Complete OpenTelemetry Observability Project Guide

Executive Summary

This comprehensive guide walks through deploying the OpenTelemetry (OTel) Astronomy Shop demonstration application—a production-grade microservice architecture for learning observability. This project showcases how to collect telemetry data (metrics, logs, and traces) from complex distributed systems using OpenTelemetry, then visualize and analyze them using industry-standard tools like Prometheus, Grafana, Jaeger, and OpenSearch[1].

The guide covers architecture understanding, step-by-step deployment on AWS, configuration management, data analysis, and integration with multiple observability backends. By completing this project, you'll demonstrate enterprise-level DevOps and observability expertise[2].

Table of Contents

Understanding OpenTelemetry

What is OpenTelemetry?

OpenTelemetry (OTel) is an open standard for data collection that is vendor-agnostic and maintained by the Cloud Native Computing Foundation (CNCF). As the second-most active CNCF project after Kubernetes, it provides a unified approach to observability[1].

Key Characteristics:

- **Vendor-agnostic**: Works with any monitoring platform (Prometheus, Grafana, Datadog, New Relic, Splunk, Dynatrace, etc.)
- **Language-agnostic**: Supports 10+ programming languages with native SDKs
- **Standardized**: Defines semantic conventions for consistent telemetry data
- **Open-source**: Community-driven development and contributions

The Three Pillars of Observability

Observability relies on three complementary data types to provide complete system visibility[3]:

1. Metrics

Purpose: Track system performance and health over time

Characteristics:

- Numerical measurements aggregated at specific intervals
- Include CPU usage, memory consumption, request rate, latency, error rate
- Stored efficiently due to time-series nature
- Enable trend analysis and anomaly detection

Example:
service_cpu_usage: 65%
http_request_duration: 150ms
error_rate: 2.5%

2. Logs

Purpose: Capture detailed, moment-specific information about system events

Characteristics:

- Text-based or structured records of events and errors
- Include timestamps, severity levels, and contextual information
- Provide detailed debugging information
- Historical record of system behavior

Example:
{
"timestamp": "2024-12-04T10:08:00Z",
"level": "ERROR",
"service": "payment-service",
"message": "Payment processing failed",
"error": "Connection timeout",
"trace_id": "4bf92f3577b34da6a3ce929d0e0e4736"
}

3. Traces

Purpose: Show the complete journey of requests across microservices

Characteristics:

- Map request flow through distributed systems
- Identify latency sources and bottlenecks
- Show service dependencies and communication patterns
- Enable root cause analysis

Example:
Request Flow: LoadGenerator → FrontEnd Proxy → Frontend → Ad Service

Span 1: LoadGenerator (0-50ms)
Span 2: FrontEnd Proxy (50-100ms)
Span 3: Frontend (100-150ms)
Span 4: Ad Service (150-250ms, ERROR)

OpenTelemetry Collector Architecture

The OTel Collector is a standalone service that receives, processes, and exports telemetry data[4]:

Applications → Collector → Monitoring Backends
↓
(Receivers)
↓
(Processors)
↓
(Exporters)
↓
Multiple Destinations

Key Components

Receivers (Data Ingestion)

- OTLP: OpenTelemetry Protocol (native, recommended)
- Jaeger: Distributed tracing format
- Prometheus: Metrics scraping
- OpenCensus: Legacy tracing format
- Many others for specific use cases

Processors (Data Enhancement)

- Batch: Group data for efficiency
- Attributes: Add metadata to spans
- Resource Detection: Automatically add resource information
- Sampling: Reduce data volume intelligently
- Memory Limiter: Prevent resource exhaustion

Exporters (Data Delivery)

- OTLP: To other OTel services or backends
- Prometheus: For metrics
- Jaeger: For traces
- OpenSearch: For logs
- Splunk, Datadog, New Relic, etc.

Extensions (Optional Features)

- Health Check: Collector status monitoring
- Authentication: Secure communication
- zPages: In-process diagnostics

Architecture Overview

OpenTelemetry Astronomy Shop Demo

The demo application is a cloud-native e-commerce system built to showcase OpenTelemetry capabilities across multiple programming languages and services[1].

Microservices Overview

The application consists of 18+ microservices built in different languages:

Service	Language	Purpose
Frontend	JavaScript/TypeScript	Web interface for the store
Frontend Proxy	Go	API gateway and request routing
Ad Service	Java	Advertisement recommendations
Cart Service	.NET	Shopping cart management
Checkout Service	Go	Order processing
Payment Service	JavaScript/Node.js	Payment processing
Shipping Service	Go	Shipping logistics
Product Catalog	Go	Product information database
Currency Service	Node.js	Currency conversion
Recommendation Service	Python	Product recommendations
Accounting Service	.NET	Financial tracking
Email Service	Python	Email notifications
Order Service	Java	Order management
Flag Service	Go	Feature flag management
Kafka	Message Queue	Event streaming

Technology Stack

Telemetry Collection:

- OpenTelemetry Collector
- OpenTelemetry SDKs (one per language)

Observability Backends (Default):

- Prometheus: Metrics storage and querying
- Grafana: Metrics visualization and dashboards
- Jaeger: Distributed tracing
- OpenSearch: Log aggregation and search

Additional Components:

- Locust: Load generation for realistic traffic
- Docker Compose: Container orchestration
- Kafka: Event streaming

Data Flow Architecture

Microservices
↓
OpenTelemetry Instrumentation
↓
OpenTelemetry Collector
↓
├→ Prometheus (Metrics)
├→ Jaeger (Traces)
├→ OpenSearch (Logs)
└→ Other Backends (Optional)
↓
Visualization & Alerting
├→ Grafana Dashboards
├→ Jaeger UI
├→ OpenSearch Dashboards
└→ Alert Rules

Prerequisites and Setup

System Requirements

Minimum Requirements:

- AWS EC2 instance with **t2.xlarge** (16GB RAM, 4 vCPU)
  - Kafka requires significant memory
  - Total data collection requires processing power
  - Free tier (t2.micro with 1GB RAM) is insufficient

Estimated Costs:

- t2.xlarge: ~$0.15/hour

- Storage: 15GB EBS at ~$1.50/month
- Data transfer: Minimal for local testing
- **Total**: ~$5-10 for 24-hour project

Required Tools

On Local Machine:

- SSH client (built-in on Linux/macOS, PuTTY on Windows)
- AWS account with EC2 permissions
- Terminal/Command prompt

On AWS EC2 Instance:

- Docker (container runtime)
- Docker Compose (container orchestration)
- Git (repository cloning)
- Ubuntu 22.04 LTS operating system

Knowledge Prerequisites

- Basic Linux command-line operations
- Understanding of microservices architecture
- Familiarity with containers and Docker
- Basic networking concepts (ports, protocols)
- No need for deep knowledge of individual languages

GitHub Repository

Official Repository:
https://github.com/open-telemetry/opentelemetry-demo

Documentation:
https://opentelemetry.io/docs/demo/

AWS Environment Configuration

Step 1: Launch EC2 Instance

Access AWS Console

1. Log in to AWS Management Console
2. Navigate to EC2 Dashboard
3. Click **Instances**
4. Click **Launch Instance**

Instance Configuration

Name: `otel-observability-project`

Operating System:

- Ubuntu 22.04 LTS (Amazon Machine Image)
- Free tier eligible, widely supported

Instance Type: t2.xlarge

- 16 GB RAM
- 4 vCPU
- Not eligible for free tier (charges apply)

Key Pair:

- Create new or use existing SSH key
- Save securely (cannot be recovered)
- Required for SSH access

Security Group Rules:

Inbound Rules:

- SSH (Port 22): From 0.0.0.0/0 (your IP recommended)
- HTTP (Port 80): From 0.0.0.0/0 (application)
- HTTP (Port 8080): From 0.0.0.0/0 (load generator, Jaeger, Grafana)
- Custom TCP (Port 4317): From 0.0.0.0/0 (OTel Collector GRPC)
- Custom TCP (Port 4318): From 0.0.0.0/0 (OTel Collector HTTP)

Outbound Rules:

- All traffic allowed (default)

Storage:

- EBS Volume: 15 GB gp3
- Sufficient for Docker images and logs

Step-by-Step Launch

Instance launches (takes 30-60 seconds)

Wait for status to change from "Pending" to "Running"

Wait for Status Checks to pass (2/2 checks)

Step 2: Connect to Instance

Option A: Using EC2 Instance Connect (Easiest)

1. Select instance in AWS Console
2. Click **Connect** button
3. Choose **EC2 Instance Connect** tab
4. Click **Connect** (opens browser terminal)

Option B: Using SSH from Terminal

After saving key pair

chmod 400 ~/Downloads/your-key-pair.pem

Connect via SSH

ssh -i ~/Downloads/your-key-pair.pem ubuntu@<PUBLIC_IP>

Example:

ssh -i ~/Downloads/your-key-pair.pem ubuntu@54.123.456.789

Step 3: Update System Packages

Run on EC2 instance

sudo apt update -y
sudo apt upgrade -y

This ensures all packages are current

Prevents version conflicts and security issues

Step 4: Install Docker and Docker Compose

Install Docker

sudo apt install -y docker.io docker-compose

Verify installation

docker --version
docker-compose --version

Add current user to docker group (optional, for convenience)

sudo usermod -aG docker ubuntu

Note: Docker Compose comes bundled with modern Docker installations as `docker compose` (newer) or separate `docker-compose` (older).

Application Deployment

Step 1: Clone OpenTelemetry Demo Repository

Navigate to home directory

cd ~

Clone the official repository

git clone https://github.com/open-telemetry/opentelemetry-demo.git

Navigate to project

cd opentelemetry-demo

List contents

ls -la

Key files:

- docker-compose.yml: Main orchestration file

- src/: Source code for all microservices

- docs/: Documentation and guides

Step 2: Understanding Docker Compose File

The docker-compose.yml file defines all services, networks, and configurations:

version: '3'

Global logging configuration

x-default-logging: &default-logging
driver: "json-file"
options:
max-size: "5m"
max-file: "2"

Network definition for service communication

networks:
default:
name: opentelemetry-demo
driver: bridge

Service definitions (18+ services)

services:

Example: Accounting Service (.NET)

accountingservice:
image: ghcr.io/open-telemetry/demo:latest-accountingservice
container_name: accountingservice
depends_on:
otelcol:
condition: service_started
environment:
- OTEL_EXPORTER_OTLP_ENDPOINT=http://otelcol:4317
- OTEL_RESOURCE_ATTRIBUTES=service.name=accountingservice
ports:
- "8200:8200"
logging: *default-logging

Example: Frontend (JavaScript/Node.js)

frontend:
image: ghcr.io/open-telemetry/demo:latest-frontend
container_name: frontend
ports:
- "3000:3000"
environment:
- PUBLIC_OTEL_EXPORTER_OTLP_TRACES_ENDPOINT=http://localhost:4318/v1/traces
depends_on:
- frontendproxy
logging: *default-logging

OpenTelemetry Collector

otelcol:
image: otel/opentelemetry-collector-k8s:latest
container_name: otelcol
command: ["--config=/etc/otel-collector-config.yml"]
volumes:
- ./src/otelcollector/otel-collector-config.yml:/etc/otel-collector-config.yml
ports:
- "4317:4317" # OTLP GRPC receiver
- "4318:4318" # OTLP HTTP receiver
- "9411:9411" # Zipkin receiver
environment:
- GOGC=80
depends_on:
- jaeger
- prometheus
logging: *default-logging

Prometheus for metrics

prometheus:
image: prom/prometheus:latest
container_name: prometheus
command:
- "--config.file=/etc/prometheus/prometheus.yml"
- "--storage.tsdb.path=/prometheus"
volumes:
- ./src/prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
ports:
- "9090:9090"
logging: *default-logging

Grafana for visualization

grafana:
image: grafana/grafana:latest
container_name: grafana
environment:
- GF_AUTH_ANONYMOUS_ENABLED=true
- GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
volumes:
- ./src/grafana/provisioning:/etc/grafana/provisioning
- ./src/grafana/dashboards:/var/lib/grafana/dashboards
ports:
- "3001:3000"
depends_on:
- prometheus
logging: *default-logging

Jaeger for tracing

jaeger:
image: jaegertracing/all-in-one:latest
container_name: jaeger
ports:
- "16686:16686" # Jaeger UI
- "4317:4317" # OTLP GRPC receiver
- "9411:9411" # Zipkin receiver
logging: *default-logging

Kafka for event streaming

kafka:
image: confluentinc/cp-kafka:7.5.0
container_name: kafka
depends_on:
- zookeeper
environment:
KAFKA_BROKER_ID: 1
KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
logging: *default-logging

Zookeeper for Kafka coordination

zookeeper:
image: confluentinc/cp-zookeeper:7.5.0
container_name: zookeeper
environment:
ZOOKEEPER_CLIENT_PORT: 2181
logging: *default-logging

... (additional services defined similarly)

Key Concepts:

Section	Purpose
version	Docker Compose file format
x-default-logging	Logging template reused by services
networks	Define service communication
services	Container definitions
image	Docker image to use
container_name	Hostname within network
ports	Port mappings: HOST:CONTAINER
environment	Environment variables
depends_on	Service startup order
volumes	Mount configurations
logging	Log collection settings

Step 3: Deploy Application

Ensure you're in the project directory

cd ~/opentelemetry-demo

Start all services

sudo docker-compose up -d

The -d flag runs in background

Docker Compose will:

1. Pull all images from registry

2. Create network: opentelemetry-demo

3. Start all 18+ services in order

4. Wait for health checks

This takes 2-5 minutes on first run

Step 4: Verify Deployment

Check running containers

sudo docker ps

Expected output: 18-20 containers running

NAME STATUS PORTS

accountingservice Up 2 minutes 8200/tcp

adservice Up 2 minutes 9555/tcp

cartserviceUp 2 minutes 7070/tcp

...

Check logs for specific service

sudo docker logs <service_name>

Example: Check Collector logs

sudo docker logs otelcol

Check for errors

sudo docker logs otelcol | grep -i error

View all logs

sudo docker-compose logs -f

Exit with Ctrl+C

Step 5: Access Application Endpoints

Once deployment succeeds, access applications via instance public IP:

Core Endpoints:

Service	URL	Purpose
Frontend	http://<IP>:80	E-commerce store frontend
Grafana	http://<IP>:3001/grafana	Metrics dashboards
Jaeger	http://<IP>:16686	Distributed tracing UI
Prometheus	http://<IP>:9090	Metrics database
Load Generator	http://<IP>:8089/	Generate synthetic traffic
Kafka	kafka:9092	Event broker (internal)
OpenSearch	http://<IP>:9200	Log storage (internal)

Finding Public IP:

Option 1: From AWS Console

Click instance → Note IPv4 Public IP

Option 2: From terminal

curl http://169.254.169.254/latest/meta-data/public-ipv4

Test Connectivity:

From your local machine

curl http://<PUBLIC_IP>:80

Should return HTML from frontend

If connection refused, wait 30 seconds and retry

Monitoring and Observability

Accessing Grafana Dashboards

Login and Navigation

1. Open browser: http://<PUBLIC_IP>:3001
2. Default credentials (no login required if anonymous enabled):
   - Username: `admin`
   - Password: `admin`

Available Dashboards

1. Demo Dashboard (Main)

- Overview of all microservices
- Request rates and latency
- Error rates per service
- Resource utilization

2. Span Metrics Demo Dashboard

- Detailed span metrics
- Service-to-service latency
- Error analysis by operation
- Top 10 services by metrics

3. OpenTelemetry Collector Dashboard

- Collector health status
- Data processing rates
- Receiver/processor/exporter metrics
- Memory and CPU usage

4. OpenSearch Logs Dashboard

- Log aggregation and search
- Error tracking
- Performance warnings
- Structured log viewing

Example: Viewing Service Metrics

1. Click on "Demo Dashboard"
2. Select service from dropdown (e.g., "adservice")
3. View metrics:
   - Latency: P50, P95, P99
   - Error Rate: Percentage of failed requests

    - Request Rate: Requests per second
    - Resource Usage: CPU, Memory
4. Time range selector: Last 5 min, 15 min, 1 hour, etc.

Understanding Jaeger Traces

Jaeger UI Navigation

1. Open browser: `http://<PUBLIC_IP>:16686`
2. Jaeger UI for distributed tracing

Trace Analysis

Search for Traces:

1. Service: Select microservice (e.g., "frontend", "adservice")
2. Operation: Select specific operation (get, post, etc.)
3. Tags: Filter by specific attributes (e.g., error=true)
4. Lookback: Time range to search (5 min, 1 hour, etc.)
5. Limit Results: Maximum traces to return
6. Click "Find Traces"

Interpreting Trace Results:

```
Request ID: 4bf92f3577b34da6a3ce929d0e0e4736
Duration: 250ms
Status: ERROR (indicated by red dot)

Span Breakdown:
├── Frontend Proxy (0-50ms)
├── Frontend (50-100ms)
├── Ad Service (100-250ms) ← ERROR HERE
└── Ad Service Internal Logic
└── Database Query (timeout)
```

Debugging with Traces:

1. **Identify Error Location**: Red spans indicate failures
2. **Check Error Message**: Click span → view logs and error details
3. **Analyze Dependencies**: See service call chain
4. **Measure Latency**: Find bottleneck services
5. **Compare Traces**: Use trace comparison tool for A/B analysis

Trace Data Elements

Element	Description
Trace ID	Unique identifier for complete request
Span ID	Unique identifier for service call
Parent Span ID	Links child spans to parent
Duration	Time taken for operation
Tags	Key-value metadata
Logs	Timestamped events within span
Status	Success, error, or unset

Prometheus Metrics Database

Accessing Prometheus

1. Open browser: `http://<PUBLIC_IP>:9090`
2. Query interface for time-series metrics

Example Queries

Request Rate (requests per second):
```
rate(http_server_request_duration_seconds_count[5m])
```

Error Rate:
```
rate(http_server_request_duration_seconds_count{status="5xx"}[5m])
/
rate(http_server_request_duration_seconds_count[5m])
```

P99 Latency:
```
histogram_quantile(0.99, rate(http_server_request_duration_seconds_bucket[5m]))
```

Service CPU Usage:
```
container_cpu_usage_seconds_total{service="adservice"}
```

Metrics Export

Download metrics in various formats:

- Prometheus text format
- JSON
- CSV

Feature Flags and Error Injection

Purpose of Feature Flags

Feature flags enable controlled error injection and scenario testing without code changes:

- Test observability tool accuracy
- Simulate production issues in controlled manner
- Validate alerting rules
- Train team on incident response

Flag Configuration File

Location:
./src/flagD/demo.flagd.json

File Structure:

```
{
"$schema": "https://flagd.dev/schema/v0/flags.json",
"flags": {
"productCatalogFailure": {
"state": "ENABLED",
"description": "Fail product catalog service on specific product",
"variants": {
"on": true,
"off": false
},
"defaultVariant": "off"
},
"recommendationServiceFailure": {
"state": "ENABLED",
"description": "Make recommendation service fail",
"variants": {
"on": true,
"off": false
},
"defaultVariant": "off"
},
"cartServiceFailure": {
"state": "ENABLED",
"description": "Make cart service fail",
"variants": {
"on": true,
"off": false
},
"defaultVariant": "off"
},
"adServiceHighCPU": {
"state": "ENABLED",
"description": "Increase CPU usage for ad service",
"variants": {
```

"on": true,
"off": false
},
"defaultVariant": "off"
}
}
}

Enabling Feature Flags

Edit Configuration File

SSH into instance

ssh -i your-key.pem ubuntu@<PUBLIC_IP>

Navigate to demo directory

cd ~/opentelemetry-demo

Edit flagD configuration

sudo nano src/flagD/demo.flagd.json

Make Changes

Change `"defaultVariant"` from `"off"` to `"on"` for desired flags:

{
"adServiceHighCPU": {
"defaultVariant": "on" // Changed from "off"
},
"cartServiceFailure": {
"defaultVariant": "on" // Changed from "off"
}
}

Save file: `Ctrl+S`, then `Ctrl+X`

Restart Services

Stop all services

sudo docker-compose down

Start services (with new flag configuration)

sudo docker-compose up -d

Wait for all services to be ready (~2 minutes)

sudo docker ps | grep -c "Up"

Should show 18-20 containers running

Monitoring Error Injection

Generate Traffic

1. Open Load Generator: http://<PUBLIC_IP>:8089
2. Locust interface appears
3. Start load test:
   - Number of users: 10
   - Spawn rate: 2 users/second
   - Target host: `http://frontend:80`
   - Click "Start swarming"

View Errors in Grafana

1. Open Grafana: http://<PUBLIC_IP>:3001
2. Select "Demo Dashboard"
3. Change time range to "Last 5 minutes"
4. Select service with enabled flags (e.g., "adservice" for High CPU)
5. Observe:
   - Error rate increases
   - Latency increases
   - CPU usage increases
   - Log entries appear

Find Errors in Jaeger

1. Open Jaeger: http://<PUBLIC_IP>:16686
2. Service: "adservice" (if CPU flag enabled)
3. Tags: "error=true" (filter to errors only)
4. Click "Find Traces"
5. Examine error traces:
   - Error message
   - Stack trace
   - Span timing
   - Related services

View Logs in OpenSearch

1. Open OpenSearch: http://<PUBLIC_IP>:9200/_dashboards
2. Create index pattern: "logs-*"
3. Discover tab
4. Filter: "service.name = adservice"
5. View error logs with context

Advanced Configuration

Switching Monitoring Backends

OpenTelemetry's vendor-agnostic nature allows using different backends:

Supported Backends

Backend	Traces	Metrics	Logs	Protocol
Jaeger	✓	-	-	OTLP
Prometheus	-	✓	-	OTLP
Datadog	✓	✓	✓	OTLP API
New Relic	✓	✓	✓	OTLP API
Splunk	✓	✓	✓	OTLP
Dynatrace	✓	✓	✓	OTLP
Elastic Stack	✓	✓	✓	OTLP

Configuration File

Location: `./src/otelcollector/otel-collector-config.yml`

This YAML file defines receivers, processors, and exporters.

Default Configuration (Jaeger/Prometheus):

```
receivers:
otlp:
protocols:
grpc:
endpoint: 0.0.0.0:4317
http:
endpoint: 0.0.0.0:4318

processors:
batch:
send_batch_size: 1024
timeout: 5s
```

memory_limiter:
check_interval: 1s
limit_mib: 512
spike_limit_mib: 128

exporters:

Jaeger for traces

jaeger:
endpoint: jaeger:14250
tls:
insecure: true

Prometheus for metrics

prometheus:
endpoint: "0.0.0.0:8889"

Logging for debugging

logging:
loglevel: debug

extensions:
health_check:
endpoint: ":13133"

service:
extensions: [health_check]
pipelines:
traces:
receivers: [otlp]
processors: [batch, memory_limiter]
exporters: [jaeger, logging]

```
metrics:
  receivers: [otlp, prometheus]
  processors: [batch, memory_limiter]
  exporters: [prometheus, logging]
```

Switching to Datadog (Example)

Prerequisites:

- Datadog account with API key
- Collector needs network access to Datadog endpoints

Configuration Changes:

exporters:
datadog:
api:
key: "${DD_API_KEY}" # Set as environment variable
site: "datadoghq.com" # Or eu.datadoghq.com for EU
host_metadata:
enabled: true
hostname_source: "config_or_system"

Keep existing exporters commented

jaeger:

endpoint: jaeger:14250

service:
pipelines:
traces:
receivers: [otlp]
processors: [batch, memory_limiter]
exporters: [datadog] # Changed from jaeger

```
metrics:
  receivers: [otlp, prometheus]
  processors: [batch, memory_limiter]
  exporters: [datadog]  # Changed from prometheus
```

Apply Changes:

Set Datadog API key

export DD_API_KEY="your-api-key-here"

Restart collector

sudo docker-compose restart otelcol

Verify connection

sudo docker logs otelcol | grep "datadog"

Resource Decorators

Add metadata to all telemetry data for better organization:

processors:
resource:
attributes:
- key: environment
value: production
action: upsert
- key: region
value: us-east-1
action: upsert
- key: team
value: platform-engineering
action: insert
- key: version
value: "1.0.0"
action: upsert

service:
pipelines:
traces:
receivers: [otlp]
processors: [resource, batch]
exporters: [jaeger]

Sampling Strategies

Reduce data volume while retaining important traces[5]:

Head-Based Sampling

Sample at trace start (simple but less intelligent):

processors:
probabilistic_sampler:
sampling_percentage: 10 # Keep 10% of traces

service:
pipelines:
traces:
receivers: [otlp]

processors: [probabilistic_sampler, batch]
exporters: [jaeger]

Tail-Based Sampling

Sample after trace completes (requires buffering):

processors:
tail_sampling:
policies:
Always sample error traces
- name: error_traces
type: status_code
status_code:
status_codes: [ERROR]

```
# Sample traces with high latency
- name: high_latency
  type: latency
  latency:
    threshold_ms: 5000

# Probabilistic fallback
- name: probabilistic_fallback
  type: probabilistic
  probabilistic:
    sampling_percentage: 5
```

service:
pipelines:
traces:
receivers: [otlp]
processors: [tail_sampling, batch]
exporters: [jaeger]

Memory Management

Prevent resource exhaustion in the collector:

processors:
memory_limiter:
check_interval: 1s # Check every 1 second
limit_mib: 512 # Max 512MB of RAM
spike_limit_mib: 128 # Allow 128MB spike before dropping

batch:
send_batch_size: 1024 # Send after collecting 1024 spans

timeout: 5s # Or after 5 seconds
send_batch_max_size: 2048

service:
pipelines:
traces:
receivers: [otlp]
processors: [memory_limiter, batch]
exporters: [jaeger]

Troubleshooting Guide

Common Issues and Solutions

1. Kafka Service Failing

Symptom: Kafka container exits immediately

Cause: Insufficient RAM

Solutions:

Check available RAM

free -h

Reduce services using docker-compose override

cat > docker-compose.override.yml << EOF
version: '3'
services:
kafka:
environment:
KAFKA_HEAP_OPTS: "-Xms256M -Xmx256M" # Reduce memory
EOF

Restart services

sudo docker-compose restart kafka

2. Port Already in Use

Symptom: `Error: bind: address already in use`

Cause: Port 80, 8080, 4317, etc. already in use

Solution:

Find process using port 8080

sudo lsof -i :8080

Kill the process

sudo kill -9 <PID>

Or use different port in docker-compose

docker-compose up -d -p 8081:8080

3. Services Not Communicating

Symptom: "Connection refused" errors between services

Cause: Services not on same Docker network or misconfigured endpoints

Solutions:

Check network

sudo docker network ls | grep opentelemetry

Inspect network

sudo docker network inspect opentelemetry-demo

Verify all services on same network

sudo docker inspect <service_name> | grep NetworkSettings

Check service endpoint configuration

sudo docker exec <service_name> env | grep OTEL

4. No Data in Grafana

Symptom: Dashboards show "no data"

Cause: Collector not receiving or exporting data

Solutions:

Check collector logs

sudo docker logs otelcol | tail -50

Check for errors

sudo docker logs otelcol | grep -i error

Verify data flow

sudo docker exec otelcol curl -s http://localhost:8889/metrics | head -20

Check receiver connectivity

sudo docker logs frontend | grep otelcol

5. High Memory/CPU Usage

Symptom: Instance becomes slow or unresponsive

Causes: Excessive data collection, sampling not configured

Solutions:

Monitor resource usage

watch -n 1 'free -h; echo "---"; top -bn1 | head -15'

Enable sampling

nano src/otelcollector/otel-collector-config.yml

Add probabilistic sampler (see Advanced Configuration)

Restart services

sudo docker-compose restart otelcol

Debugging Commands

View all logs

sudo docker-compose logs -f

View specific service logs

sudo docker logs -f <service_name>

Get into container

sudo docker exec -it <service_name> /bin/bash

Check metrics endpoint

curl http://localhost:8889/metrics

Check Prometheus scrape status

curl http://localhost:9090/api/v1/targets

Verify Jaeger is receiving traces

curl http://localhost:16686/api/services

Check health of all services

sudo docker-compose ps

Cleanup and Restart

Stop all services

sudo docker-compose down

Remove volumes and data

sudo docker-compose down -v

Clean up images (careful, removes all local images)

sudo docker image prune -a

Full reset

sudo docker system prune -a --volumes

Restart fresh

sudo docker-compose up -d

Best Practices

Implementation Best Practices

1. Semantic Conventions

Follow OpenTelemetry semantic conventions for consistency[6]:

Service Names:
service.name: "payment-service"
service.namespace: "production"
service.version: "1.0.0"

HTTP Attributes:
http.method: "POST"
http.url: "https://api.example.com/v1/payments"
http.status_code: 200
http.response_content_length: 1024

Database Operations:
db.system: "postgresql"
db.name: "orders_db"
db.operation: "SELECT"
db.statement: "SELECT * FROM orders WHERE id = ?"

2. Attribute Management

Minimize Cardinality[7]:

✓ GOOD:
environment: "production" (low cardinality)
http.method: "GET" (fixed values)
service.name: "api-service" (bounded)

✗ PROBLEMATIC:
user.id: "12345678" (high cardinality → millions of values)

request.id: "uuid-..." (unbounded cardinality)
customer.email: "user@example.com" (unbounded)

Resource Attributes:

processors:
resource:
attributes:
- key: environment
value: production
- key: deployment
value: kubernetes
- key: version
value: "1.2.3"

3. Sensitive Data Handling

Never collect:

- Personal Identifiable Information (PII)
- Credit card numbers
- API keys or secrets
- Passwords

Redaction Strategy:

processors:
attributes:
actions:
Drop sensitive attributes
- key: password
action: delete
- key: credit_card
action: delete
- key: api_key
action: delete

```
# Hash user IDs for cardinality
- key: user.id
  action: hash_sha256
```

4. Instrumentation Strategy

Prioritize Critical Components:

1. **API Endpoints** (entry points to system)
2. **Database Operations** (performance bottlenecks)
3. **Message Queues** (async processing)
4. **External Service Calls** (dependencies)
5. **Business Operations** (revenue impact)

Avoid Excessive Detail:

- Don't instrument every function call
- Focus on cross-service boundaries
- Use sampling for verbose operations

5. Alert Configuration

Create Alerts Based on SLOs:

Grafana Alert: Error Rate > 5%

expr: |
(rate(http_requests_total{status="5xx"}[5m]) /
rate(http_requests_total[5m])) > 0.05
notification: on-call-team

Grafana Alert: P99 Latency > 1 second

expr: |
histogram_quantile(0.99, rate(http_duration_seconds_bucket[5m])) > 1
notification: dev-team

Operational Best Practices

1. Centralized Configuration

Use ConfigMaps in Kubernetes

kubectl create configmap otel-config --from-file=config.yaml

Or environment variables for Docker

docker run -e OTEL_CONFIG_YAML="$(cat config.yaml)" ...

2. Health Checks

extensions:
health_check:
endpoint: ":13133"

Kubernetes liveness probe

livenessProbe:
httpGet:
path: /healthz
port: 13133
initialDelaySeconds: 5
periodSeconds: 10

3. Rate Limiting and Backpressure

processors:
batch:
send_batch_size: 1024
timeout: 5s

exporters:
otlp:
endpoint: backend:4317
sending_queue:
queue_size: 5000 # Buffer up to 5000 items
storage: memory # Use memory storage
retry_on_failure:
enabled: true
initial_interval: 5s
max_interval: 30s

4. Testing and Validation

Test Observability:

Generate test data

docker run -it grafana/loki-canary:latest
--url=http://localhost:3100

Test metrics collection

curl -X POST http://localhost:4318/v1/metrics
-d @test_metrics.json

Validate trace collection

curl -X POST http://localhost:4318/v1/traces
-d @test_traces.json

5. Documentation and Runbooks

Create Incident Runbooks:

Scenario	Runbook
High Error Rate	Check error logs → Review traces → Check deployments
High Latency	Check CPU/Memory → Review database queries → Check network
Service Unavailable	Check logs → Verify network connectivity → Restart service
Data Gaps	Check collector health → Verify receiver configuration

---

Conclusion

This OpenTelemetry observability project demonstrates enterprise-grade monitoring capabilities using industry-standard tools and practices. By completing this guide, you have:

Achievements

✓ Deployed a production-grade microservice application
✓ Understood observability's three pillars: metrics, logs, and traces
✓ Configured OpenTelemetry Collector for telemetry collection
✓ Integrated multiple observability backends (Prometheus, Grafana, Jaeger)
✓ Performed root cause analysis using distributed traces
✓ Implemented error injection and scenario testing
✓ Mastered advanced configuration and troubleshooting

Career Impact

This project significantly enhances your DevOps and Site Reliability Engineering (SRE) credentials:

- **Portfolio Piece**: Add to GitHub and resume
- **Interview Preparation**: Demonstrates hands-on observability expertise
- **Promotion Candidate**: Shows advanced infrastructure knowledge
- **Team Value**: Become observability expert for your organization

Next Steps

1. **Document Your Learning**: Write technical blog post on your experience
2. **Share on LinkedIn**: Tag @Cloud Champ and @Open Telemetry
3. **Customize the Demo**: Integrate with your preferred observability backend
4. **Scale to Production**: Adapt patterns to your organization's environment
5. **Contribute**: Submit improvements to the open-source project

---

References

[1] OpenTelemetry Project. (2024). OpenTelemetry Documentation. Retrieved from https://opentelemetry.io/

[2] Cloud Champ. (2024, August 22). Observability DevOps Project - OpenTelemetry Demo Application [Video]. YouTube.

[3] CrowdStrike. (2024, October). The Three Pillars of Observability: Logs, Metrics, and Traces. Retrieved from https://www.crowdstrike.com/

[4] OpenTelemetry. (2024). Collector Configuration. Retrieved from https://opentelemetry.io/docs/collector/configuration/

[5] Better Stack. (2025, March 2). Essential OpenTelemetry Best Practices for Robust Observability. Retrieved from https://betterstack.com/

[6] OpenTelemetry. (2024). Semantic Conventions. Retrieved from https://opentelemetry.io/docs/specs/semconv/

[7] Grafana. (2024, November). OpenTelemetry Best Practices. Retrieved from https://grafana.com/blog/2023/12/18/

Document Version: 1.0
Last Updated: December 4, 2024
Author: DevOps Engineering Team
Status: Complete and Ready for Implementation