Highlighting OpenTelemetry: Standardized Telemetry Collection

OpenTelemetry is a vendor-neutral set of open-source tools, APIs, and SDKs that standardize the collection and export of telemetry data – metrics, logs, and traces – from your software applications. In your data platform, OpenTelemetry is pivotal for achieving deep, consistent observability across all services, regardless of their underlying technology. It enables unified monitoring, tracing, and logging, essential for understanding system behavior and troubleshooting issues.
This guide will demonstrate basic and advanced use cases of OpenTelemetry, leveraging your **Advanced Track** local environment setup and its integration with Grafana Alloy and Grafana.
Reference: This guide builds upon the concepts and setup described in **Section 4.2. Core Technology Deep Dive** of the **Core Handbook** and the **Progressive Path Setup Guide Deep-Dive Addendum**, specifically emphasizing OpenTelemetry's role in the **Observability** section and the "Highlighting Grafana Alloy" document.

Basic Use Case: Instrumenting an Application for Metrics Collection

Objective: To demonstrate how to instrument a Python application (FastAPI) to emit application-specific metrics using OpenTelemetry, and how these metrics are then collected by Grafana Alloy and visualized in Grafana.
Role in Platform: Enable collection of custom, application-level metrics (e.g., request counts, latency, business-specific events) from services, providing granular insights beyond basic infrastructure metrics.
Setup/Configuration (Local Environment - Advanced Track):

1. **Ensure all Advanced Track services are running:** docker compose up --build -d from your project root. This includes fastapi_ingestor, grafana-alloy, and grafana.
2. **Install OpenTelemetry Python SDK and Exporters:** Your fastapi_app/requirements.txt should include necessary OpenTelemetry packages.
   ```
   # fastapi_app/requirements.txt
   fastapi
   uvicorn
   python-dotenv
   kafka-python
   pydantic
   # OpenTelemetry packages
   opentelemetry-api
   opentelemetry-sdk
   opentelemetry-exporter-otlp
   ```

opentelemetry-instrumentation-fastapi
opentelemetry-instrumentation-requests
opentelemetry-sdk-extension-aws
opentelemetry-distro
opentelemetry-instrumentation-logging # For logs
opentelemetry-sdk-metrics # For custom metrics

3. **Instrument FastAPI application:** Modify fastapi_app/app/main.py to initialize OpenTelemetry and configure it to export metrics to Grafana Alloy.
   Example fastapi_app/app/main.py (conceptual additions):

```
# fastapi_app/app/main.py
import os
import json
from datetime import datetime
from typing import Optional

from fastapi import FastAPI, HTTPException, status
from pydantic import BaseModel, Field
from kafka import KafkaProducer

# --- OpenTelemetry Imports and Setup ---
from opentelemetry import metrics
from opentelemetry import trace
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.metrics import MeterProvider
from opentelemetry.sdk.metrics.export import (
    ConsoleMetricExporter,
    PeriodicExportingMetricReader,
)
from opentelemetry.exporter.otlp.proto.http.trace_exporter import OTLPSpanExporter
from opentelemetry.exporter.otlp.proto.http.metric_exporter import OTLPMetricExporter
from opentelemetry.instrumentation.fastapi import FastAPIInstrumentor
from opentelemetry.instrumentation.logging import LoggingInstrumentor # For logs


# Resource for identifying the service
resource = Resource.create({
    "service.name": "fastapi-ingestor",
    "service.version": "1.0.0",
    "env.type": "local-dev"
})
```

```
# Configure OTLP Exporter endpoint (Grafana Alloy)
# This should match the otelcol.receiver.otlp config in alloy-config.river
OTEL_EXPORTER_OTLP_ENDPOINT = os.getenv("OTEL_EXPORTER_OTLP_ENDPOINT",
"http://grafana-alloy:4318")

# Metrics Provider (for custom metrics)
metric_reader = PeriodicExportingMetricReader(
    OTLPMetricExporter(endpoint=f"{OTEL_EXPORTER_OTLP_ENDPOINT}/v1/metrics")
)
meter_provider = MeterProvider(resource=resource, metric_readers=[metric_reader])
metrics.set_meter_provider(meter_provider)
meter = metrics.get_meter("fastapi.ingestion.app")

# Create counters for business metrics
financial_tx_counter = meter.create_counter(
    "financial.transactions.ingested",
    description="Number of financial transactions ingested",
    unit="1"
)
insurance_claim_counter = meter.create_counter(
    "insurance.claims.ingested",
    description="Number of insurance claims ingested",
    unit="1"
)

# Tracer Provider (for distributed tracing)
trace_exporter =
OTLPSpanExporter(endpoint=f"{OTEL_EXPORTER_OTLP_ENDPOINT}/v1/traces")
trace.set_tracer_provider(TracerProvider(resource=resource))
trace.get_tracer_provider().add_span_processor(
    BatchSpanProcessor(trace_exporter)
)

# Instrument logging to include trace/span IDs
LoggingInstrumentor().instrument(set_logging_format=True)

# --- Pydantic Models and FastAPI App Init (as before) ---
class FinancialTransaction(BaseModel):
    transaction_id: str = Field(..., example="FT-20231026-001")
    timestamp: datetime = Field(..., example="2023-10-26T14:30:00Z")
    account_id: str = Field(..., example="ACC-001")
    amount: float = Field(..., gt=0, example=150.75)
    currency: str = Field(..., max_length=3, example="USD")
```

```
    transaction_type: str = Field(..., example="debit")
    merchant_id: Optional[str] = Field(None, example="MER-XYZ")
    category: Optional[str] = Field(None, example="groceries")

class InsuranceClaim(BaseModel):
    claim_id: str = Field(..., example="IC-20231026-001")
    timestamp: datetime = Field(..., example="2023-10-26T15:00:00Z")
    policy_number: str = Field(..., example="POL-987654")
    claim_amount: float = Field(..., gt=0, example=1000.00)
    claim_type: str = Field(..., example="auto")
    claim_status: str = Field(..., example="submitted")
    customer_id: str = Field(..., example="CUST-ABC")
    incident_date: datetime = Field(..., example="2023-09-15T08:00:00Z")

app = FastAPI(
    title="Financial/Insurance Data Ingestor API",
    description="API for ingesting various financial and insurance data into the data
platform.",
    version="1.0.0",
)

# Instrument FastAPI application with OpenTelemetry
FastAPIInstrumentor.instrument_app(app)

# --- Kafka Producer Setup (as before) ---
KAFKA_BROKER = os.getenv("KAFKA_BROKER", "kafka:29092")
KAFKA_TOPIC_FINANCIAL = os.getenv("KAFKA_TOPIC_FINANCIAL",
"raw_financial_transactions")
KAFKA_TOPIC_INSURANCE = os.getenv("KAFKA_TOPIC_INSURANCE",
"raw_insurance_claims")

try:
    producer = KafkaProducer(
        bootstrap_servers=[KAFKA_BROKER],
        value_serializer=lambda v: json.dumps(v).encode('utf-8'),
        retries=5,
        linger_ms=100,
        batch_size=16384
    )
    print(f"Kafka Producer initialized for broker: {KAFKA_BROKER}")
except Exception as e:
    print(f"Error initializing Kafka Producer: {e}")
    producer = None
```

```
# --- API Endpoints ---
@app.get("/health", tags=["Monitoring"])
async def health_check():
    return {"status": "healthy", "message": "Welcome to Financial/Insurance Data Ingestor API!"}

@app.post("/ingest-financial-transaction/", status_code=status.HTTP_200_OK, tags=["Ingestion"])
async def ingest_financial_transaction(transaction: FinancialTransaction):
    try:
        if producer:
            producer.send(KAFKA_TOPIC_FINANCIAL, transaction.dict())
            print(f"Financial transaction ingested and sent to Kafka topic '{KAFKA_TOPIC_FINANCIAL}': {transaction.transaction_id}")
        else:
            print("Kafka producer not available. Skipping send.")

        # Increment custom metric
        financial_tx_counter.add(1, {"transaction.type": transaction.transaction_type, "currency": transaction.currency})

        return {"message": "Financial transaction ingested successfully", "transaction_id": transaction.transaction_id}
    except Exception as e:
        raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR, detail=f"Failed to ingest transaction: {e}")

@app.post("/ingest-insurance-claim/", status_code=status.HTTP_200_OK, tags=["Ingestion"])
async def ingest_insurance_claim(claim: InsuranceClaim):
    try:
        if producer:
            producer.send(KAFKA_TOPIC_INSURANCE, claim.dict())
            print(f"Insurance claim ingested and sent to Kafka topic '{KAFKA_TOPIC_INSURANCE}': {claim.claim_id}")
        else:
            print("Kafka producer not available. Skipping send.")

        # Increment custom metric
        insurance_claim_counter.add(1, {"claim.type": claim.claim_type, "claim.status": claim.claim_status})
```

```
        return {"message": "Insurance claim ingested successfully", "claim_id": claim.claim_id}
    except Exception as e:
        raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR, detail=f"Failed to ingest claim: {e}")
```

Note: You'll need to add opentelemetry-sdk-metrics to your fastapi_app/requirements.txt to run this. Also, for BatchSpanProcessor you'd need to from opentelemetry.sdk.trace.export import BatchSpanProcessor.

4. **Configure Grafana Alloy:** Ensure observability/alloy-config.river has an otelcol.receiver.otlp component to receive metrics, traces, and logs from FastAPI, forwarding them to Grafana (acting as a Prometheus remote write endpoint).
   Example observability/alloy-config.river (relevant snippet for OTLP receiver):

```
# observability/alloy-config.river
# ...
prometheus.remote_write "default" {
  url = "http://grafana:9090/api/prom/push"
}

otelcol.receiver.otlp "default" {
  http { } # Listen for OTLP HTTP
  grpc { } # Listen for OTLP gRPC
  output {
    metrics = [prometheus.remote_write.default.receiver]
    # Traces and logs would go to other exporters/receivers if configured
    traces = [] # For this basic case, we might not forward traces/logs yet
    logs = []
  }
}
# ...
```

Steps to Exercise:

1. **Rebuild and Restart FastAPI and Grafana Alloy:**
   docker compose up --build -d fastapi_ingestor grafana-alloy

2. **Generate data:** Run python3 simulate_data.py. This will send requests to the FastAPI ingestor.
3. **Access Grafana:** Go to http://localhost:3000.
4. **Query Custom Metrics:**
   - Open the "Explore" view and select your Prometheus data source.
   - Enter PromQL queries for the new custom metrics:
     - financial_transactions_ingested_total
     - insurance_claims_ingested_total

    - You can also filter by attributes:
        - financial_transactions_ingested_total{transaction_type="purchase"}
        - insurance_claims_ingested_total{claim_type="auto"}

Verification:

- **Grafana:** The custom metrics (financial_transactions_ingested_total, insurance_claims_ingested_total) will appear in Grafana, and their values will increase as simulate_data.py sends data. The attributes (e.g., transaction_type, claim_type) will also be visible as labels, demonstrating successful OpenTelemetry instrumentation for metrics.

Advanced Use Case 1: Distributed Tracing for End-to-End Latency

Objective: To demonstrate how OpenTelemetry automatically propagates trace context across service boundaries (e.g., FastAPI calling Kafka, and conceptually, Kafka triggering a Spark job), allowing for end-to-end tracing and bottleneck identification.
Role in Platform: Provide deep visibility into the entire data flow path, helping to pinpoint latency bottlenecks and error origins across microservices and distributed processing stages.
Setup/Configuration:

1. **Ensure Basic Use Case setup is complete** (FastAPI is instrumented for traces as shown in the basic setup's main.py).
2. **Ensure Grafana Alloy is configured to forward traces:**
    - You'll need a trace backend. For local setup, we can conceptually demonstrate by configuring Alloy to forward to a dummy endpoint or a simple local Jaeger instance (if you have one). If you don't have a Jaeger instance, we'll confirm trace *export* from FastAPI.

Example observability/alloy-config.river (additions for traces):
```
# ...
# 1. Define an OTLP receiver for traces
otelcol.receiver.otlp "default" { # This receiver is already configured in basic setup
  http { }
  grpc { }
  output {
    metrics = [prometheus.remote_write.default.receiver]
    traces = [otelcol.exporter.otlp.jaeger_mock.input] # Forward traces to a specific exporter
    logs = []
  }
}

# 2. Define an OTLP exporter for traces (to a conceptual Jaeger/Tempo endpoint or just logs)
# For a real Jaeger, uncomment this in docker-compose.yml and add here:
# otelcol.exporter.otlp "jaeger_mock" {
#   client {
#     endpoint = "http://jaeger-all-in-one:4318" # Conceptual Jaeger/Tempo endpoint
#   }
# }
```

```
# OR, for a simple local demo without Jaeger UI, you can send to console
otelcol.exporter.logging "trace_logger" {
  log_level = "debug"
  output {
    traces = [otelcol.exporter.logging.trace_logger.input]
  }
}
# Then change otelcol.receiver.otlp -> traces = [otelcol.exporter.logging.trace_logger.input]
# ...
```

Note: To fully visualize traces, you'd need a Jaeger or Tempo UI. For this local demo, we'll primarily observe that traces are being sent by FastAPI and received/forwarded by Grafana Alloy.

Steps to Exercise:

1. **Ensure FastAPI is configured to export traces** (as shown in basic setup).
2. **Ensure Grafana Alloy is configured to receive and forward traces.**
3. **Restart affected services:** docker compose up --build -d fastapi_ingestor grafana-alloy.
4. **Generate API calls:** Run python3 simulate_data.py.
5. **Observe Grafana Alloy Logs:**
   docker compose logs -f grafana-alloy

   You should see messages indicating that Grafana Alloy is receiving OTLP trace data (spans) from fastapi-ingestor.
   If you configured otelcol.exporter.logging for traces, you'll see detailed JSON representations of the spans in Alloy's logs.
6. **Conceptual Trace Visualization (if Jaeger is integrated):**
   - If you had Jaeger running (http://localhost:16686), you would search for traces related to fastapi-ingestor service.
   - **Expected:** You would see individual traces, each representing an API request, with spans showing the duration of the HTTP request, Kafka message production, and potentially downstream Spark processing (if Spark was also instrumented).

Verification:

- **Grafana Alloy Logs:** Logs confirm that Grafana Alloy is successfully receiving and forwarding trace data from the instrumented FastAPI application. This demonstrates OpenTelemetry's capability to collect distributed traces, which are critical for debugging end-to-end performance and errors in complex data pipelines.

Advanced Use Case 2: Structured Logging and Contextualization

Objective: To demonstrate how OpenTelemetry integrates with standard logging frameworks to automatically inject trace and span IDs into log messages, enabling easier correlation of logs with specific requests and traces.

Role in Platform: Enhance debuggability by linking application logs to distributed traces,

providing rich context for troubleshooting complex data flow issues.

Setup/Configuration:

1. **Ensure Basic Use Case setup is complete** (FastAPI has LoggingInstrumentor().instrument(set_logging_format=True)).
2. **Ensure Grafana Alloy is configured to receive logs:** Add an otelcol.exporter.logging for logs in Alloy's config, or if you have a Loki instance, configure otelcol.exporter.loki.
   Example fastapi_app/app/main.py (logging setup, from basic setup):

```
# ...
from opentelemetry.instrumentation.logging import LoggingInstrumentor
# ...
LoggingInstrumentor().instrument(set_logging_format=True)
# ...
```

   Example observability/alloy-config.river (additions for logs):

```
# ...
otelcol.receiver.otlp "default" {
  http { }
  grpc { }
  output {
    metrics = [prometheus.remote_write.default.receiver]
    traces = [] # Keep traces as before, or comment out if not using
    logs = [otelcol.exporter.logging.log_printer.input] # Forward logs to a logging exporter
  }
}

# Exporter to print logs to Alloy's stdout
otelcol.exporter.logging "log_printer" {
  log_level = "debug"
}
# ...
```

Steps to Exercise:

1. **Rebuild and Restart services:** docker compose up --build -d fastapi_ingestor grafana-alloy.
2. **Generate API calls:** Run python3 simulate_data.py.
3. **Observe FastAPI Logs (Docker Compose):**
   docker compose logs -f fastapi_ingestor

   Look for log messages emitted by your FastAPI application.
4. **Observe Grafana Alloy Logs:**
   docker compose logs -f grafana-alloy

   You should see the logs forwarded by FastAPI appearing in Grafana Alloy's stdout

(because of log_printer exporter).

Verification:

- **Logs Content:** Log messages from FastAPI (in both its own container logs and Grafana Alloy's logs if forwarded) will contain additional fields like trace_id and span_id. These IDs will correspond to the traces generated for the respective API requests. This demonstrates how OpenTelemetry enriches logs with tracing context, making it easier to connect log events to specific request flows in a distributed system.

Advanced Use Case 3: Custom Metric Types and Attributes for Business Monitoring

Objective: To demonstrate how to define and record custom OpenTelemetry metrics (beyond basic counters, e.g., gauges or histograms) with rich attributes, allowing for deep business-level monitoring of your data platform.

Role in Platform: Collect domain-specific business metrics (e.g., "number of fraudulent transactions detected", "insurance claim processing duration", "data quality validation success rate"), providing insights directly relevant to business value and data quality.

Setup/Configuration:

1. **Ensure Basic Use Case setup is complete** (FastAPI is instrumented and sending basic counters).
2. **Modify fastapi_app/app/main.py:** Add a histogram metric to track the processing duration of ingestion requests or a gauge to track queue sizes. Add more specific attributes to existing counters.
   Example fastapi_app/app/main.py (further conceptual additions to existing meter):

```
# ... existing OpenTelemetry setup ...

# Add a Histogram to track ingestion latency
ingestion_latency_histogram = meter.create_histogram(
  "ingestion.request.duration",
  description="Duration of data ingestion requests",
  unit="ms",
  # Explicitly define buckets for better granularity in Grafana
  # Recommended to use power-of-2 values for buckets
  boundaries=[0.01, 0.05, 0.1, 0.2, 0.5, 1.0, 2.5, 5.0, 10.0, 20.0, 50.0, 100.0, 200.0,
500.0, 1000.0, 2000.0, 5000.0, 10000.0]
)

# --- inside ingest_financial_transaction endpoint ---
# ...
    import time
    start_time = time.time()
    # ... existing Kafka send logic ...
    end_time = time.time()
```

```
        duration_ms = (end_time - start_time) * 1000
        ingestion_latency_histogram.record(duration_ms, {"endpoint":
"/ingest-financial-transaction", "status": "success"})
# ... similar for insurance claims ...
```

Note: Actual FastAPIInstrumentor already captures HTTP request durations. This example shows adding a *custom* duration metric for specific internal logic if needed.

Steps to Exercise:

1. **Rebuild and Restart FastAPI:** docker compose up --build -d fastapi_ingestor.
2. **Generate API calls:** Run python3 simulate_data.py.
3. **Access Grafana:** Go to http://localhost:3000.
4. **Query Custom Metrics with Attributes:**
    - Open the "Explore" view and select your Prometheus data source.
    - For the custom counters, use sum by (transaction_type, currency) (financial_transactions_ingested_total).
    - For the histogram, query rate(ingestion_request_duration_bucket{endpoint="/ingest-financial-transaction" }[1m]) or histogram_quantile(0.99, sum by(le, endpoint) (rate(ingestion_request_duration_bucket[1m]))).
    - **Observe:** The graphs will show the totals segmented by the attributes you defined (e.g., total financial transactions by transaction_type and currency). The histogram will provide insights into the distribution of your ingestion latency.

Verification:

- **Grafana Dashboards:** The custom metrics appear with their associated attributes as labels. You can create panels that display these metrics aggregated or filtered by the attributes, demonstrating OpenTelemetry's ability to capture rich, multi-dimensional business and operational data. This is crucial for building comprehensive dashboards that provide insights into business process performance and data quality.

This concludes the guide for OpenTelemetry.