Highlighting Grafana Alloy: Unified Telemetry Collection

Grafana Alloy is an OpenTelemetry Collector distribution that acts as a powerful and highly configurable agent for collecting, processing, and exporting telemetry data – metrics, logs, and traces – from your data platform services. It is the central nervous system for your observability stack, gathering data from various sources (like cAdvisor, FastAPI, Spark) and forwarding it to monitoring tools like Grafana.

This guide will demonstrate basic and advanced use cases of Grafana Alloy, leveraging your **Advanced Track** local environment setup and its integration with other observability components.

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 Grafana Alloy's role in the **Observability** section.

Basic Use Case: Collecting Metrics from Prometheus Endpoints

Objective: To demonstrate how Grafana Alloy scrapes Prometheus-compatible metrics endpoints (like those exposed by cAdvisor and FastAPI) and forwards them to Grafana for visualization.

Role in Platform: Act as the primary metrics collector, standardizing the ingestion of operational data from diverse services into your monitoring system.

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 cAdvisor, fastapi_ingestor (with Prometheus instrumentation), grafana-alloy, and grafana.
2. **Verify Grafana Alloy configuration:** Review your observability/alloy-config.river file. It should contain prometheus.scrape blocks for various services.
   Example observability/alloy-config.river snippets (conceptual):

```
# observability/alloy-config.river
# ... other components ...

prometheus.remote_write "default" {
  # This receiver sends all scraped metrics to Grafana (acting as a Prometheus storage)
  url = "http://grafana:9090/api/prom/push" # Grafana's Prometheus-compatible remote write endpoint
}
```

```
# Scrape metrics from cAdvisor (container metrics)
prometheus.scrape "cadvisor" {
  targets    = [{"__address__" = "cadvisor:8080"}] # 'cadvisor' is the service name in docker-compose
  forward_to = [prometheus.remote_write.default.receiver]
  job        = "cadvisor" # Label for the metrics
}

# Scrape metrics from FastAPI ingestor (application metrics)
prometheus.scrape "fastapi_ingestor" {
  targets    = [{"__address__" = "fastapi_ingestor:8000"}] # 'fastapi_ingestor' is the service name
  metrics_path = "/metrics" # The path where FastAPI exposes its metrics
  forward_to = [prometheus.remote_write.default.receiver]
  job        = "fastapi_ingestor" # Label for the metrics
}

# ... and potentially scrape Spark JMX exporter, Kafka JMX exporter, etc.
```

3. **Ensure Grafana is accessible:** http://localhost:3000.
4. **Generate activity:** Run python3 simulate_data.py to create traffic to FastAPI.

Steps to Exercise:

1. **Observe Grafana Alloy Logs:**
   - Open a terminal and watch the logs of the grafana-alloy container:
     ```
     docker compose logs -f grafana-alloy
     ```
   - You should see messages indicating that Grafana Alloy is actively "scraping" or "collecting" metrics from cadvisor:8080/metrics and fastapi_ingestor:8000/metrics.
2. **Access Grafana Dashboards:**
   - Go to http://localhost:3000.
   - Navigate to a dashboard that displays container metrics (e.g., "Docker Container Overview") and another that shows FastAPI application metrics (e.g., "Health Dashboard" or a custom one).
   - Look for panels showing CPU, memory usage for containers (from cAdvisor via Alloy) and API request rates, latency (from FastAPI via Alloy).

Verification:

- **Grafana Alloy Logs:** Confirm that Alloy logs show successful scraping attempts and no connection errors to the target endpoints.
- **Grafana Dashboards:** Metrics from both cAdvisor (container resources) and FastAPI (API performance) are populating correctly in Grafana, demonstrating that Grafana Alloy is successfully collecting and forwarding this data.

Advanced Use Case 1: Processing and Relabeling Metrics

Objective: To demonstrate how Grafana Alloy can process and transform metrics before forwarding them, including relabeling, filtering, and adding new attributes. This is crucial for standardizing metric names, adding useful metadata, and reducing noise.
Role in Platform: Cleanse, enrich, and standardize telemetry data, ensuring consistency and usability for monitoring and alerting.
Setup/Configuration:

1. **Ensure Basic Use Case is running.**
2. **Modify observability/alloy-config.river:** Add a relabel block to an existing prometheus.scrape component. Let's relabel the instance label for FastAPI metrics to be more descriptive, and add a static environment label.
   Example observability/alloy-config.river snippet (modification to fastapi_ingestor scrape):

   ```
   # observability/alloy-config.river
   # ...
   prometheus.scrape "fastapi_ingestor" {
     targets    = [{"__address__" = "fastapi_ingestor:8000"}]
     metrics_path = "/metrics"
     forward_to = [prometheus.remote_write.default.receiver]
     job        = "fastapi_ingestor"

     # Relabeling example:
     # Change 'instance' label (which might be 'fastapi_ingestor:8000')
     # to just the service name and add an environment label.
     relabel_configs {
       source_labels = ["__address__"]
       regex         = "(.*):.*"
       target_label  = "instance"
       replacement   = "$1" # Keep only the service name part (e.g., "fastapi_ingestor")
     }
     relabel_configs {
       source_labels = [] # Empty means apply to all metrics
       target_label  = "environment"
       replacement   = "local_dev" # Add a static 'environment' label
     }
     # Example: drop metrics starting with 'go_'
     # relabel_configs {
     #   source_labels = ["__name__"]
     #   regex         = "go_.*"
     #   action        = "drop"
   ```

```
  # }
}
# ...
```

Steps to Exercise:

1. **Restart Grafana Alloy:** docker compose restart grafana-alloy to load the new configuration.
2. **Generate traffic:** Run python3 simulate_data.py.
3. **Inspect Metrics in Grafana:**
   - Go to http://localhost:3000.
   - Open the "Explore" view in Grafana and select your Prometheus data source.
   - Enter a PromQL query for a FastAPI metric, for example: http_requests_total.
   - **Observe:** Instead of instance="fastapi_ingestor:8000", you should now see instance="fastapi_ingestor" and a new label environment="local_dev". This confirms the relabeling.
   - If you applied the drop rule, verify that metrics like go_goroutines are no longer present.

Verification:

- **Grafana Metrics Explorer:** Queries show the modified labels (e.g., instance="fastapi_ingestor", environment="local_dev"), confirming that Grafana Alloy successfully applied the relabeling rules.
- **Logs:** Grafana Alloy logs might show "processing" or "relabeling" messages if debug logging is enabled.

Advanced Use Case 2: Aggregating Metrics for Service-Level Objectives (SLOs)

Objective: To demonstrate how Grafana Alloy can preprocess and aggregate metrics from multiple instances of a service (or different services) before sending them to the monitoring backend. This is vital for calculating service-level metrics needed for SLOs (e.g., total requests across all replicas, aggregated latency).

Role in Platform: Enable the calculation of high-level service health indicators and facilitate alerting on SLO violations, crucial for data platform reliability.

Setup/Configuration:

1. **Simulate multiple FastAPI instances (conceptual in docker-compose.yml):** While we only have one fastapi_ingestor in the standard setup, you can conceptualize scaling it or having another similar service. For this demo, we'll demonstrate aggregation across multiple *jobs* or *instances* that might appear from different sources.
2. **Modify observability/alloy-config.river to aggregate:** Use prometheus.rule component within Alloy to define recording rules.

   Example observability/alloy-config.river snippet (conceptual, assumes fastapi_ingestor and another ingestor_replica job):

   ```
   # observability/alloy-config.river
   ```

```
# ...
prometheus.scrape "fastapi_ingestor" {
  targets    = [{"__address__" = "fastapi_ingestor:8000"}]
  metrics_path = "/metrics"
  forward_to = [prometheus.remote_write.default.receiver]
  job        = "fastapi_ingestor" # Job label for this instance
}

# CONCEPTUAL: If you had another FastAPI replica or a similar service
# prometheus.scrape "ingestor_replica" {
#   targets    = [{"__address__" = "another_ingestor:8000"}]
#   metrics_path = "/metrics"
#   forward_to = [prometheus.remote_write.default.receiver]
#   job        = "ingestor_replica" # Different job label
# }

# Define a recording rule to aggregate total requests across all ingestor instances/jobs
prometheus.rule "ingestor_total_requests_sum" {
  label_match {
    name  = "job"
    value = "(fastapi_ingestor|ingestor_replica)" # Matches both original and conceptual
replica
  }
  rules = [
    {
      record = "ingestor_api_total_requests_sum" # New aggregated metric name
      expr   = "sum by (le, path) (rate(http_requests_total[1m]))" # Sums rate over all
matched jobs/instances
    }
  ]
  forward_to = [prometheus.remote_write.default.receiver]
}
# ...
```

Steps to Exercise:

1. **Restart Grafana Alloy:** docker compose restart grafana-alloy.
2. **Generate traffic:** Run python3 simulate_data.py.
3. **Query Aggregated Metrics in Grafana:**
   - Go to http://localhost:3000.
   - Open the "Explore" view.
   - Query the new aggregated metric: ingestor_api_total_requests_sum.
   - **Observe:** This metric should now represent the combined request rate from all configured FastAPI instances (in this conceptual example, it will just reflect

fastapi_ingestor if ingestor_replica is not active, but the rule is set up for aggregation).

Verification:

- **Grafana Explore:** The aggregated metric ingestor_api_total_requests_sum can be queried, demonstrating that Grafana Alloy is applying recording rules to create new, higher-level metrics from your raw scraped data. This is foundational for calculating SLOs like "total ingestion RPS."

Advanced Use Case 3: Fan-out Data to Multiple Monitoring Backends (Conceptual)

Objective: To demonstrate Grafana Alloy's flexibility in sending telemetry data to multiple destinations simultaneously. For example, metrics to Grafana (as Prometheus storage) and traces to a separate Jaeger/Tempo backend.

Role in Platform: Enable a multi-faceted observability strategy, allowing different types of telemetry data to be routed to specialized analysis tools without requiring multiple agents on each service.

Setup/Configuration:

1. **Ensure Basic Use Case is running.**
2. **Add another telemetry backend (conceptual):** For this local setup, let's simulate sending traces to a dummy endpoint (representing Jaeger/Tempo) while metrics still go to Grafana.
3. **Modify observability/alloy-config.river:**
   - Add an otelcol.exporter.otlp block for traces.
   - Modify otelcol.receiver.otlp to forward traces to this new exporter.
   - Ensure your fastapi_app is instrumented for traces (e.g., opentelemetry-instrument fastapi_app.app.main).

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

```
# observability/alloy-config.river
# ... existing prometheus.scrape and remote_write "default" for metrics ...

# 1. Define an OTLP receiver for traces (FastAPI/Spark would send traces here)
otelcol.receiver.otlp "default" {
  http { } # Listen for OTLP HTTP
  grpc { } # Listen for OTLP gRPC
  output {
    # Forward traces to a specific exporter
    traces  = [otelcol.exporter.otlp.jaeger_mock.input]
    metrics = [prometheus.remote_write.default.receiver] # Metrics still go to Grafana
    logs    = [] # No specific log forwarding for this example
  }
}

# 2. Define an OTLP exporter for traces (e.g., to a mock Jaeger/Tempo endpoint)
```

```
otelcol.exporter.otlp "jaeger_mock" {
  client {
    endpoint = "http://jaeger-all-in-one:4318" # Conceptual Jaeger/Tempo endpoint in Docker Compose
    # In a real setup, this would be your Jaeger/Tempo URL
  }
}

# Update docker-compose.yml for conceptual Jaeger/Tempo service
# services:
#   jaeger-all-in-one:
#     image: jaegertracing/all-in-one:latest
#     ports:
#       - "16686:16686" # Jaeger UI
#       - "4318:4318"   # OTLP HTTP receiver
#       - "4317:4317"   # OTLP gRPC receiver
#     healthcheck:
#       test: ["CMD-SHELL", "wget -q -O - http://localhost:16686/actuator/health | grep -q 'UP'"]
#       interval: 5s
#       timeout: 3s
#       retries: 5
```

Steps to Exercise:

1. **Update docker-compose.yml (if adding Jaeger/Tempo mock):** Add the conceptual jaeger-all-in-one service.
2. **Rebuild and Restart services:** docker compose up --build -d.
3. **Ensure FastAPI is instrumented for OpenTelemetry traces:**
   - You would typically run your FastAPI app with opentelemetry-instrument uvicorn fastapi_app.app.main:app --host 0.0.0.0 --port 8000. This is usually done via entrypoint in Dockerfile or command in docker-compose.yml.
   - Ensure the environment variable OTEL_EXPORTER_OTLP_ENDPOINT=http://grafana-alloy:4318 (or 4317 for gRPC) is set for FastAPI (and Spark if instrumented). This tells your services to send traces to Alloy.
4. **Generate traffic:** Run python3 simulate_data.py.
5. **Observe Grafana Alloy Logs:** You should see logs indicating Alloy receiving and forwarding both metrics (to Grafana) and traces (to jaeger-mock endpoint).
6. **Verify Traces (Conceptual):** If you had a real Jaeger UI running, you would access it (http://localhost:16686) and search for traces from your fastapi_ingestor service.

Verification:

- **Grafana Alloy Logs:** Logs confirm dual forwarding of telemetry data types to different conceptual backends.
- **Backend Status:** While a full trace visualization isn't possible without a fully deployed Jaeger, the logs confirm Alloy's capability to fan out data. This demonstrates Alloy's ability to act as a universal collector and router for various telemetry signals, enabling a

comprehensive and flexible observability strategy.