opentelemetry-python-contrib.readthedocs.io/en/latest/instrumentation/logging/logging.html#:~:text=The%20integration%20is%20opt,true). Thus, when these JSON logs appear in Cloud Logging, you can filter them by trace ID.
* **Testing the Service:** Run the service locally or in a dev container and verify OTel is adding spans and metrics. (OTel auto-instrumentation should emit OTLP to localhost:4317 by default.)

## GCP Infrastructure Setup

1. **Project and APIs:** Create a dedicated GCP project for the PoC. Ensure the **Cloud Monitoring**, **Cloud Trace**, and **Cloud Logging** APIs are enabled[[6]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-gke#:~:text=Running%20the%20Google,requires%20the%20following%20resources). These services collect the telemetry. (New projects have these APIs enabled by default[[6]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-gke#:~:text=Running%20the%20Google,requires%20the%20following%20resources), but verify in APIs & Services.)
2. **Service Accounts & IAM:**
3. For **GKE Autopilot**: Enable Workload Identity. Create a Google service account (GSA) that will be used by the OTel Collector pods. Grant that GSA the roles roles/monitoring.metricWriter, roles/logging.logWriter, and roles/cloudtrace.agent[[7]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-gke#:~:text=,roles%2Fcloudtrace.agent). Then bind it to the Kubernetes service account in the collector namespace. This lets the collector push metrics to Monitoring and logs/traces to Cloud Logging/Trace.
4. For **Cloud Run**: By default, Cloud Run uses the Compute Engine default service account, which already has roles/monitoring.metricWriter and roles/logging.logWriter[[8]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=By%20default%2C%20Cloud%20Run%20jobs,logs%20described%20in%20this%20document). Ensure the Cloud Run Admin role (roles/run.admin) and Secret Manager Secret Accessor (roles/secretmanager.secretAccessor) are also granted to deploy services with sidecars[[9]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=Ask%20your%20administrator%20to%20grant,IAM%20roles%20on%20your%20project). (This is needed because we’ll store the collector config in Secret Manager.)
5. **Networking:** (If needed, configure VPC or firewall rules so that your services can talk to each other. For a simple PoC in one region, the default networking is usually sufficient.)
6. **Cluster/Service Provisioning:**
7. **GKE Autopilot:** Use the Cloud Console or gcloud to create an Autopilot GKE cluster (or standard GKE if preferred). Autopilot handles node provisioning. Ensure it’s in the same region as your project resources.
8. **Cloud Run:** No cluster needed – we will deploy containers directly to Cloud Run. Enable the Cloud Run API and set up gcloud for Cloud Run (if not already).

## Deploying OpenTelemetry Collector

We will deploy the Google-built OTel Collector in each environment to receive OTLP data and export to GCP:

* **Collector Config:** Use Google’s recommended collector configuration as a base (available in the *opentelemetry-operations-collector* GitHub). The config should include: OTLP receivers (gRPC on 4317, HTTP on 4318), necessary processors (batching, memory limiter, resource detection for GCP), and exporters: googlecloud (exports traces/logs to Cloud Trace/Logging/Monitoring) and googlemanagedprometheus (for metrics to Managed Prometheus/Cloud Monitoring)[[10]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=exporters%3A%20,contrib%2Ftree%2Fmain%2Fexporter%2Fgooglecloudexporter)[[11]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=,contrib%2Ftree%2Fmain%2Fexporter%2Fgooglemanagedprometheusexporter%20googlemanagedprometheus). For example, in the service: section define pipelines for logs, metrics/otlp, and traces all using the OTLP receiver, with googlecloud or googlemanagedprometheus exporters as appropriate[[12]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=service%3A%20extensions%3A%20,batch%20exporters)[[13]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=,batch%20exporters)[[14]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=,googlecloud). This ensures logs and traces go to Cloud Logging/Trace, and metrics to Cloud Monitoring.
* **On GKE Autopilot:** Deploy the collector as a Kubernetes workload. Google provides a kustomize manifest. For example:
* export GOOGLE\_CLOUD\_PROJECT=PROJECT\_ID  
  export PROJECT\_NUMBER=PROJECT\_NUMBER  
  kubectl create namespace opentelemetry  
  kubectl kustomize https://github.com/GoogleCloudPlatform/otlp-k8s-ingest.git/k8s/base \  
   | sed "s/{{PROJECT\_ID}}/$GOOGLE\_CLOUD\_PROJECT/g; s/{{PROJECT\_NUMBER}}/$PROJECT\_NUMBER/g" \  
   | kubectl apply -f -
* This creates a Deployment in namespace opentelemetry with the collector pod (using the GCP-built image). It automatically configures the correct exporters and service account (the manifest uses Workload Identity)[[15]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-gke#:~:text=The%20Collector%20pipeline%20can%20be,of%20your%20Google%20Cloud%20project). If using Helm or the OTel Operator, similarly configure a otelcol deployment in sidecar mode or as a DaemonSet.
* **On Cloud Run:** We deploy the collector as a *sidecar* to our service. First create a secret with the collector config YAML:
* gcloud secrets create otel-collector-config --data-file=config.yaml --project=PROJECT\_ID
* In the Cloud Run service YAML, add an additional container for the collector. For example:
* containers:  
  - image: gcr.io/myproj/my-fastapi-app  
   name: app  
   ports: [{containerPort: 8080}]  
   env:  
   - name: OTEL\_EXPORTER\_OTLP\_ENDPOINT  
   value: "http://localhost:4317"  
  - image: us-docker.pkg.dev/cloud-ops-agents-artifacts/google-cloud-opentelemetry-collector/otelcol-google:latest  
   name: collector  
   args: ["--config=/etc/otelcol-google/config.yaml"]  
   volumeMounts:  
   - mountPath: /etc/otelcol-google/  
   name: config  
  volumes:  
  - name: config  
   secret:  
   secretName: otel-collector-config
* This runs the OTel Collector sidecar in the same pod as the app. The app container is pointed at localhost:4317 (the collector) via OTEL\_EXPORTER\_OTLP\_ENDPOINT. The collector reads its config from the mounted secret. Use the annotation run.googleapis.com/launch-stage: ALPHA and run.googleapis.com/container-dependencies: "{app:[collector]}" to ensure correct startup order[[16]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=,google%3A0.131.0%22%20args).

## Instrumentation and Configuration

* **Configure the SDK:** In your FastAPI code, set the OTLP endpoint to the collector. For GKE, the collector service might listen on a Kubernetes Service (e.g. otel-collector.opentelemetry.svc.cluster.local:4317). For Cloud Run, we set OTEL\_EXPORTER\_OTLP\_ENDPOINT=http://localhost:4317 as above.
* **Add Instrumentation Libraries:** Ensure the Python app has the OTel SDK and any required auto-instrumentation libraries installed. For example, opentelemetry-exporter-otlp is used to send to the collector.
* **Logging:** Enable OTel log correlation. E.g. add in code or set OTEL\_PYTHON\_LOG\_CORRELATION=true, then use the standard Python logging module. Logs will automatically contain [trace\_id=… span\_id=…] in the output[[5]](https://opentelemetry-python-contrib.readthedocs.io/en/latest/instrumentation/logging/logging.html#:~:text=The%20integration%20is%20opt,true). In production, use JSON logging for rich logs in Cloud Logging.
* **Launch the Services:** Deploy the FastAPI app to both environments:
* **On GKE:** Package the app in a container (e.g. using Cloud Build). Create a Kubernetes Deployment and Service. You may instrument it by injecting an OTel sidecar (Kubernetes operator can do this with an annotation), or simply configure the app to push to a separate collector service via OTLP. For simplicity, one can also deploy the app and collector in the same namespace and have the app send OTLP to the collector service.
* **On Cloud Run:** Deploy the service with the YAML above (app + collector). Verify it starts and can be invoked (e.g. test via curl).

## Validation and Evidence

1. **Generate Traffic:** Send test requests to your service (e.g. curl https://<service>/ping and /process). Include bursts and some failure cases (timeout or error endpoint) to produce various spans and metrics.
2. **Check Cloud Trace:** Go to the Cloud Trace console. You should see traces corresponding to your requests, with spans for each service call. A successful end-to-end trace (frontend → backend) confirms distributed tracing works. (You can search traces by latency or by trace ID carried in logs.)
3. **Check Cloud Monitoring:** In Metrics Explorer or Dashboards, look for your custom metrics (e.g. workload.googleapis.com/request\_count) or for generic OTEL metrics. You can also chart request latency. The Managed Prometheus metrics (prefixed prometheus.googleapis.com) should appear if using that exporter[[12]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=service%3A%20extensions%3A%20,batch%20exporters)[[13]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=,batch%20exporters).
4. **Check Cloud Logging:** In Cloud Logging, find logs from your service. They should include trace: and span: IDs if you enabled log correlation. These IDs should match the trace IDs seen in Cloud Trace. For example, filter logs by the trace ID to see logs for a given request.
5. **Capture Evidence:** Take screenshots of:
6. A Cloud Trace **trace view** (waterfall) showing the spans and latency breakdown.
7. A Cloud Monitoring **dashboard or chart** showing the key metrics (latency, error count, throughput).
8. A Cloud Logging view showing log entries with embedded trace\_id.

All these prove end-to-end visibility. Success criteria are: *every* request has a visible trace in Cloud Trace; metrics are charted in Cloud Monitoring; and logs are correlated.

## Recommendations for Success

* **Use Multiple Backends:** Besides GCP, you can configure a secondary backend to show vendor neutrality. For example, run Jaeger or install Prometheus + Grafana. OTel Collector can export to Jaeger or Prometheus/Grafana in parallel, demonstrating flexibility. (OpenTelemetry is explicitly vendor-neutral[[1]](https://opentelemetry.io/#:~:text=).)
* **Error Injection:** Include failure cases in your app (timeouts, exceptions) to ensure errors show up in traces and logs. This highlights OTel’s diagnostic value.
* **Automate Deployment:** For repeatability, consider using Terraform or Helm charts. Although the PoC is manual now, codifying the steps ensures production rollout is smoother. For example, use a Helm chart or KubernetesOperator custom resource to deploy the collector, and Terraform to create GCP roles and secrets.
* **Monitor Costs:** Keep an eye on Cloud Monitoring and Logging usage. OTel can generate high-volume telemetry; check billing data. Use log-sampling or metric retention settings if needed. (Cloud Logging pricing is ~$0.01/GB/month after free tier.)
* **Document the Process:** Record all configuration steps, commands, and troubleshooting tips. This documentation will speed up full implementation later.

If the PoC succeeds, we will have concrete evidence (screenshots and data) that OTel provides robust tracing, metrics, and logging integration on GCP. This justifies an enterprise-wide rollout, with GCP-native observability for our services (and optional additional backends as needed).

**Outcome:** The PoC should convincingly show *connected* traces (spanning services), ingested metrics, and correlated logs in GCP. Achieving this with OTel will demonstrate its value as a unified observability solution. If all criteria are met (traces + metrics + logs flowing to Cloud Trace/Monitoring/Logging), we can recommend moving to production instrumentation using this approach.

[[1]](https://opentelemetry.io/#:~:text=) OpenTelemetry

<https://opentelemetry.io/>

[[2]](https://opentelemetry-python-contrib.readthedocs.io/en/latest/instrumentation/fastapi/fastapi.html#:~:text=import%20fastapi%20from%20opentelemetry,FastAPIInstrumentor) OpenTelemetry FastAPI Instrumentation — OpenTelemetry Python Contrib documentation

<https://opentelemetry-python-contrib.readthedocs.io/en/latest/instrumentation/fastapi/fastapi.html>

[[3]](https://google-cloud-opentelemetry.readthedocs.io/en/latest/examples/cloud_monitoring/README.html#:~:text=from%20opentelemetry%20import%20metrics%20from,resources%20import%20Resource) [[4]](https://google-cloud-opentelemetry.readthedocs.io/en/latest/examples/cloud_monitoring/README.html#:~:text=,) Cloud Monitoring Exporter Example — Google Cloud OpenTelemetry documentation

<https://google-cloud-opentelemetry.readthedocs.io/en/latest/examples/cloud_monitoring/README.html>

[[5]](https://opentelemetry-python-contrib.readthedocs.io/en/latest/instrumentation/logging/logging.html#:~:text=The%20integration%20is%20opt,true) OpenTelemetry Logging Instrumentation — OpenTelemetry Python Contrib documentation

<https://opentelemetry-python-contrib.readthedocs.io/en/latest/instrumentation/logging/logging.html>

[[6]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-gke#:~:text=Running%20the%20Google,requires%20the%20following%20resources) [[7]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-gke#:~:text=,roles%2Fcloudtrace.agent) [[15]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-gke#:~:text=The%20Collector%20pipeline%20can%20be,of%20your%20Google%20Cloud%20project) Deploy Google-Built OpenTelemetry Collector on Google Kubernetes Engine  |  Google Cloud Observability

<https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-gke>

[[8]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=By%20default%2C%20Cloud%20Run%20jobs,logs%20described%20in%20this%20document) [[9]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=Ask%20your%20administrator%20to%20grant,IAM%20roles%20on%20your%20project) [[10]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=exporters%3A%20,contrib%2Ftree%2Fmain%2Fexporter%2Fgooglecloudexporter) [[11]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=,contrib%2Ftree%2Fmain%2Fexporter%2Fgooglemanagedprometheusexporter%20googlemanagedprometheus) [[12]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=service%3A%20extensions%3A%20,batch%20exporters) [[13]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=,batch%20exporters) [[14]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=,googlecloud) [[16]](https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run#:~:text=,google%3A0.131.0%22%20args) Deploy Google-Built OpenTelemetry Collector on Cloud Run  |  Google Cloud Observability

<https://cloud.google.com/stackdriver/docs/instrumentation/opentelemetry-collector-cloud-run>