Highlighting Grafana: Interactive Data Visualization and Monitoring

Grafana is an open-source platform for interactive data visualization and monitoring. In your data platform, it is the primary interface for gaining real-time operational insights, visualizing key performance indicators (KPIs), and setting up alerts based on the telemetry data collected by Grafana Alloy from all services. It transforms raw metrics into actionable dashboards, crucial for maintaining system health and reliability.
This guide will demonstrate basic and advanced use cases of Grafana, leveraging your **Advanced Track** local environment setup and its integration with Grafana Alloy and other 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's role in the **Observability** section and the "Highlighting Grafana Alloy" document.

Basic Use Case: Real-time Operational Dashboards

Objective: To demonstrate how Grafana visualizes real-time operational metrics collected by Grafana Alloy from various services (FastAPI, Kafka, Spark, cAdvisor) in a unified dashboard.
Role in Platform: Provide a single pane of glass for monitoring the health, performance, and resource utilization of all data platform components.
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 (with Prometheus instrumentation), kafka, spark, cAdvisor, grafana-alloy, and grafana.
2. **Verify Grafana is accessible:** Navigate to http://localhost:3000 in your web browser. (Initial login: admin/admin if not configured otherwise).
3. **Ensure Grafana Alloy is collecting metrics:** Confirm grafana-alloy is running and its logs show successful scraping from fastapi_ingestor:8000/metrics and cadvisor:8080/metrics, and forwarding to Grafana (as covered in "Highlighting Grafana Alloy: Basic Use Case").
4. **Generate activity:** Run python3 simulate_data.py to continuously send data, ensuring all services are active and generating metrics.

Steps to Exercise:

1. **Access Grafana UI:** Open your web browser and go to http://localhost:3000.
2. **Navigate to a Dashboard:**
    - From the left-hand navigation, click on "Dashboards" and select a pre-provisioned dashboard (e.g., "Health Dashboard", "Docker Container Overview", or "Kafka Overview"). Your setup should ideally include a "Health Dashboard" that pulls metrics from various sources.

    - If no dashboards are pre-provisioned, you can import one from Grafana Labs (e.g., "Node Exporter Full" ID 1860 for host metrics, or "cAdvisor / Prometheus Host and Container" ID 14210, which relies on cAdvisor metrics).
3. **Observe Real-time Metrics:**
    - **FastAPI Metrics:** Look for panels showing "API Request Rate (RPS)" and "API Latency (P99)". These reflect the performance of your ingestion layer.
    - **Kafka Metrics:** Find panels for "Kafka Consumer Lag" (for raw_financial_transactions and raw_insurance_claims topics) and "Kafka Broker Throughput." These indicate the health of your streaming buffer and whether consumers are keeping up.
    - **Spark Metrics:** Observe panels showing "Spark CPU Utilization" or "Spark Memory Usage," reflecting the resources consumed by your data processing jobs.
    - **Container Metrics (via cAdvisor):** Look at overall "Container CPU Usage" and "Container Memory Usage" to see resource consumption across individual Docker services.

Verification:

- **Grafana Dashboards:** All panels in the selected dashboards are actively populating with data, and the graphs reflect the ongoing activity of your data platform components (e.g., fluctuating RPS for FastAPI, low and stable consumer lag for Kafka, varying CPU/memory for Spark). This confirms the end-to-end flow of telemetry data and Grafana's ability to visualize it in real-time.

Advanced Use Case 1: Alerting on Service Level Objective (SLO) Violations

Objective: To demonstrate how to configure Grafana alerts to notify operational teams when key Service Level Objectives (SLOs) for the data platform are violated.
Role in Platform: Enable proactive monitoring and rapid response to system health deviations, reducing Mean Time To Detection (MTTD) and preventing minor issues from escalating into major incidents.
Setup/Configuration:

1. **Ensure Basic Use Case is running:** Data is flowing and metrics are being collected in Grafana.
2. **Understand SLOs:** Refer to testing-observability-addendum (Section 5.6.1) for examples of SLIs and SLOs (e.g., "Kafka Consumer Lag < 10,000 messages for 99.9% of time").
3. **Identify a Metric for Alerting:** We'll use Kafka consumer lag for raw_financial_transactions.
4. **(Optional) Configure a Notification Channel:** For real alerts, you'd configure an email, Slack, PagerDuty, etc., channel in Grafana ("Alerting" -> "Contact points"). For this local demo, you'll primarily observe the alert status in the Grafana UI.

Steps to Exercise:

1. **Create a Grafana Alert Rule:**

- In Grafana UI, navigate to "Alerting" -> "Alert Rules".
- Click "New alert rule".
- **Choose Data Source:** Select your Prometheus data source (which Grafana Alloy sends to).
- **Define Query (PromQL):** Enter a query for Kafka consumer lag.
  kafka_consumergroup_group_lag{consumergroup="spark-financial-consumer-group", topic="raw_financial_transactions"}
  # Replace 'spark-financial-consumer-group' with your actual Spark consumer group name

- **Define Condition:** Set a threshold that will be easily violated for demonstration.
  - **Condition:** WHEN last() OF A IS ABOVE 5000 (meaning if lag is over 5000 messages).
  - **For:** 2m (meaning the condition must be true for 2 consecutive minutes before the alert fires).
- **Configure Notifications (optional but recommended for real use):** Add a "Contact point" to send notifications.
- **Add Annotations:** This is crucial for alert fatigue mitigation. Provide summary, description, remediation steps, dashboard_link, and runbook_link (conceptual links to your internal runbook documentation, as detailed in testing-observability-addendum, Section 5.6.2).
  - summary: "High Kafka Consumer Lag for Financial Transactions"
  - description: "The Spark job consuming 'raw_financial_transactions' is falling behind. Data freshness for financial data is impacted."
  - remediation: "1. Check Spark job logs for errors (e.g., OOM). 2. Verify Spark cluster resources (CPU/memory) in Grafana. 3. Consider scaling Spark executors or optimizing job logic. 4. Refer to runbook for detailed steps."
  - dashboard_link: http://localhost:3000/d/<your-kafka-dashboard-uid>
  - runbook_link: /runbooks/kafka_consumer_lag.md
- **Save Alert Rule.**

2. **Simulate an SLO Violation:**
   - Ensure simulate_data.py is running and actively sending financial data to Kafka.
   - **Pause the Spark container:** docker compose pause spark. This will cause the Kafka consumer lag for raw_financial_transactions to rapidly increase.
   - Wait for the "For" duration you set (e.g., 2 minutes).
3. **Observe Alert Firing:**
   - In Grafana, go back to "Alerting" -> "Alert Rules".
   - Your newly created alert rule should transition from "OK" to "Pending" and then to "Firing" (red status).
   - Click on the alert rule to see its current state, active alerts, and the annotation details.

Verification:

- **Grafana Alerting:** The alert rule correctly changes status to "Firing" when the defined

condition (high Kafka consumer lag) is met for the specified duration.
- **Dashboards:** The corresponding consumer lag panel in your Kafka dashboard will clearly show the spike in lag, providing visual context for the alert.
- **Alert Annotations:** The alert's details display the rich context (summary, description, remediation, links), demonstrating how Grafana helps in managing alert fatigue and guiding incident response.

Advanced Use Case 2: Templating Dashboards for Dynamic Views

Objective: To demonstrate how Grafana's templating feature allows you to create dynamic dashboards where users can filter or select data based on variables (e.g., Kafka topic, container name, service name).
Role in Platform: Provide flexible, self-service monitoring interfaces, allowing different teams or users to customize their view of the data without creating countless static dashboards.
Setup/Configuration:

1. **Ensure Basic Use Case is running:** Metrics from multiple sources (FastAPI, Kafka, cAdvisor) are flowing into Grafana with various labels (job, topic, instance, container).

Steps to Exercise:

1. **Create a New Dashboard:**
    - In Grafana, click the "+" icon on the left-hand navigation and select "New Dashboard".
    - Click "Add a new panel" to start.
2. **Add a Template Variable:**
    - Go to "Dashboard settings" (the gear icon on the top right).
    - Select "Variables" from the left menu.
    - Click "Add variable".
    - **Name:** topic
    - **Type:** Query
    - **Data source:** Your Prometheus data source.
    - **Query:** label_values(kafka_consumergroup_group_lag, topic) (This fetches all unique Kafka topic names that report consumer lag metrics).
    - **Selection Options:** Enable "Multi-value" and "Include All option".
    - Click "Add". You'll now see a dropdown menu at the top of your dashboard.
3. **Create Panels Using the Variable:**
    - Go back to your dashboard (click "Dashboard" in the top left).
    - Add a new panel.
    - **Query:** In the PromQL query editor, use the variable: kafka_consumergroup_group_lag{topic=~"$topic"}

      The ~ regex operator allows selecting multiple values from the dropdown.
    - **Observe:** The panel will initially show data for all topics (if "All" is selected).
    - **Change Variable:** Select a specific topic from the "topic" dropdown (e.g.,

raw_financial_transactions). The panel will instantly update to show data only for that topic.

4. **Repeat for other metrics (Optional):**
    - Add another variable for job (using label_values(__name__, job) to get all job names).
    - Create a panel for FastAPI RPS: rate(http_requests_total{job=~"$job"}[1m]).
    - Change the job variable to switch between viewing FastAPI, cAdvisor, etc.

Verification:

- **Dynamic Dashboards:** The dashboard demonstrates dynamic filtering and visualization based on the selected template variable, proving that Grafana can provide highly customizable views of your telemetry data. This empowers users to quickly focus on specific pipelines or services.

Advanced Use Case 3: Integrating with OpenMetadata for Data Context (Conceptual)

Objective: To conceptually demonstrate how Grafana dashboards can be enriched with context from OpenMetadata, bridging the gap between operational monitoring and data governance information.

Role in Platform: Provide a holistic view for data practitioners, allowing them to understand not just *if* a pipeline is failing, but *what* specific data assets are affected, their owners, and their business descriptions.

Setup/Configuration:

1. **Ensure OpenMetadata is running and populated:** Your Kafka topics and Delta Lake tables should be cataloged in OpenMetadata (http://localhost:8585).
2. **Identify Relevant Assets:** Know the URL structure for OpenMetadata assets (e.g., http://localhost:8585/metadata/explore/topics/<topic-fqn> or http://localhost:8585/metadata/explore/tables/<table-fqn>).

Steps to Exercise (Conceptual):

1. **Add Custom Link to a Grafana Panel:**
    - Go to an existing panel in a Grafana dashboard (e.g., your Kafka consumer lag panel for raw_financial_transactions).
    - Click on the panel title, then "Edit Panel".
    - Go to the "Links" tab in the panel options.
    - Click "Add link".
    - **Title:** "View in OpenMetadata"
    - **Type:** Absolute URL
    - **URL:** http://localhost:8585/metadata/explore/topics/{{__series.labels.topic}}

      Explanation: {{__series.labels.topic}} is a Grafana variable that will dynamically pull the topic label from the metric currently displayed in the panel.
    - **Open in new tab:** Yes.

    - Save the link and the panel.
2. **Interact with the Link:**
    - Go back to the dashboard view.
    - Click on the panel title (e.g., the Kafka consumer lag panel). A small dropdown will appear with "View in OpenMetadata".
    - Click the link. It should open a new tab directly to the corresponding Kafka topic's page in OpenMetadata.
3. **Discuss Enhancements:**
    - **Annotations for Lineage:** While not directly pulling full lineage into Grafana, you could set up Airflow DAGs to create Grafana annotations that point to specific Spline/OpenMetadata lineage URLs whenever a critical batch job completes, showing "New data loaded for X, see lineage at Y".
    - **Embedded iFrames (Less Recommended):** For internal tools, one could theoretically embed OpenMetadata views within Grafana using iFrame panels, though this can have security and layout complexities.
    - **Enrichment in Grafana Alloy:** Conceptually, Grafana Alloy could be extended (though more advanced) to pull descriptive metadata from OpenMetadata and attach it as labels to metrics, enriching them with business context (e.g., data_owner, data_sensitivity).

Verification (Conceptual):

- **Contextual Links:** The ability to navigate directly from an operational metric in Grafana to its corresponding asset in OpenMetadata demonstrates a practical integration of observability and governance. This provides a richer understanding of data assets and their operational status for data consumers and operations teams. This bridges the gap between "what's broken" and "whose data is affected."

This concludes the guide for Grafana.