

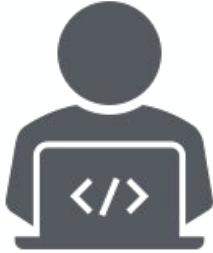
Microservices Architecture

Drake Jerez, Luis Soler, Sam Reeves

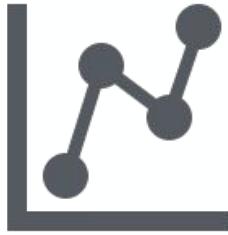
djerez@purdue.edu, solerl@purdue.edu, reeves51@purdue.edu



Introduction



Context: Developing a microservice-based application using Python, HTTP, and gRPC.



Goal: Instrument the application to generate and collect traces and metrics using OpenTelemetry (OTel).



Objective: Achieve real-time visibility into service health, performance, and transaction flows.

Problem Statement: **Observability**

Microservice architecture brings the opportunity for integrating telemetry services independently.

This addresses the observability demand that most application providers have, which in turn enables optimization and quick remediation.



Complexity: Modern distributed systems have complex inter-service dependencies.



The "Black Box" Issue: When a failure occurs (e.g., high latency), it is difficult to pinpoint the root cause without deep visibility.



Why We Care: Downtime is expensive. We need to know exactly which service failed and why, without manually digging through isolated logs.

Key Ideas and Challenges

Distributed Tracing

Tracking a single request as it propagates across multiple service boundaries (Service A → Service B → Database).

This allows us to visualize the entire lifecycle of a transaction, identifying exactly where bottlenecks occur (e.g., waiting for a database query vs. waiting for an external API).

Containerization

Utilizing Docker to package services as lightweight, portable units.

Ensures consistency between development, testing, and production environments, eliminating the "it works on my machine" problem by bundling dependencies directly with the application code.

OpenTelemetry

Provides a unified standard for generating and collecting data, preventing vendor lock-in and allowing us to switch backend visualization tools (like Jaeger, Zipkin, or Prometheus) without rewriting application code.

Key Ideas and Challenges (Cont.)

Custom Instrumentation

Integrating OTel SDKs into custom Python logic without impacting application performance.

Automatic instrumentation libraries can only capture standard HTTP calls; we must manually code "spans" for internal business logic to get meaningful data, which risks adding latency if done poorly.

Traffic Simulation

Creating realistic, high-volume traffic patterns to stress-test the telemetry pipeline.

A static system generates no useful data. We must script dynamic scenarios (e.g., sudden spikes, error rates) to verify that our monitoring alerts trigger correctly under stress.

Collector Config

Aggregating data from diverse sources (logs, metrics, traces) and exporting it correctly to visualization tools.

The OpenTelemetry Collector is a complex data processing pipeline. Configuring its receivers, processors, and exporters to handle high throughput without dropping data packets is a significant engineering hurdle.

Solution Architecture

Application Container	Auth Container	DB Container	Telemetry Layer	Telemetry Visualization
Built on Python 3.11 and FastAPI; exposes port 8080 for HTTP/gRPC ingress. Orchestrates requests between Auth and DB services while managing trace context propagation.	Lightweight Python service utilizing PyJWT for stateless token generation. Validates headers to secure internal endpoints without retaining session data, ensuring high horizontal scalability.	Python service using a lightweight SQLite backend. Eliminates complex external dependencies while maintaining persistence and supporting artificial latency injection for realistic trace visualization.	OpenTelemetry Collector (Contrib image) deployed as a Kubernetes sidecar. Configured with OTLP receivers to batch traces and metrics locally before transmission to prevent network congestion.	A dual-export pipeline routing trace data to Jaeger for waterfall analysis and metrics to Prometheus for time-series health monitoring of the cluster.

Baseline Solution

Environment	Observability State	Performance Benchmark	Purpose	Resource Footprint
Uninstrumented Kubernetes Deployment. The application microservices are deployed to the cluster using standard Kubernetes manifests (Deployments/Services) but <i>without</i> any OpenTelemetry libraries or sidecars.	"Dark" Cluster / CLI Only. Debugging relies entirely on manual kubectl logs commands and describing pod events, representing the "blind" state of standard deployments.	Establishing Latency "Floor". We will measure the application's response time and throughput in this raw state to determine the minimum latency before instrumentation overhead is introduced.	Variable Isolation. By comparing this baseline against the fully instrumented deployment, we can scientifically calculate the exact CPU/Memory cost of running OpenTelemetry in production.	Base CPU & Memory Usage. We will record the resting and active resource consumption of the plain Python containers to establish a cost baseline, ensuring we can accurately quantify the overhead introduced by the future OTEL sidecars.

Deployment Plan

- **Cluster Infrastructure:**
Application services (Auth, App, DB) deployed as isolated Pods on a Kubernetes cluster, managed via declarative manifests for independent scaling and resilience.

▪ This isolation ensures that a failure in one service (e.g., Auth) does not crash the others, allowing us to maintain partial system availability during testing.
- **Network Strategy:**
Internal traffic flows via stable ClusterIP services, while external access (Traffic Generator) is managed through Ingress controllers.

▪ We utilize standard K8s networking to simulate real-world traffic entry points, routing external requests to the correct internal microservice via a load balancer.
- **Instrumentation Layer:**
OpenTelemetry Collector Sidecars are injected into every application pod, scraping metrics locally to ensure low-latency data collection.

▪ By running the collector as a sidecar, trace data is offloaded immediately from the application process to the local collector agent, minimizing the performance impact on the main business logic.

Evaluation Plan

Telemetry vs Load

Metric: Telemetry volume must correlate 1:1 with traffic generator volume.

We will graph the "Requests Sent" by the generator against "Spans Received" by the collector; a discrepancy indicates data loss in the pipeline.

System Health

Metric: 100% of Pods must remain "Running" during standard load tests.

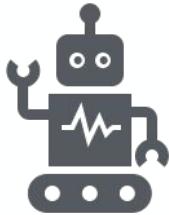
Using Kubernetes Liveness and Readiness probes to verify that the overhead of telemetry collection does not crash the application containers.

Trace Completeness

Test: Trigger a specific "test-id" request and verify the full end-to-end trace appears in the visualizer without broken spans.

Manually validating that the parent trace ID is successfully propagated from the App Service -> Auth Service -> Database Service, ensuring no broken links in the distributed graph.

Conclusion



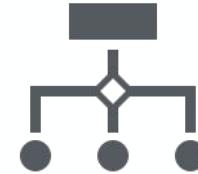
Full Observability:
Transformation of a "black box" system into a transparent, observable architecture.

Moving from reactive debugging (waiting for users to complain) to proactive monitoring (seeing errors as they happen).



Scalability Validation: Proof that the telemetry pipeline can scale alongside the application traffic.

Demonstrating that the OpenTelemetry Collector does not become a bottleneck even as we increase the number of application pods.



Project Status: Planning phase complete; ready for implementation of the traffic generator and OTEL instrumentation.

Architecture is defined, technologies are selected, and the team is ready to begin the coding and deployment phases.

Thank You

