Documentation

End-to-End Real-Time Monitoring with Kafka, InfluxDB, and Grafana (Windows, Without Docker)

This guide provides all the necessary steps, commands, and code snippets to set up a real-time data pipeline from a Python Kafka producer, through a Python Kafka consumer writing to InfluxDB, and finally visualized in Grafana. This is designed for a **Windows environment without Docker**.

---

1. Prerequisites

- **Java Development Kit (JDK):** Kafka requires Java.[4] Download and install a recent JDK (e.g., OpenJDK 11 or 17).
  - **Download:** Search for "OpenJDK 17 download" and get the installer for Windows.
  - **Installation:** Follow the installer instructions.
  - **Environment Variable:** Ensure `JAVA_HOME` is set to your JDK installation directory (e.g., `C:\Program Files\Java\jdk-17`) and `JAVA_HOME\bin` is added to your system `Path`.
    - To check: Open Command Prompt and type `java -version`.
- **Python:** Python 3.8+ is recommended.
  - **Download:** Visit python.org and download the latest installer.
  - **Installation:** Run the installer. **Crucially, ensure "Add Python to PATH" is checked during installation.**
  - To check: Open Command Prompt and type `python --version` and `pip --version`.

---

2. Setup Kafka (with Zookeeper)

Kafka relies on Zookeeper for metadata management. For simplicity, we'll use a Kafka distribution that bundles Zookeeper.

2.1. Download Kafka

1. Go to the Apache Kafka downloads page: https://kafka.apache.org/downloads
2. Download a stable binary release that includes Zookeeper (e.g., **Scala 2.13 - Kafka 3.6.1**). Look for a `.tgz` or `.zip` file.

- *Note: Newer Kafka versions (4.0.0+) primarily use KRaft and might not bundle Zookeeper.[5] Using an older stable version like 3.6.1 simplifies this setup.*

3. Extract the downloaded archive (e.g., `kafka_2.13-3.6.1.tgz`) to a simple path like `C:\kafka`. This will create a directory like `C:\kafka\kafka_2.13-3.6.1`. We'll refer to this as `%KAFKA_HOME%`.

2.2. Configure Kafka

1. Navigate to `%KAFKA_HOME%\config`.
2. **Edit `server.properties`:**
   - Open `server.properties` in a text editor (like Notepad++ or VS Code).
   - Find the line `log.dirs=/tmp/kafka-logs` and change it to a Windows-friendly path, e.g.:
   - Properties

```
log.dirs=C:/kafka-logs
```

   -
   -
3. **Edit `zookeeper.properties`:**
   - Open `zookeeper.properties` in a text editor.
   - Find the line `dataDir=/tmp/zookeeper` and change it to a Windows-friendly path, e.g.:
   - Properties

```
dataDir=C:/zookeeper-data
```

   -
   -

2.3. Start Zookeeper & Kafka

Important: Open a **separate Command Prompt window for each service** (Zookeeper, Kafka, Producer, Consumer). Do not close these windows as long as you want the services to run.

1. **Start Zookeeper:**
   - Open a new Command Prompt.
   - Navigate to your Kafka directory:
   - DOS

```
cd C:\kafka\kafka_2.13-3.6.1
```

- Run Zookeeper:

```
.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
```

- You should see Zookeeper start logging messages. Look for "binding to" and "Started."

2. **Start Kafka Broker:**
   - Open a **new** Command Prompt.
   - Navigate to your Kafka directory:

```
cd C:\kafka\kafka_2.13-3.6.1
```

- Run Kafka:

```
.\bin\windows\kafka-server-start.bat .\config\server.properties
```

- Kafka will start logging. Look for "Kafka Server started."

---

3. Setup InfluxDB OSS v2

3.1. Download InfluxDB

1. Go to the InfluxData downloads page: https://www.influxdata.com/downloads/
2. Select **InfluxDB OSS**, then choose **Version 2.7.12** (or the latest stable `2.x` Windows version).
3. Select **Windows** as the platform.
4. Download the `influxdb2-2.7.12-windows.zip` file.
5. Extract the contents of the ZIP file to a simple path, e.g., `C:\influxdb`.

3.2. Start InfluxDB

1. Open a **new** Command Prompt.
2. Navigate to your InfluxDB directory:
3. DOS

```
cd C:\influxdb
```

4.
5.
6. Start InfluxDB:
7. DOS

```
.\influxd.exe
```

8.
    - You should see InfluxDB start logging.
9. **Initial Setup (First Run Only):**
    - Open your web browser and go to http://localhost:8086.
    - Follow the on-screen prompts for initial setup:
        - **Username:** admin (or choose your own)
        - **Password:** your_secure_password (choose a strong one)
        - **Organization Name:** my-org (or choose your own, e.g., sensor_data_org)
        - **Initial Bucket Name:** sensor_bucket (or choose your own, e.g., kafkametrics15-7). **Remember this exact name!**
    - **Crucially, after setup, InfluxDB will display an "Operator Token" or "All Access API Token". Copy this token immediately and save it securely.** You will not see it again. This token is required for all API interactions.

---

4. Setup Grafana

4.1. Download & Install Grafana

1. Go to the Grafana downloads page: https://grafana.com/grafana/download
2. Select **Grafana Open Source**, then choose the latest **stable version (e.g., 12.0.2)**.
3. Select **Windows** as the platform.
4. Download the installer (usually an .msi file).
5. Run the installer and follow the on-screen prompts. Default installation path is usually fine (e.g., C:\Program Files\GrafanaLabs\grafana).

4.2. Start Grafana

1. Grafana often installs as a Windows service. You can check the "Services" application (search for `services.msc`) and start `Grafana` if it's not running.
2. Alternatively, you can manually start it from the command line:
    - Open a **new** Command Prompt.
    - Navigate to the Grafana `bin` directory:
    - DOS

```
cd "C:\Program Files\GrafanaLabs\grafana\bin"
```

    -
    -
    - Start Grafana:
    - DOS

```
.\grafana-server.exe
```

    -
    -
    - You should see Grafana logging messages.

4.3. Access Grafana UI

1. Open your web browser and go to `http://localhost:3000`.
2. **Login:**
    - **Default Username:** `admin`
    - **Default Password:** `admin`
    - You will be prompted to change the password on first login.

---

5. Python Code for Producer & Consumer

Ensure you have Python installed and added to PATH (from Prerequisites).

5.1. Install Python Libraries

Open a **new** Command Prompt and run:

DOS

```
pip install kafka-python influxdb-client
```

5.2. Kafka Producer Script (`kafka_producer.py`)

Create a file named `kafka_producer.py` and paste the following:

Python

```
import time
import json
import random
from kafka import KafkaProducer

# Configuration
KAFKA_BOOTSTRAP_SERVERS = 'localhost:9092'
KAFKA_TOPIC = 'sensor_data_topic'

# Initialize Kafka Producer
producer = KafkaProducer(
    bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

print(f"Kafka Producer initialized for topic: {KAFKA_TOPIC}")

# Simulate sensor data
sensor_ids = ['sensor_A', 'sensor_B', 'sensor_C']

def generate_sensor_data(sensor_id):
    """Generates a dictionary with simulated sensor readings."""
    temperature = round(random.uniform(20.0, 30.0), 2)
    humidity = round(random.uniform(50.0, 60.0), 2)
    timestamp = int(time.time() * 1000) # Milliseconds Unix timestamp
    return {
        'sensor_id': sensor_id,
        'temperature': temperature,
        'humidity': humidity,
        'timestamp': timestamp
    }

try:
    while True:
        for sensor_id in sensor_ids:
            data = generate_sensor_data(sensor_id)
            producer.send(KAFKA_TOPIC, data)
            print(f"Sent to Kafka: {data}")
        time.sleep(2) # Send data every 2 seconds for all sensors

except KeyboardInterrupt:
```

```
    print("Stopping producer...")
finally:
    producer.close()
    print("Producer closed.")
```

5.3. Kafka Consumer to InfluxDB Script (`kafka_to_influx.py`)

Create a file named `kafka_to_influx.py` and paste the following. **Crucially, update `INFLUXDB_TOKEN`, `INFLUXDB_ORG`, and `INFLUXDB_BUCKET` to match your InfluxDB setup!**

```
Python

import json
from kafka import KafkaConsumer
from influxdb_client import InfluxDBClient, Point, WriteOptions
from influxdb_client.client.write_api import SYNCHRONOUS

# Kafka Configuration
KAFKA_BOOTSTRAP_SERVERS = 'localhost:9092'
KAFKA_TOPIC = 'sensor_data_topic'
KAFKA_GROUP_ID = 'influxdb_consumer_group'

# InfluxDB Configuration
INFLUXDB_URL = "http://localhost:8086"
INFLUXDB_TOKEN = "YOUR_INFLUXDB_ALL_ACCESS_TOKEN" # <<--- IMPORTANT:
REPLACE WITH YOUR ACTUAL TOKEN
INFLUXDB_ORG = "my-org"                    # <<--- IMPORTANT: REPLACE WITH YOUR ACTUAL
ORGANIZATION NAME
INFLUXDB_BUCKET = "sensor_bucket"          # <<--- IMPORTANT: REPLACE WITH YOUR
ACTUAL BUCKET NAME

# Initialize InfluxDB Client
client = InfluxDBClient(url=INFLUXDB_URL, token=INFLUXDB_TOKEN, org=INFLUXDB_ORG)
write_api = client.write_api(write_options=SYNCHRONOUS)

# Initialize Kafka Consumer
consumer = KafkaConsumer(
    KAFKA_TOPIC,
    bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,
    group_id=KAFKA_GROUP_ID,
    value_deserializer=lambda m: json.loads(m.decode('utf-8')),
    auto_offset_reset='earliest', # Start reading from the beginning if no offset is committed
    enable_auto_commit=True,
    consumer_timeout_ms=1000 # Stop after 1 second if no messages are available
)
```

```
print(f"Kafka Consumer initialized for topic: {KAFKA_TOPIC}, group: {KAFKA_GROUP_ID}")
print(f"Writing to InfluxDB URL: {INFLUXDB_URL}, Org: {INFLUXDB_ORG}, Bucket:
{INFLUXDB_BUCKET}")

try:
    for message in consumer:
        try:
            data = message.value
            print(f"Received from Kafka: {data}")

            # Create an InfluxDB Point
            # Measurement: sensor_measurements
            # Tags: sensor_id
            # Fields: temperature, humidity
            # Timestamp: uses the 'timestamp' from the Kafka message, converted to nanoseconds
            point = Point("sensor_measurements") \
                .tag("sensor_id", data['sensor_id']) \
                .field("temperature", data['temperature']) \
                .field("humidity", data['humidity']) \
                .time(data['timestamp'], write_precision="ms") # Use 'ms' for milliseconds precision

            write_api.write(bucket=INFLUXDB_BUCKET, record=point)
            print(f"SUCCESS: Wrote to InfluxDB: {point.to_line_protocol()}")

        except Exception as e:
            print(f"Error processing message or writing to InfluxDB: {e}")
            print(f"Problematic data: {message.value}") # Print the data that caused the error

except KeyboardInterrupt:
    print("Stopping consumer...")
except Exception as e:
    print(f"An unexpected error occurred in the consumer: {e}")
finally:
    consumer.close()
    client.close()
    print("Consumer and InfluxDB client closed.")
```

5.4. Run Python Scripts

Important: Open a **new** Command Prompt for each script.

1. **Start Kafka Producer:**
    - Open a new Command Prompt.
    - Navigate to the directory where you saved `kafka_producer.py`.
    - Run:

- DOS

python kafka_producer.py

-
-
- You should see messages like "Sent to Kafka: {'sensor_id': 'sensor_A', ...}"

2. **Start Kafka Consumer:**
   - Open a **new** Command Prompt.
   - Navigate to the directory where you saved `kafka_to_influx.py`.
   - Run:
   - DOS

python kafka_to_influx.py

-
-
- You should see messages like "Received from Kafka: {...}" and "SUCCESS: Wrote to InfluxDB: ..." If you see "bucket not found", double-check the `INFLUXDB_BUCKET`, `INFLUXDB_ORG`, and `INFLUXDB_TOKEN` in your script against your InfluxDB UI settings.

---

6. Configure Grafana Data Source

1. **Access Grafana UI:** `http://localhost:3000` (login `admin`/`admin` if first time).
2. **Add Data Source:**
   - Hover over the **gear icon** (Configuration) in the left menu.
   - Click **Data sources**.
   - Click **Add data source**.
   - Search for and select **InfluxDB**.
3. **InfluxDB Data Source Settings:**
   - **Name:** `InfluxDB Local` (or any descriptive name)
   - **Query Language:** Select `Flux`
   - **HTTP:**
     - **URL:** `http://localhost:8086`
   - **InfluxDB Details:**
     - **Organization:** `my-org` (or your actual InfluxDB organization name, **case-sensitive**)
     - **Token:** Paste your **All Access API Token** from InfluxDB (the one you copied during InfluxDB setup).

        - **Default Bucket:** `sensor_bucket` (or your actual InfluxDB bucket name, **case-sensitive**)
        - **Min time interval:** `1s` (or `10s` if data comes slower)
    - Click **Save & Test**. You should see "Data source is working."

---

7. Build Grafana Dashboard Panels & Queries

1. **Create New Dashboard:**
    - Hover over the **+ icon** in the left menu.
    - Click **Dashboard**.
    - Click **Add new panel**.
2. **Common Panel Types & Queries (Flux):**
    - **Replace `"your_bucket_name"` with your actual bucket name** (e.g., `sensor_bucket` or `kafkametics15-7`).
    - ### Panel 1: Temperature Over Time (Time Series)
        - **Visualization:** `Time series`
        - **Query:**
        - Code snippet

```
from(bucket: "your_bucket_name")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r._measurement == "sensor_measurements")
  |> filter(fn: (r) => r._field == "temperature")
  |> group(columns: ["sensor_id"]) // Separate line for each sensor
  |> yield(name: "temperature_trend")
```

        -
        -
        - **Panel Options:** Customize `Axes`, `Legend`, `Tooltip` as needed. Set Units to `Temperature -> Celsius`.
    - ### Panel 2: Humidity Over Time (Time Series)
        - **Visualization:** `Time series`
        - **Query:**
        - Code snippet

```
from(bucket: "your_bucket_name")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
```

```
  |> filter(fn: (r) => r._measurement == "sensor_measurements")
  |> filter(fn: (r) => r._field == "humidity")
  |> group(columns: ["sensor_id"])
  |> yield(name: "humidity_trend")
```

-
-
- **Panel Options:** Set Units to Humidity -> Percent (0-100).

Panel 3: Current Temperature (Stat or Gauge)

- **Visualization:** Stat or Gauge
- **Query:**
- Code snippet

```
from(bucket: "your_bucket_name")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r._measurement == "sensor_measurements")
  |> filter(fn: (r) => r._field == "temperature")
  |> group(columns: ["sensor_id"])
  |> last()
  |> yield(name: "current_temp")
```

-
-
- **Panel Options (for Gauge):**
  - Set Min to 0, Max to 50.
  - Add thresholds: e.g., 20 (Green), 25 (Yellow), 30 (Red).
  - Set Units to Temperature -> Celsius.
  - Set Value Options -> Show value to All series if you want a gauge per sensor.

Panel 4: Current Humidity (Stat or Gauge)

- **Visualization:** Stat or Gauge
- **Query:**
- Code snippet

```
from(bucket: "your_bucket_name")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r._measurement == "sensor_measurements")
  |> filter(fn: (r) => r._field == "humidity")
  |> group(columns: ["sensor_id"])
  |> last()
  |> yield(name: "current_humidity")
```

- ■
- ■
- ■ **Panel Options (for Gauge):**
  - ■ Set `Min` to `0`, `Max` to `100`.
  - ■ Add thresholds: e.g., `40` (Yellow), `60` (Yellow), `50` (Green, between 40-60).
  - ■ Set Units to `Humidity -> Percent (0-100)`.
  - ■ Set `Value Options -> Show value` to `All series`.

- **Panel 5: Average Daily Temperature (Time Series)**
  - ■ **Visualization:** `Time series`
  - ■ **Query:**
  - ■ Code snippet

```
from(bucket: "your_bucket_name")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r._measurement == "sensor_measurements" and r._field == "temperature")
  |> aggregateWindow(every: 1d, fn: mean, createEmpty: false) // Daily average
  |> group(columns: ["sensor_id"])
  |> yield(name: "avg_daily_temperature")
```

- ■
- ■
- ■ **Panel Options:** Set Units to `Temperature -> Celsius`.

3. **Save Dashboard:** Click the `Save` icon at the top of the dashboard and give it a name.

---

8. Stopping the Services

To shut down your setup cleanly:

1. **Kafka Producer:** Go to its Command Prompt window and press `Ctrl+C`.
2. **Kafka Consumer:** Go to its Command Prompt window and press `Ctrl+C`.
3. **Kafka Broker:** Go to its Command Prompt window and press `Ctrl+C`.
4. **Zookeeper:** Go to its Command Prompt window and press `Ctrl+C`.
5. **InfluxDB:** Go to its Command Prompt window and press `Ctrl+C`.
6. **Grafana:** If running as a service, stop it via `services.msc`. If running from Command Prompt, press `Ctrl+C`.

This guide should provide a robust foundation for your future reference. Remember to keep your API tokens and passwords secure!

USING PROMETHEUS
Okay, absolutely! Here's the comprehensive documentation for your real-time monitoring stack using Kafka, Prometheus (with Pushgateway), and Grafana on Windows. This document is designed to be a self-contained reference, covering setup, configuration, code, queries, and troubleshooting.

Real-Time Sensor Data Monitoring Stack: Kafka -> Prometheus -> Grafana (Windows)

Table of Contents

---

1. Introduction

This document provides a detailed guide to setting up a real-time data monitoring pipeline on a Windows environment. The stack consists of:

- **Apache Kafka:** A distributed streaming platform used for publishing and subscribing to streams of records (our sensor data).

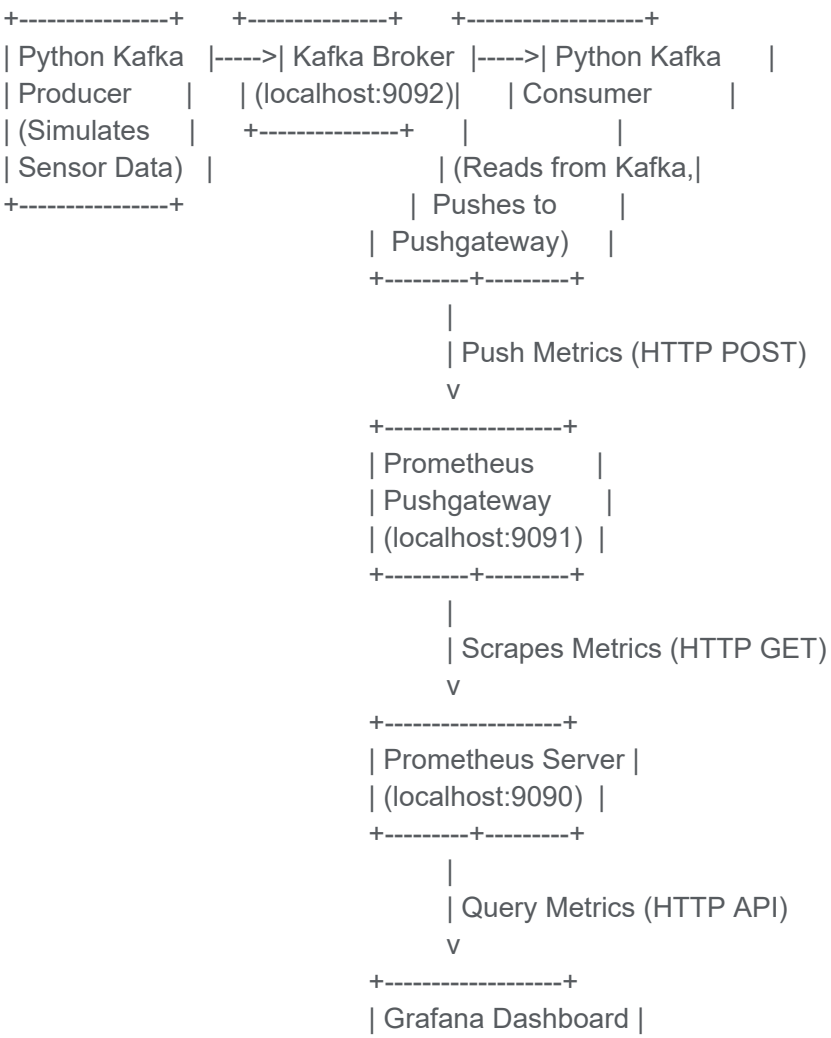
- **Prometheus:** An open-source monitoring system with a time-series database. Unlike InfluxDB, Prometheus primarily operates on a "pull" model, scraping metrics from configured endpoints.
- **Prometheus Pushgateway:** An intermediary service that allows ephemeral or batch jobs to push their metrics to it, which Prometheus then scrapes. This is essential because our Kafka consumer is a short-lived process relative to Prometheus's scrape cycles.
- **Grafana:** A leading open-source platform for analytics and interactive visualization. It will connect to Prometheus to query and display our sensor data on dynamic dashboards.

Our workflow will be:

Kafka Producer (Python) -> Kafka Broker -> Kafka Consumer (Python) -> Prometheus Pushgateway -> Prometheus Server -> Grafana Dashboard

---

2. System Architecture



```
                    | (localhost:3000)  |
                    +-------------------+
```

---

3. Prerequisites

Ensure these software components are installed and configured on your Windows machine before proceeding.

3.1. Java Development Kit (JDK)

Kafka requires Java.

- **Download:** Visit openjdk.java.net or search for "OpenJDK 17 download" and get the installer for Windows.
- **Installation:** Follow the installer instructions.
- **Environment Variable Configuration:**
    - Search for "Environment Variables" in Windows Start Menu and open "Edit the system environment variables".
    - Click "Environment Variables..."
    - Under "System variables", click "New...".
        - **Variable name:** `JAVA_HOME`
        - **Variable value:** `C:\Program Files\Java\jdk-17` (or your actual JDK installation path).
    - Find the `Path` variable under "System variables" and click "Edit...".
    - Click "New" and add `%JAVA_HOME%\bin`.
    - Click OK on all windows.
- **Verification:** Open a new Command Prompt and run:
- DOS

```
java -version
```

-
- You should see output showing your Java version.

3.2. Python

Python 3.8 or newer is recommended.

- **Download:** Visit python.org/downloads/windows/ and download the latest Windows installer.
- **Installation:** Run the installer. **Crucially, ensure you check "Add Python to PATH" during the installation process.** This makes `python` and `pip` commands available directly in your Command Prompt.
- **Verification:** Open a new Command Prompt and run:

- DOS

```
python --version
pip --version
```

-
- You should see their respective versions.

---

4. Component Setup & Configuration

General Rule: For each server component (Zookeeper, Kafka, Prometheus, Pushgateway, Grafana), open a **separate Command Prompt window** to run it. Do not close these windows as long as you want the services to remain running.

4.1. Apache Kafka (with Zookeeper)

Kafka relies on Zookeeper for metadata. We'll use a Kafka distribution that bundles Zookeeper for simplicity.

4.1.1. Download Kafka

1. Go to the Apache Kafka downloads page: https://kafka.apache.org/downloads
2. Download a stable binary release that includes Zookeeper (e.g., **Scala 2.13 - Kafka 3.6.1**). Look for a `.tgz` or `.zip` file.
3. Extract the downloaded archive (e.g., `kafka_2.13-3.6.1.tgz`) to a simple, root-level path like `C:\kafka`. This will create a directory structure, e.g., `C:\kafka\kafka_2.13-3.6.1`. We will refer to this as `%KAFKA_HOME%`.

4.1.2. Configure Kafka

1. Navigate to the Kafka `config` directory: `C:\kafka\kafka_2.13-3.6.1\config` (i.e., `%KAFKA_HOME%\config`).
2. **Edit `server.properties`:**
    - Open `server.properties` in a text editor (Notepad++, VS Code, etc.).
    - Find the line `log.dirs=/tmp/kafka-logs` and change it to a Windows-friendly path:
    - Properties

```
log.dirs=C:/kafka-logs
```

    -
    - (Create this folder `C:\kafka-logs` if it doesn't exist).
3. **Edit `zookeeper.properties`:**
    - Open `zookeeper.properties` in a text editor.
    - Find the line `dataDir=/tmp/zookeeper` and change it to a Windows-friendly path:

- Properties

```
dataDir=C:/zookeeper-data
```

-
- (Create this folder `C:\zookeeper-data` if it doesn't exist).

4.1.3. Start Zookeeper & Kafka Broker

1. **Start Zookeeper:**
   - Open a **new** Command Prompt.
   - Navigate to your Kafka installation directory:
   - DOS

```
cd C:\kafka\kafka_2.13-3.6.1
```

   -
   -
   - Execute the Zookeeper startup script:
   - DOS

```
.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
```

   -
   -
   - Wait for Zookeeper to start. You should see "binding to" and "Started" messages in the console output.
2. **Start Kafka Broker:**
   - Open a **new** Command Prompt.
   - Navigate to your Kafka installation directory:
   - DOS

```
cd C:\kafka\kafka_2.13-3.6.1
```

   -
   -
   - Execute the Kafka broker startup script:
   - DOS

.\bin\windows\kafka-server-start.bat .\config\server.properties

-
-
- Wait for Kafka to start. Look for "Kafka Server started" in the console output.

4.2. Prometheus Server

4.2.1. Download Prometheus

1. Go to the Prometheus download page: https://prometheus.io/download/
2. Download the latest stable release for **Windows (64-bit)** (e.g., `prometheus-2.x.x.windows-amd64.zip`).
3. Extract the ZIP file to a simple path, e.g., `C:\prometheus`. This will create a directory like `C:\prometheus\prometheus-2.x.x.windows-amd64`. We'll refer to this as `%PROMETHEUS_HOME%`.

4.2.2. Configure Prometheus (`prometheus.yml`)

1. Navigate to your Prometheus installation directory: `C:\prometheus\prometheus-2.x.x.windows-amd64` (i.e., `%PROMETHEUS_HOME%`).
2. Open the `prometheus.yml` file in a text editor.
3. Modify the `scrape_configs` section to include the Pushgateway as a target. This tells Prometheus where to find the metrics that your consumer pushes.
4. YAML

```
# my global config
global:
  scrape_interval: 15s # Prometheus scrapes targets every 15 seconds. You can reduce this for lower latency.
  evaluation_interval: 15s

# Alertmanager configuration (optional, for alerts)
alerting:
  alertmanagers:
    - static_configs:
        - targets:
          # - alertmanager:9093

# Load rules (optional, for recording rules or alerts)
rule_files:
  # - "first_rules.yml"

# A list of scrape configurations.
scrape_configs:
```

```
  # Prometheus scrapes its own metrics
  - job_name: "prometheus"
    static_configs:
      - targets: ["localhost:9090"]

  # Configuration for scraping metrics from the Pushgateway
  - job_name: "pushgateway"
    honor_labels: true # IMPORTANT: This preserves original labels (like sensor_id) from pushed
metrics
    static_configs:
      - targets: ["localhost:9091"] # Pushgateway's default port
```

4.2.3. Start Prometheus Server

1. Open a **new** Command Prompt.
2. Navigate to your Prometheus installation directory:
3. DOS

```
cd C:\prometheus\prometheus-2.x.x.windows-amd64
```

4.
5.
6. Start Prometheus, specifying the configuration file:
7. DOS

```
.\prometheus.exe --config.file=prometheus.yml
```

8.
9.
10. Wait for Prometheus to start. You can verify it's running by opening your web browser to http://localhost:9090. Go to Status -> Targets to ensure the pushgateway target is listed as UP.

4.3. Prometheus Pushgateway

4.3.1. Download Prometheus Pushgateway

1. Go to the Pushgateway releases page: https://github.com/prometheus/pushgateway/releases
2. Download the latest stable release for **Windows (64-bit)** (e.g., pushgateway-1.x.x.windows-amd64.zip).

3. Extract the ZIP file to a simple path, e.g., `C:\pushgateway`. This will create a directory like `C:\pushgateway\pushgateway-1.x.x.windows-amd64`. We'll refer to this as `%PUSHGATEWAY_HOME%`.

4.3.2. Start Prometheus Pushgateway

1. Open a **new** Command Prompt.
2. Navigate to your Pushgateway installation directory:
3. DOS

```
cd C:\pushgateway\pushgateway-1.x.x.windows-amd64
```

4.
5.
6. Start the Pushgateway executable:
7. DOS

```
.\pushgateway.exe
```

8.
9.
10. You should see logs indicating it's listening on `http://localhost:9091`. You can also verify this by navigating to `http://localhost:9091` in your web browser.

4.4. Grafana

4.4.1. Download & Install Grafana

1. Go to the Grafana downloads page: https://grafana.com/grafana/download
2. Select **Grafana Open Source**, then choose the latest **stable version (e.g., 12.0.2)**.
3. Select **Windows** as the platform.
4. Download the installer (usually an `.msi` file).
5. Run the installer and follow the on-screen prompts. The default installation path is typically `C:\Program Files\GrafanaLabs\grafana`.

4.4.2. Start Grafana

1. Grafana often installs as a Windows service and starts automatically. You can check the "Services" application (search for `services.msc` in Windows Start) and ensure the `Grafana` service is running.
2. Alternatively, you can manually start it from the command line (if it's not running as a service or you prefer manual control):
    - Open a **new** Command Prompt.
    - Navigate to the Grafana `bin` directory:
    - DOS

```
cd "C:\Program Files\GrafanaLabs\grafana\bin"
```

-
-
- Start Grafana:
- DOS

```
.\grafana-server.exe
```

-
-
- You should see Grafana logging messages in the console.

4.4.3. Access Grafana UI

1. Open your web browser and navigate to `http://localhost:3000`.
2. **Login:**
   - **Default Username:** `admin`
   - **Default Password:** `admin`
   - You will be prompted to change the password on your first login.

---

5. Python Application Code

These scripts will send and receive data. Ensure you have the necessary Python libraries installed.

5.1. Install Python Libraries

Open a **new** Command Prompt and run the following command to install the required libraries:

```
DOS

pip install kafka-python prometheus_client
```

5.2. Kafka Producer Script (`kafka_producer.py`)

This script simulates sensor data and publishes it to a Kafka topic.

1. Create a file named `kafka_producer.py` in a convenient directory (e.g., `C:\sensor_scripts`).
2. Paste the following code into the file:
3. Python

```
import time
import json
import random
from kafka import KafkaProducer

# Configuration
KAFKA_BOOTSTRAP_SERVERS = 'localhost:9092'
KAFKA_TOPIC = 'sensor_data_topic'

# Initialize Kafka Producer
producer = KafkaProducer(
    bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

print(f"Kafka Producer initialized for topic: {KAFKA_TOPIC}")

# Simulate sensor data
sensor_ids = ['sensor_A', 'sensor_B', 'sensor_C']

def generate_sensor_data(sensor_id):
    """Generates a dictionary with simulated sensor readings."""
    temperature = round(random.uniform(20.0, 30.0), 2)
    humidity = round(random.uniform(50.0, 60.0), 2)
    timestamp = int(time.time() * 1000) # Milliseconds Unix timestamp
    return {
        'sensor_id': sensor_id,
        'temperature': temperature,
        'humidity': humidity,
        'timestamp': timestamp
    }

try:
    while True:
        for sensor_id in sensor_ids:
            data = generate_sensor_data(sensor_id)
            # Send the data to Kafka
            producer.send(KAFKA_TOPIC, data)
            print(f"Sent to Kafka: {data}")
        time.sleep(2) # Send data every 2 seconds for all sensors

except KeyboardInterrupt:
    print("Stopping producer...")
finally:
    producer.close()
    print("Producer closed.")
```

4.
5.

5.3. Kafka Consumer to Prometheus Pushgateway Script (`kafka_to_prometheus.py`)

This script consumes messages from Kafka, converts them into Prometheus metrics, and pushes them to the Prometheus Pushgateway.

1. Create a file named `kafka_to_prometheus.py` in the same directory as your producer script.
2. Paste the following code into the file:
3. Python

```
import json
from kafka import KafkaConsumer
from prometheus_client import CollectorRegistry, Gauge, push_to_gateway

# Kafka Configuration
KAFKA_BOOTSTRAP_SERVERS = 'localhost:9092'
KAFKA_TOPIC = 'sensor_data_topic'
KAFKA_GROUP_ID = 'prometheus_consumer_group' # Consumer group ID

# Prometheus Pushgateway Configuration
PROMETHEUS_PUSHGATEWAY_URL = 'localhost:9091'
PROMETHEUS_JOB_NAME = 'sensor_data_job' # A name for this collection of metrics

# Initialize Prometheus Metrics Registry
# Use a custom registry to avoid default metrics if you only want yours
registry = CollectorRegistry()

# Define Prometheus Gauges for temperature and humidity
# Gauges are for values that can go up and down (like temperature)
# Labels ['sensor_id'] allow us to differentiate metrics from different sensors
temperature_gauge = Gauge(
    'sensor_temperature_celsius', # Metric name
    'Current temperature in Celsius from sensor', # Metric description
    ['sensor_id'], # Labels
    registry=registry # Associate with our custom registry
)

humidity_gauge = Gauge(
    'sensor_humidity_percent',
    'Current humidity in percent from sensor',
    ['sensor_id'],
```

```
    registry=registry
)

# Initialize Kafka Consumer
consumer = KafkaConsumer(
    KAFKA_TOPIC,
    bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,
    group_id=KAFKA_GROUP_ID,
    value_deserializer=lambda m: json.loads(m.decode('utf-8')),
    auto_offset_reset='earliest', # Start reading from the beginning if no offset is committed
    enable_auto_commit=True,
    consumer_timeout_ms=1000 # Stop after 1 second if no messages are available, helps with
graceful shutdown
)

print(f"Kafka Consumer initialized for topic: {KAFKA_TOPIC}, group: {KAFKA_GROUP_ID}")
print(f"Pushing metrics to Prometheus Pushgateway at: {PROMETHEUS_PUSHGATEWAY_URL},
job: {PROMETHEUS_JOB_NAME}")

try:
    for message in consumer:
        try:
            data = message.value
            print(f"Received from Kafka: {data}")

            sensor_id = data['sensor_id']
            temperature = data['temperature']
            humidity = data['humidity']

            # Set the gauge values for the specific sensor_id
            temperature_gauge.labels(sensor_id=sensor_id).set(temperature)
            humidity_gauge.labels(sensor_id=sensor_id).set(humidity)

            # Push all metrics currently in the registry to the Pushgateway
            # This updates the metrics for all sensors that have sent data recently
            # Prometheus will then scrape these from the Pushgateway
            push_to_gateway(PROMETHEUS_PUSHGATEWAY_URL,
job=PROMETHEUS_JOB_NAME, registry=registry)
            print(f"SUCCESS: Pushed metrics for sensor '{sensor_id}' to Pushgateway.")

        except Exception as e:
            print(f"Error processing message or pushing to Prometheus Pushgateway: {e}")
            print(f"Problematic data: {message.value}")

except KeyboardInterrupt:
    print("Stopping consumer...")
```

```
except Exception as e:
    print(f"An unexpected error occurred in the consumer: {e}")
finally:
    consumer.close()
    print("Consumer closed.")
    # The prometheus_client library handles its own lifecycle, no explicit client close needed.
```

---

6. Running the Entire Stack

Follow the order of starting components, using a **separate Command Prompt window for each** and keeping them open.

1. **Start Zookeeper:** (from Section 4.1.3)
2. DOS

```
cd C:\kafka\kafka_2.13-3.6.1
.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
```

3.
4.
5. **Start Kafka Broker:** (from Section 4.1.3)
6. DOS

```
cd C:\kafka\kafka_2.13-3.6.1
.\bin\windows\kafka-server-start.bat .\config\server.properties
```

7.
8.
9. **Start Prometheus Pushgateway:** (from Section 4.3.2)
10. DOS

```
cd C:\pushgateway\pushgateway-1.x.x.windows-amd64
.\pushgateway.exe
```

11.
12.
13. **Start Prometheus Server:** (from Section 4.2.3)
14. DOS

```
cd C:\prometheus\prometheus-2.x.x.windows-amd64
```

```
.\prometheus.exe --config.file=prometheus.yml
```

15.
    - Verify Prometheus UI at `http://localhost:9090`. Check `Status -> Targets`. The `pushgateway` target should eventually show `UP`.
16. **Start Grafana:** (from Section 4.4.2)
    - If running as a service, it might already be started.
    - Manual start:
    - DOS

```
cd "C:\Program Files\GrafanaLabs\grafana\bin"
.\grafana-server.exe
```

    -
    -
    - Verify Grafana UI at `http://localhost:3000`.
17. **Start Kafka Producer:** (from Section 5.2)
    - Navigate to your `sensor_scripts` directory.
18. DOS

```
cd C:\sensor_scripts
python kafka_producer.py
```

19.
    - You should see "Sent to Kafka: {...}" messages.
20. **Start Kafka Consumer:** (from Section 5.3)
    - Navigate to your `sensor_scripts` directory.
21. DOS

```
cd C:\sensor_scripts
python kafka_to_prometheus.py
```

22.
    - You should see "Received from Kafka: {...}" and "SUCCESS: Pushed metrics..." messages.

---

7. Grafana Dashboard Setup

Now that all services are running and data is flowing to Prometheus via Pushgateway, let's configure Grafana to visualize it.

7.1. Add Prometheus Data Source

1. **Access Grafana UI:** Open your web browser and go to http://localhost:3000. Log in if prompted.
2. **Navigate to Data Sources:**
    - In the left-hand menu, hover over the **gear icon** (Configuration).
    - Click on **Data sources**.
3. **Add Prometheus Data Source:**
    - Click the **Add data source** button.
    - Search for and select **Prometheus**.
4. **Configure Prometheus Data Source:**
    - **Name:** Prometheus Local (or any descriptive name like SensorDataPrometheus)
    - **HTTP:**
        - **URL:** http://localhost:9090 (This is the Prometheus server's address, **not** the Pushgateway's)
    - Keep other settings as default for now.
    - Click the **Save & Test** button at the bottom. You should see a green "Data source is working" message.

7.2. Build Dashboard Panels (PromQL Queries)

Now, create a new Grafana dashboard and add panels using the Prometheus data source.

1. **Create New Dashboard:**
    - In the left menu, hover over the **+ icon** (Create).
    - Click **Dashboard**.
    - Click **Add new panel**.
2. **Common Panel Types & PromQL Queries:**
    - **Key Metrics:**
        - sensor_temperature_celsius: The name of our temperature metric.
        - sensor_humidity_percent: The name of our humidity metric.
        - sensor_id: The label we attached to differentiate sensors.
    - ### Panel 1: Temperature Over Time (All Sensors)
        - **Visualization:** Time series
        - **Query (PromQL):**
        - Code snippet

sensor_temperature_celsius

-
    - **Explanation:** This query selects all data points for the sensor_temperature_celsius metric. Since sensor_id is a label on this metric, Grafana will automatically display a separate line for each unique sensor_id (e.g., sensor_A, sensor_B, sensor_C).

        - **Panel Options:**
            - Set **Units** to `Temperature -> Celsius`.
            - Customize `Legend`, `Tooltip`, `Axes` as desired.
    - **Panel 2: Humidity Over Time (All Sensors)**
        - **Visualization:** `Time series`
        - **Query (PromQL):**
        - Code snippet

sensor_humidity_percent

        -
            - **Explanation:** Similar to temperature, this shows humidity trends for all sensors, with a line per `sensor_id`.
        - **Panel Options:**
            - Set **Units** to `Humidity -> Percent (0-100)`.
    - **Panel 3: Current Temperature (Gauge or Stat Panel)**
        - **Visualization:** `Stat` or `Gauge`
        - **Query (PromQL - for a specific sensor, e.g., sensor_A):**
        - Code snippet

sensor_temperature_celsius{sensor_id="sensor_A"}

        -
            - **Explanation:** Filters the temperature metric to show only the value from `sensor_A`.
        - **Panel Options (for Gauge):**
            - Set `Min` to `0` and `Max` to `50` (or appropriate range).
            - Add **Thresholds** (e.g., Green: `20`, Yellow: `25`, Red: `30`) for visual cues.
            - Set **Units** to `Temperature -> Celsius`.
            - **Tip:** Duplicate this panel for `sensor_B` and `sensor_C` by just changing the `sensor_id` label in the query. For a single gauge showing the *average* of all sensors, use `avg(sensor_temperature_celsius)`.
    - **Panel 4: Current Humidity (Gauge or Stat Panel)**
        - **Visualization:** `Stat` or `Gauge`
        - **Query (PromQL - for a specific sensor, e.g., sensor_B):**
        - Code snippet

sensor_humidity_percent{sensor_id="sensor_B"}

-
-
- **Panel Options (for Gauge):**
  - Set Min to 0 and Max to 100.
  - Add **Thresholds** (e.g., Green: 45-55, Yellow: 30-45 & 55-70, Red: 0-30 & 70-100).
  - Set **Units** to Humidity -> Percent (0-100).

Panel 5: Average Temperature Across All Sensors (Time Series)

- **Visualization:** Time series
- **Query (PromQL):**
- Code snippet

avg by (sensor_id) (sensor_temperature_celsius)

-
  - **Explanation:** This will calculate the average temperature for each sensor_id over the selected time range in Grafana. If you want a single average line *across all sensors combined*, use avg(sensor_temperature_celsius).
- **Panel Options:** Set Units to Temperature -> Celsius.

Panel 6: Most Recent Scrape Time for Metrics (Table Panel)

- **Visualization:** Table
- **Query (PromQL):**
- Code snippet

timestamp(sensor_temperature_celsius)

-
  - **Explanation:** This query returns the Unix timestamp when the sensor_temperature_celsius metric was last scraped by Prometheus. Useful for confirming data freshness.
- **Panel Options:**
  - Go to the **Transformations** tab: Add a transformation Organize fields and keep only Value and Time fields (or sensor_id).
  - Go to the **Field options** tab: Select Value field, set Unit to Misc -> Unix timestamp (s).
  - Go to Overrides tab: Add override for Value. Choose Unit -> Datetime (MM/DD HH:mm:ss) or desired format.

3. **Save Dashboard:** Once you're happy with your panels, click the Save icon (floppy disk) at the top of the dashboard and give it a name (e.g., "Sensor Monitoring Dashboard").

---

8. Stopping the Services

To shut down your entire monitoring stack gracefully, close the Command Prompt windows in the **reverse order** of how you started them:

1. **Kafka Consumer:** Go to its Command Prompt window and press Ctrl+C.
2. **Kafka Producer:** Go to its Command Prompt window and press Ctrl+C.
3. **Grafana:** If running as a Windows service, stop it via the services.msc application. If running manually from a Command Prompt, press Ctrl+C.
4. **Prometheus Server:** Go to its Command Prompt window and press Ctrl+C.
5. **Prometheus Pushgateway:** Go to its Command Prompt window and press Ctrl+C.
6. **Kafka Broker:** Go to its Command Prompt window and press Ctrl+C.
7. **Zookeeper:** Go to its Command Prompt window and press Ctrl+C.

---

9. Troubleshooting & Important Notes

- **Separate Command Prompts:** Always ensure each service is running in its own dedicated Command Prompt window. This prevents conflicts and allows for independent control.
- **Case Sensitivity:** Prometheus metric names and labels are case-sensitive (e.g., sensor_temperature_celsius vs Sensor_Temperature_Celsius). Ensure your Python code and PromQL queries use identical casing.
- **Prometheus Scrape Interval:** The scrape_interval in prometheus.yml directly impacts how "real-time" your data appears in Grafana. A 15-second interval means metrics might sit on Pushgateway for up to 15 seconds before Prometheus scrapes them. You can reduce this to 5s or 1s for faster updates, but be mindful of resource usage.
- **Pushgateway Ephemeral Metrics:** By default, Prometheus Pushgateway will keep metrics even after Prometheus has scraped them. If your job_name and grouping_key are unique per push, metrics can accumulate. For continuous sensor data, this is generally fine as the values are updated. For truly ephemeral metrics, you might need to use the Pushgateway's API to delete metrics after a push (though not typically needed for this sensor use case).
- **Prometheus UI (localhost:9090):** Use this UI to debug your metrics directly. You can enter PromQL queries to see raw data and check the Status -> Targets page to confirm Prometheus is scraping the Pushgateway.
- **Grafana Time Range:** Always double-check the time range selector in the top-right corner of your Grafana dashboard. If it's set to "Last 5 minutes" and your data is older, you won't see anything.

- **System Resources:** Running all these services locally can consume a fair amount of RAM and CPU. Monitor your system resources if you encounter performance issues.
- **`honor_labels: true`:** This setting in `prometheus.yml` under the `pushgateway` job is crucial. It tells Prometheus to respect the labels (like `sensor_id`) that your consumer pushes to the Pushgateway, rather than trying to overwrite them. Without it, your `sensor_id` label might not appear correctly.

COMPLETE LOG MONITORING

Yes, absolutely! The combination of **Filebeat, Logstash, Loki, and Grafana** is a very popular and powerful stack for log monitoring and visualization. This is often referred to as the "LFG stack" (Loki, Grafana) or "ELK with Loki" if you're replacing Elasticsearch with Loki.

Here's why this setup is often preferred and how it would work on Windows without Docker:

Architecture:

1. **Log Source:** Your Windows system generates logs (event logs, application logs, file-based logs).
2. **Filebeat:** Filebeat (an Elastic Beat, like Winlogbeat) collects logs from various sources (files, event logs, custom inputs). It's designed to be lightweight and efficient for shipping data.
3. **Logstash:** Logstash is an open-source data collection pipeline that has a vast ecosystem of plugins. It can:
    - Ingest data from Filebeat.
    - Process and transform logs (parsing, filtering, enriching, anonymizing).
    - Output processed logs to various destinations, including Loki.
4. **Loki:** Grafana Loki is a horizontally scalable, highly available, multi-tenant log aggregation system. Unlike Elasticsearch, Loki indexes *only* metadata (labels) about log streams, not the full log content. This makes it very cost-effective and efficient for storing large volumes of logs.
5. **Grafana:** Grafana connects to Loki as a data source and visualizes the logs. Loki's query language, LogQL (inspired by Prometheus's PromQL), allows you to filter and aggregate logs based on labels and perform simple text searches.

Advantages of this setup over Kafka-only for logs:

- **Loki's Log-Specific Design:** Loki is purpose-built for logs. It's designed to be "like Prometheus, but for logs," meaning it scales well for logs by indexing only labels, not the full text. This makes it more resource-efficient for log storage compared to storing raw logs in Kafka topics indefinitely or trying to query them directly.
- **Logstash for Transformation:** Logstash is incredibly powerful for transforming and enriching logs before they reach Loki. You can parse complex log formats (e.g., using Grok filters), add geographical data, remove sensitive information, or normalize fields.

This is harder to do purely with Winlogbeat sending directly to Kafka without a separate processing layer.
- **Grafana's Native Loki Integration:** Grafana has first-class support for Loki, including the "Logs" panel specifically designed to display log streams, and LogQL for powerful querying.
- **Simpler Querying in Grafana:** While a Kafka plugin exists for Grafana, querying raw logs in Kafka can be limited. Loki provides a much richer and more efficient querying experience with LogQL.
- **No Kafka for Long-Term Storage:** Kafka is a streaming platform, not a long-term data store. While you can configure long retention, it's generally not ideal for storing all your historical log data. Loki is designed for this.

How to Perform Complete Real-Time Log Monitoring and Visualization using Filebeat, Logstash, Loki, and Grafana on Windows (without Docker):

This will be a more involved setup than the Kafka one due to the additional components.

Prerequisites (Repeat from previous answer, plus additions):

- **Java Development Kit (JDK):** Logstash requires Java. Ensure you have JDK 11 or higher installed.
    - Set `JAVA_HOME` environment variable.
    - Add `%JAVA_HOME%\bin` to `Path`.
- **Windows Administrator Privileges:** For service installation and directory permissions.

---

1. Install and Configure Grafana

(Same as before, as it's the visualization layer)

1. **Download Grafana MSI:** https://grafana.com/grafana/download
2. **Install Grafana:** Run the MSI, follow prompts. Default path: `C:\Program Files\GrafanaLabs\grafana`.
3. **Access Grafana:** `http://localhost:3000` (admin/admin).

---

2. Install and Configure Loki

Loki is distributed as a single binary for local setups.

1. **Download Loki:**
    - Go to the Grafana Loki releases page: https://grafana.com/oss/loki/

    - Scroll down to "Download" and find the latest `loki-windows-amd64.zip` (or similar for your architecture).
2. **Extract Loki:**
    - Create a dedicated directory, e.g., `C:\Loki`.
    - Extract the downloaded zip file into this directory. You should see `loki-windows-amd64.exe`.
3. **Create Loki Configuration File:**
    - In `C:\Loki`, create a file named `loki-local-config.yaml`.
    - Paste the following basic configuration. This runs Loki in "monolithic" mode, suitable for a single-node setup.
    - YAML

```yaml
auth_enabled: false

server:
  http_listen_port: 3100
  grpc_listen_port: 9095

common:
  path_prefix: C:\Loki\data
  replication_factor: 1
  ring:
    instance_addr: 127.0.0.1
    kvstore:
      store: inmemory
  storage:
    filesystem:
      directory: C:\Loki\data\chunks
    boltdb_shipper:
```

```
    active_index_directory: C:\Loki\data\boltdb-shipper-active
    cache_location: C:\Loki\data\boltdb-shipper-cache
    resync_interval: 5s
  tsdb:
    dir: C:\Loki\data\tsdb

query_range:
  align_queries_with_step: true
  cache_results: true

schema_config:
  configs:
    - from: 2020-10-24 # Date when you start collecting logs
      store: boltdb-shipper
      object_store: filesystem
      schema: v11
      period: 24h
      index:
        prefix: index_
        period: 24h

# Disable all ingesters that are not needed for single binary mode
memberlist:
  join_members: []
```

```
# Remove the following sections if you plan to use a persistent key-value store instead of inmemory
# table_manager:
#   retention_deletes_enabled: true
#   retention_period: 336h # 14 days, adjust as needed

compactor:
  working_directory: C:\Loki\data\compactor
  compaction_interval: 10m
```

-
-
- **Important:** Create the directories specified in the `common.path_prefix`, `filesystem.directory`, `boltdb_shipper.*`, `tsdb.dir`, and `compactor.working_directory` (e.g., `C:\Loki\data`, `C:\Loki\data\chunks`, etc.).
- **Retention:** The example configuration sets up some basic storage. For production, you'd likely use S3, GCS, or MinIO for object storage and BoltDB Shipper for indexing. For this local setup, it uses the filesystem.

4. **Start Loki:**
   - Open a **new Command Prompt as Administrator**.
   - Navigate to your Loki directory:
   - Bash

```
cd C:\Loki
```

-
-
- Start Loki:
- Bash

```
.\loki-windows-amd64.exe --config.file=loki-local-config.yaml
```

-
- Keep this window open. You should see Loki starting up.

---

3. Install and Configure Logstash

Logstash will receive logs from Filebeat and send them to Loki.

1. **Download Logstash:**
    - Go to the Elastic Logstash downloads page: https://www.elastic.co/downloads/logstash
    - Download the latest stable version for Windows (e.g., `logstash-8.x.x-windows-x86_64.zip`).
2. **Extract Logstash:**
    - Create a dedicated directory, e.g., `C:\Logstash`.
    - Extract the downloaded zip file into this directory.
3. **Install Loki Output Plugin for Logstash:**
    - Open a **new Command Prompt as Administrator**.
    - Navigate to your Logstash directory:
    - Bash

```
cd C:\Logstash
```

    -
    -
    - Install the `logstash-output-loki` plugin:
    - Bash

```
.\bin\logstash-plugin install logstash-output-loki
```

    -
    - This will download and install the plugin.
4. **Create Logstash Configuration File:**
    - Navigate to `C:\Logstash\config`.
    - Create a file named `logstash.conf`.
    - Paste the following configuration:
    - Code snippet

```
input {
  beats {
    port => 5044 # Port Filebeat will connect to
```

```
  }
}

filter {
  # Example filter: parse Windows Event Logs if Winlogbeat sends them
  # If using Filebeat for text files, you'll need different filters (e.g., grok, json)
  if [agent][type] == "winlogbeat" {
    # Standard Winlogbeat fields are usually already well-structured
    # You might want to add/remove fields or perform specific transformations
    mutate {
      add_field => { "log_source_host" => "%{host][name]}" }
      remove_field => [ "[@version]", "ecs", "log", "agent", "input", "host", "event", "winlog", "tags" ] # Example fields to remove
    }
  }
  # Example: For generic file logs if Filebeat sends them
  if [message] =~ /^(\{.*\}|\[.*\])$/ {
      json {
          source => "message"
          target => "json_parsed" # Parse JSON messages into a nested field
      }
      mutate {
          remove_field => [ "message" ] # Remove original message if parsed
      }
  }
}
```

```
output {
  loki {
    url => "http://localhost:3100/loki/api/v1/push" # Your Loki instance URL
    # Labels for Loki are crucial for efficient querying.
    # You must define labels that are relatively low-cardinality.
    # Avoid using fields that change frequently (like full log messages).
    # Common labels: job, host, application, level
    labels => {
      "job" => "filebeat_logs"
      "host" => "%{[log_source_host]}"
      "level" => "%{[log][level]}" # Assuming Winlogbeat provides log.level
      "channel" => "%{[event][provider]}" # For Windows event logs
    }
    # The message field to send to Loki. Default is "@message" or "message".
    # If you parsed JSON, you might want to send the original message or a specific parsed field.
    # message_field => "message" # or "json_parsed.original_message" if you structured it
  }
  # For debugging, you can also output to stdout
  # stdout { codec => rubydebug }
}
```

-
-
- **Crucial:** Pay close attention to the `filter` and `labels` sections.
  - **Filters:** Adjust the `filter` section based on the structure of your logs. If you're collecting plain text files, you'll need `grok` or `dissect` filters to extract meaningful fields. If you're using Winlogbeat for event logs, many fields will already be structured.

        - **Labels:** The `labels` in the Loki output are vital for efficient querying in Grafana. Loki indexes these labels, not the full log content.
            - `job`: A static identifier for the source of logs.
            - `host`: The hostname of the machine generating logs.
            - `level`: The log level (info, warn, error, etc.).
            - `channel` or `application`: To identify the specific application or log channel.
            - **Avoid high-cardinality labels:** Do *not* use fields that are unique to every log line (e.g., timestamps, full messages, unique IDs) as labels, as this will defeat Loki's efficiency and consume excessive resources. Use filters to extract low-cardinality fields for labels.
5. **Start Logstash:**
    - Open **another new Command Prompt as Administrator**.
    - Navigate to your Logstash directory:
    - Bash

```
cd C:\Logstash
```

    -
    -
    - Start Logstash with your configuration:
    - Bash

```
.\bin\logstash.bat -f .\config\logstash.conf
```

    -
    - Keep this window open. You should see Logstash starting up and listening for Beats connections.

---

4. Install and Configure Filebeat

Filebeat will collect logs and send them to Logstash.

1. **Download Filebeat:**
    - Go to the Elastic Beats download page: https://www.elastic.co/downloads/beats/filebeat
    - Download the latest stable version for Windows (e.g., `filebeat-8.x.x-windows-x86_64.zip`).
2. **Extract Filebeat:**
    - Create a dedicated directory, e.g., `C:\Program Files\Filebeat`.

- Extract the downloaded archive into this directory.

3. **Configure Filebeat:**
   - Navigate to `C:\Program Files\Filebeat`.
   - Open `filebeat.yml` in a text editor.
   - **Configure Inputs:** Comment out or remove default inputs and add your specific log inputs.
     - **For Windows Event Logs (similar to Winlogbeat):**
     - YAML

```yaml
filebeat.inputs:

  - type: winlog

    event_logs:

      - name: Application

      - name: System

      - name: Security

    #processors:

    #  - add_fields:

    #      target: ''

    #      fields:

    #        log_source_host: 'my_windows_server_1' # Custom field to identify source
```

     -
     -
     - **For File-based Logs (e.g., `C:\ProgramData\MyApp\logs\*.log`):**
     - YAML

```yaml
filebeat.inputs:

  - type: filestream

    id: my-app-logs
```

```
  paths:
    - C:\ProgramData\MyApp\logs\*.log
  encoding: utf-8
  #fields: # Add custom fields that Logstash can use for labels/parsing
  #  application: my_windows_app
  #  log_type: application_log
  #processors:
  #  - add_fields:
  #      target: ''
  #      fields:
  #        log_source_host: 'my_windows_server_1'
```

        -
        - You can have multiple inputs. Make sure to adjust paths and add any custom fields you need for identification or parsing in Logstash.
    - **Disable Elasticsearch Output:** Comment out the `output.elasticsearch` section.
    - **Configure Logstash Output:** Uncomment and configure the `output.logstash` section.
    - YAML

```
output.logstash:
  hosts: ["localhost:5044"] # Your Logstash server address and Beats input port
```

    -
    -
    - Save `filebeat.yml`.
4. **Install Filebeat as a Windows Service:**
    - Open **PowerShell as Administrator**.
    - Navigate to your Filebeat installation directory:
    - PowerShell

```
cd 'C:\Program Files\Filebeat'
```

-
-
- Install the Filebeat service:
- PowerShell

```
.\install-service-filebeat.ps1
```

-
- (If execution policy error, run `Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser` first).
- Start the Filebeat service:
- PowerShell

```
Start-Service Filebeat
```

-
-
- Check status:
- PowerShell

```
Get-Service Filebeat
```

-
- You should see "Running".

Now, Filebeat is sending your logs to Logstash, Logstash is processing them and sending them to Loki.

---

5. Integrate Loki with Grafana

1. **Add Loki Data Source in Grafana:**
   - Log in to Grafana (`http://localhost:3000`).
   - On the left sidebar, hover over "Connections" (plug icon), then click "Data sources".
   - Click "Add data source".
   - Select "Loki".
   - **Settings:**

        - **Name:** Loki (or any descriptive name)
        - **HTTP:**
            - **URL:** http://localhost:3100 (Your Loki instance URL)
        - **Derived Fields (Optional but Recommended):** This allows you to link directly from log lines to traces (if you also use tracing) or other dashboards. Not strictly necessary for basic log viewing.
    - Click "Save & test". You should see "Data source is working".
2. **Create a Grafana Dashboard for Logs:**
    - On the left sidebar, hover over the "Dashboards" icon, then click "New dashboard".
    - Click "Add a new panel".
    - In the "Query" tab:
        - **Data source:** Select Loki (the Loki data source you just created).
        - **LogQL Query:** This is where you'll query your logs.
            - To see all logs: {job="filebeat_logs"}
            - To filter by host: {job="filebeat_logs", host="my_windows_server_1"}
            - To filter by log level and search for text: {job="filebeat_logs", level="error"} |= "failed"
            - You can use the "Label browser" button to see available labels from your logs.
        - **Visualization:** Choose the "Logs" visualization (under "Visualization" tab). This is optimized for displaying log lines.
    - Adjust the "Time range" in the top right to "Last 5 minutes" or "Last 15 minutes" to see real-time data. Enable "Refresh dashboard" at a suitable interval (e.g., 5s).
    - You can create multiple panels to show log rates, specific error counts (using metric queries in LogQL, e.g., count_over_time({job="filebeat_logs", level="error"}[5m])), etc.

Directory Structure Summary:

C:\

├── GrafanaLabs\

│ └── grafana\

│ ├── bin\

│ ├── conf\

│ ├── data\

│ └── (other Grafana files)

```
│
├── Loki\
│   ├── loki-windows-amd64.exe
│   ├── loki-local-config.yaml
│   └── data\
│       ├── boltdb-shipper-active\
│       ├── boltdb-shipper-cache\
│       ├── chunks\
│       ├── compactor\
│       └── tsdb\
│
├── Logstash\
│   ├── bin\
│   ├── config\
│   │   └── logstash.conf
│   └── (other Logstash files)
│
└── Program Files\
    └── Filebeat\
        ├── filebeat.yml
        └── (other Filebeat files)
```

Commands Summary:

Grafana:

- **Access Grafana:** http://localhost:3000

Loki:

- Start Loki (Command Prompt Admin):
  cd C:\Loki
  .\loki-windows-amd64.exe --config.file=loki-local-config.yaml

Logstash:

- Install Loki Plugin (Command Prompt Admin):
  cd C:\Logstash
  .\bin\logstash-plugin install logstash-output-loki
- Start Logstash (Command Prompt Admin):
  cd C:\Logstash
  .\bin\logstash.bat -f .\config\logstash.conf

Filebeat:

- Install Filebeat Service (PowerShell Admin):
  cd 'C:\Program Files\Filebeat'
  .\install-service-filebeat.ps1
- Start Filebeat Service (PowerShell Admin):
  Start-Service Filebeat
- Stop Filebeat Service (PowerShell Admin):
  Stop-Service Filebeat
- Get Filebeat Service Status (PowerShell Admin):
  Get-Service Filebeat

Important Notes for this Stack:

- **Logstash Resource Usage:** Logstash can be a resource-intensive component, especially with complex filters or high log volumes. Monitor its CPU and memory usage.
- **Loki Data Persistence:** The `C:\Loki\data` directory in the `loki-local-config.yaml` is where Loki stores its index and chunk data. Ensure this path is correct and has enough space. For production, you'd typically use cloud object storage (S3, GCS, Azure Blob Storage) for chunks and a separate key-value store (like DynamoDB, Cassandra, or BoltDB with S3 for backups) for the index, but for a local setup, filesystem storage is fine.
- **Label Cardinality:** This is the most critical concept with Loki. Always strive to keep your labels low-cardinality. If you have too many unique label combinations, Loki's performance will suffer, and storage will increase dramatically. Use Logstash to filter and extract meaningful, low-cardinality labels.
- **Error Handling:** Check the logs for Filebeat, Logstash, and Loki for any errors.
- **Scalability:** For a serious production environment, you'd move to a distributed Loki setup (using object storage and potentially multiple Loki components like ingester, querier, distributor) and run Filebeat/Logstash on dedicated machines.

This setup provides a more robust and log-specific solution compared to directly using Kafka for log storage and querying.

Using docker:

You're right to ask about Docker! Using Docker is often the preferred way to set up stacks like Filebeat, Logstash, Loki, and Grafana (FLLG) because it simplifies installation, configuration, and dependency management significantly. It also provides isolation and portability.

Here's a comprehensive guide to setting up the FLLG stack on Windows using Docker Compose:

Overall Architecture (with Docker):

1. **Log Source:** Your Windows host system (or other containers if you log from them).
2. **Filebeat (Container):** Runs in a Docker container, configured to collect logs from your Windows host and send them to the Logstash container.
3. **Logstash (Container):** Runs in a Docker container, receives logs from Filebeat, processes them, and sends them to the Loki container.
4. **Loki (Container):** Runs in a Docker container, receives and stores logs from Logstash.
5. **Grafana (Container):** Runs in a Docker container, connects to Loki to visualize the logs.

Prerequisites:

1. **Docker Desktop for Windows:** This is the primary tool you need. It includes Docker Engine, Docker CLI, and Docker Compose.
    - **System Requirements:** Ensure your Windows version supports WSL 2 (Windows Subsystem for Linux 2) for optimal performance. Windows 10/11 Pro, Enterprise, or Education are generally recommended.
    - **Installation:**
        - Download Docker Desktop from the official Docker website: https://www.docker.com/products/docker-desktop
        - Run the installer. During installation, ensure the "Use WSL 2 instead of Hyper-V" option is selected if your system supports it (recommended).
        - Follow the on-screen instructions. You might be prompted to enable WSL 2 features or restart your computer.
        - After installation, start Docker Desktop. You should see the Docker whale icon in your system tray.
    - **Verification:** Open PowerShell or Command Prompt and run:
    - Bash

```
docker --version
```

```
docker compose version # Note: For newer Docker Desktop, 'docker compose' is used instead of 'docker-compose'
```

    -
    - You should see version numbers, confirming Docker and Docker Compose are installed.
2. **Windows Administrator Privileges:** Still needed for installing Docker Desktop and managing some host-level configurations.

---

1. Project Setup and Directory Structure

Create a main project directory that will hold all your Docker-related files and configurations.

```
C:\fllg-stack\
├── docker-compose.yml
├── grafana\
│   └── provisioning\
│       ├── datasources\
│       │   └── loki.yml
│       └── dashboards\
│           └── default.yml
├── loki\
│   └── loki-local-config.yaml
├── logstash\
│   ├── pipeline\
│   │   └── logstash.conf
│   └── config\
│       └── logstash.yml
├── filebeat\
│   ├── filebeat.yml
```

│ └── filebeat.template.json (optional, for custom index templates if using Elasticsearch/Kibana alongside)

---

2. Create Configuration Files

We'll start by defining the configuration files for each service.

2.1. loki/loki-local-config.yaml

This configuration is similar to the standalone Loki setup, but paths are relative to the *container's* filesystem.

YAML

```yaml
auth_enabled: false

server:
  http_listen_port: 3100
  grpc_listen_port: 9095

common:
  path_prefix: /loki # Base path inside the container for Loki data
  replication_factor: 1
  ring:
    instance_addr: 127.0.0.1
    kvstore:
      store: inmemory # In-memory KV store for simplicity in Docker
  storage:
    filesystem:
      directory: /loki/chunks # Chunks will be stored here
```

```
    boltdb_shipper:
      active_index_directory: /loki/boltdb-shipper-active
      cache_location: /loki/boltdb-shipper-cache
      resync_interval: 5s
    tsdb:
      dir: /loki/tsdb

query_range:
  align_queries_with_step: true
  cache_results: true

schema_config:
  configs:
    - from: 2020-10-24
      store: boltdb-shipper
      object_store: filesystem
      schema: v11
      period: 24h
      index:
        prefix: index_
        period: 24h

memberlist:
  join_members: []
```

```
compactor:
  working_directory: /loki/compactor
  compaction_interval: 10m
```

- **Important:** The `path_prefix` and other `directory` settings are paths *inside the Docker container*. We'll map these to host volumes in `docker-compose.yml` to ensure data persistence.

2.2. `logstash/pipeline/logstash.conf`

This is your Logstash pipeline configuration. Notice the `hosts` in the `loki` output are now the *service names* defined in `docker-compose.yml`.

```
Code snippet
input {
  beats {
    port => 5044
    host => "0.0.0.0" # Listen on all interfaces inside the container
  }
}

filter {
  # Example: For Windows Event Logs collected by Filebeat
  if [agent][type] == "filebeat" and [event][provider] { # Using event.provider to identify Windows Event Logs
    mutate {
      add_field => { "log_source_type" => "windows_event_log" }
      add_field => { "log_level" => "%{[log][level]}" } # Extract log level if available
      # Remove fields that might be high-cardinality or not needed as labels
      remove_field => [ "[@version]", "ecs", "agent", "input", "host", "event", "tags", "[log][level]" ]
```

```
    }
  }
  # Example: For generic file logs collected by Filebeat
  else if [message] =~ /^(\{.*\}|\[.*\])$/ {
      json {
          source => "message"
          target => "json_parsed"
      }
      mutate {
          remove_field => [ "message" ]
          add_field => { "log_source_type" => "application_log" }
      }
  }
  # Add a default log_level if not explicitly set by Filebeat or parsing
  if ![log_level] {
    mutate {
      add_field => { "log_level" => "info" }
    }
  }
  # Add hostname from Filebeat for labeling
  mutate {
    add_field => { "source_host" => "%{[beat][hostname]}" }
  }
}
```

```
output {
  loki {
    url => "http://loki:3100/loki/api/v1/push" # Use the service name 'loki' defined in docker-compose.yml
    labels => {
      "job" => "filebeat_logs"
      "host" => "%{[source_host]}"
      "level" => "%{[log_level]}"
      "source_type" => "%{[log_source_type]}"
      "channel" => "%{[event][provider]}" # For Windows event logs, if available
    }
    # message_field => "message" # If you removed 'message' and want to use 'json_parsed', specify it here
  }
  # stdout { codec => rubydebug } # Uncomment for debugging
}
```

- **Important:**
  - `host => "0.0.0.0"` in `beats` input means Logstash will listen on all network interfaces *inside its container*.
  - `url => "http://loki:3100/loki/api/v1/push"`: We use the Docker service name `loki` for inter-container communication. Docker's internal DNS handles resolving `loki` to the correct IP address of the Loki container.
  - The `filter` section is crucial and will need adjustment based on the *actual* format of the logs you are collecting from your Windows host.

2.3. `logstash/config/logstash.yml`

Basic Logstash settings.

```
YAML
```

```
# You can add global Logstash settings here if needed

# For instance, to change the pipeline file location if it's not default

path.config: /usr/share/logstash/pipeline/logstash.conf
```

2.4. filebeat/filebeat.yml

This Filebeat configuration will be mounted into the Filebeat container. It needs to read logs from the *host* machine (your Windows machine).

```
YAML

filebeat.inputs:

  - type: winlog

    event_logs:

      - name: Application

      - name: System

      - name: Security

    processors:

      - add_fields:

          target: ''

          fields:

            log_source_identifier: 'windows-docker-host' # Unique identifier for this host


  # Example for collecting a specific file log from the Windows host

  # - type: filestream

  #   id: my-windows-app-logs

  #   paths:

  #     - C:\path\to\your\app\logs\*.log # This path MUST be accessible by the Filebeat container via a volume mount
```

```
  #   fields:
  #     application: 'my_windows_app'
  #     log_type: 'custom_app_log'
  #   processors:
  #     - add_fields:
  #         target: ''
  #         fields:
  #           log_source_identifier: 'windows-docker-host'


output.logstash:
  hosts: ["logstash:5044"] # Use the service name 'logstash' defined in docker-compose.yml
  loadbalance: true


logging.level: info
logging.to_files: false # Or true if you want logs inside the container for debugging
logging.metrics.enabled: true
```

- **Important:**
  - `hosts: ["logstash:5044"]`: Filebeat will send logs to the `logstash` service on port `5044`.
  - **Volume Mounts for Logs:** This is the *most critical part* for Filebeat. Filebeat runs inside a container, but it needs to access logs on your *Windows host machine*. This is achieved through Docker volume mounts in `docker-compose.yml`. For `winlog` input, Filebeat uses Windows APIs that Docker Desktop handles well. For `filestream` (file-based logs), you need to explicitly mount the host directory into the container. For example, if your app logs are in `C:\my_app_logs` on your Windows host, you might mount it as `/host_logs` in the container and configure Filebeat to read from `/host_logs/my_app.log`.

2.5. `grafana/provisioning/datasources/loki.yml`

This file automatically configures Loki as a data source in Grafana.

```yaml
apiVersion: 1

datasources:
  - name: Loki
    type: loki
    access: proxy
    orgId: 1
    url: http://loki:3100 # Use the service name 'loki' for inter-container communication
    version: 1
    editable: true
    isDefault: true
```

2.6. grafana/provisioning/dashboards/default.yml (Optional but Recommended)

This file enables dashboard provisioning, allowing you to easily import dashboards into Grafana. You can later add actual dashboard JSON files.

```yaml
apiVersion: 1

providers:
  - name: 'default'
    orgId: 1
    folder: ''
    type: file
```

```
    disableDeletion: false
    editable: true
    options:
      path: /etc/grafana/provisioning/dashboards
```

---

3. Create `docker-compose.yml`

This file defines all your services (Loki, Logstash, Filebeat, Grafana), their Docker images, ports, volumes, and network settings.

Create `docker-compose.yml` in your `C:\fllg-stack\` directory.

```
YAML

version: '3.8'

networks:
  logging_network:
    driver: bridge

volumes:
  grafana_data:    # Persistent volume for Grafana data (dashboards, users)
  loki_data:       # Persistent volume for Loki data (chunks, index)

services:
  loki:
    image: grafana/loki:latest
    container_name: loki
    ports:
```

```
    - "3100:3100" # Expose Loki HTTP port to host
    volumes:
      - ./loki/loki-local-config.yaml:/etc/loki/local-config.yaml:ro # Mount Loki config
      - loki_data:/loki # Mount persistent volume for Loki data
    command: -config.file=/etc/loki/local-config.yaml # Tell Loki to use the mounted config
    networks:
      - logging_network
    restart: unless-stopped


  logstash:
    image: docker.elastic.co/logstash/logstash:8.14.0 # Use a compatible Logstash version
    container_name: logstash
    ports:
      - "5044:5044" # Filebeat will send to this port
    volumes:
      - ./logstash/pipeline:/usr/share/logstash/pipeline:ro # Mount pipeline config
      - ./logstash/config/logstash.yml:/usr/share/logstash/config/logstash.yml:ro # Mount logstash.yml
    environment:
      LS_JAVA_OPTS: "-Xmx512m -Xms512m" # Adjust Java heap size as needed
    networks:
      - logging_network
    depends_on:
      - loki # Logstash depends on Loki being up
    restart: unless-stopped
    # Add an init_container if plugin installation is needed:
```

```
    # entrypoint: >
    #   bash -c '
    #     if [ ! -f /usr/share/logstash/plugins_installed ]; then
    #       logstash-plugin install logstash-output-loki && \
    #       touch /usr/share/logstash/plugins_installed;
    #     fi && \
    #     /usr/local/bin/docker-entrypoint.sh logstash -f /usr/share/logstash/pipeline/logstash.conf
    #   '
    # Note: Modern Logstash images might include common plugins or handle this better.
    # The `grafana/logstash-output-loki` image already includes the plugin.
    # If using `docker.elastic.co/logstash/logstash`, you might need this entrypoint logic.
    # For simplicity, we'll assume the plugin is pre-installed or rely on `grafana/logstash-output-loki`
below.


  filebeat:
    image: docker.elastic.co/beats/filebeat:8.14.0 # Use a compatible Filebeat version
    container_name: filebeat
    user: root # Needed for accessing system logs and mounting volumes
    command: filebeat -e -c /etc/filebeat/filebeat.yml
    volumes:
      - ./filebeat/filebeat.yml:/etc/filebeat/filebeat.yml:ro # Mount Filebeat config
      # For Windows Event Logs, Docker Desktop usually handles it through its integration.
      # For file-based logs, you MUST mount the host directory:
      - type: bind # This is important for host paths
        source: C:/path/to/your/app/logs # Replace with your actual host log path
        target: /host_app_logs # Path inside the container for these logs
```

```
      read_only: true

    # Example for Windows event logs which usually don't require explicit volume mounts beyond Docker Desktop itself:

    # - \\.\pipe\docker_engine:\\.\pipe\docker_engine:ro # If you run into issues with winlog, sometimes this helps, but typically not needed for Docker Desktop.

  networks:

    - logging_network

  depends_on:

    - logstash # Filebeat depends on Logstash being up

  restart: unless-stopped

  # Required for Windows Event Log collection with Filebeat in Docker:

  cap_add:

    - SYS_ADMIN # Needed for some system-level operations on the host

  # For Windows Event Logs, Filebeat running in Docker Desktop (Linux VM) uses special mechanisms.

  # It communicates with a helper service on the Windows host.

  # Ensure your `filebeat.yml` for winlog type specifies the event_logs correctly.


grafana:

  image: grafana/grafana:latest

  container_name: grafana

  ports:

    - "3000:3000" # Expose Grafana UI port to host

  volumes:

    - grafana_data:/var/lib/grafana # Persistent volume for Grafana data

    - ./grafana/provisioning:/etc/grafana/provisioning:ro # Mount provisioning config
```

```
  environment:
    GF_SECURITY_ADMIN_USER: admin
    GF_SECURITY_ADMIN_PASSWORD: your_strong_password # CHANGE THIS PASSWORD!
  networks:
    - logging_network
  depends_on:
    - loki # Grafana depends on Loki being up
  restart: unless-stopped
```

- **Key points in `docker-compose.yml`:**
  - `networks`: Defines a custom bridge network for all services to communicate with each other using their service names (e.g., `loki`, `logstash`, `grafana`).
  - `volumes`:
    - `grafana_data` and `loki_data`: These are Docker *named volumes*. Docker manages where these are stored on your host (usually in `C:\ProgramData\Docker\volumes`). They ensure that your Grafana dashboards, users, and Loki's stored logs persist even if you stop or remove the containers.
    - `./loki/loki-local-config.yaml:/etc/loki/local-config.yaml:ro`: This is a *bind mount*. It maps your host's `C:\fllg-stack\loki\loki-local-config.yaml` file directly to `/etc/loki/local-config.yaml` inside the Loki container. `:ro` means read-only.
    - **Filebeat Host Log Access:** The crucial part for Filebeat to access your Windows host logs is the `type: bind` volume mount:
    - YAML

```
- type: bind
  source: C:/path/to/your/app/logs # Your actual Windows host path
  target: /host_app_logs           # Path inside the Filebeat container
  read_only: true
```

        - ■
            - **For Windows Event Logs (`winlog` input):** Docker Desktop handles the `winlog` input from Filebeat running in a Linux container by using a special Windows agent. You typically don't need explicit volume mounts for event logs, but ensuring the `user: root` and `cap_add: SYS_ADMIN` are present is good practice. If you face issues, some guides mention mounting the Docker engine pipe (`//./pipe/docker_engine`).
    - `ports`: Maps container ports to host ports, allowing you to access Grafana (3000) and Logstash (5044) from your Windows browser/Filebeat.
    - `depends_on`: Ensures services start in a logical order (e.g., Logstash starts after Loki).
    - `restart: unless-stopped`: Configures containers to restart automatically unless explicitly stopped.
    - **Logstash Plugin Installation:** If you're using the generic `docker.elastic.co/logstash/logstash` image, you might need an `entrypoint` or `command` modification to install the `logstash-output-loki` plugin the first time the container starts. The commented-out section in `logstash` service shows an example. If you use a pre-built image like `grafana/logstash-output-loki` (if one exists and is maintained), that step might be skipped. For this guide, we're sticking with the Elastic image, so be aware you might need to uncomment and adapt the `entrypoint` if the plugin isn't there by default.

---

4. Deployment and Verification

4.1. Open PowerShell / Command Prompt as Administrator

4.2. Navigate to your project directory:

```
Bash

cd C:\fllg-stack

```

4.3. Start the stack:

```
Bash

docker compose up -d

```

- The `-d` flag runs the containers in detached mode (in the background).

- Docker Compose will download the necessary images, create the network and volumes, and start the services. This might take some time on the first run.

4.4. Verify container status:

```
Bash

docker ps

```

You should see `loki`, `logstash`, `filebeat`, and `grafana` containers listed with "Up" status.

4.5. Check logs (for debugging):

```
Bash

docker compose logs -f loki

docker compose logs -f logstash

docker compose logs -f filebeat

docker compose logs -f grafana

```

The `-f` flag tails the logs in real-time. This is very useful for troubleshooting.

4.6. Access Grafana:

1. Open your web browser and go to `http://localhost:3000`.
2. Log in with `admin` and the password you set in `docker-compose.yml` (`your_strong_password`). You'll be prompted to change it.
3. Since you used provisioning, the "Loki" data source should already be configured.
4. Navigate to the "Explore" section in Grafana (compass icon on the left sidebar).
5. Select "Loki" as your data source.
6. Start querying your logs using LogQL. For example, to see all logs sent by Filebeat:
7. Code snippet

```
{job="filebeat_logs"}
```

8.
9. You should start seeing log lines appear if Filebeat is correctly collecting and sending them.

---

5. Managing the Stack

- **Stop the stack:**
- Bash

docker compose down

-
- This stops and removes the containers and the default network. It *does not* remove named volumes (like `grafana_data`, `loki_data`), so your data persists.
- **Stop and remove everything (including volumes - use with caution!):**
- Bash

docker compose down --volumes

-
- This will delete your Grafana data and Loki log data.
- **Restart a specific service:**
- Bash

docker compose restart filebeat

-
-
- **Rebuild images (if you change Dockerfiles - not used in this example but good to know):**
- Bash

docker compose build

-
-

Directory Structure Summary (Dockerized):

C:\fllg-stack\

├── docker-compose.yml # Defines all services and their relationships

```
├── grafana\
│   ├── provisioning\
│   │   ├── datasources\
│   │   │   └── loki.yml        # Grafana data source auto-provisioning
│   │   └── dashboards\
│   │       └── default.yml     # Grafana dashboard auto-provisioning
├── loki\
│   └── loki-local-config.yaml  # Loki server configuration
├── logstash\
│   ├── pipeline\
│   │   └── logstash.conf       # Logstash processing pipeline
│   └── config\
│       └── logstash.yml        # Logstash general configuration
├── filebeat\
│   └── filebeat.yml            # Filebeat configuration for log collection
```

Advantages of the Dockerized Setup:

- **Isolation:** Each component runs in its own isolated environment, preventing conflicts with other software on your system.
- **Reproducibility:** The `docker-compose.yml` file serves as a blueprint, allowing you to recreate the exact same environment on any machine with Docker installed.
- **Portability:** You can easily move this entire stack to another Windows machine, Linux server, or even a cloud VM by simply copying the `fllg-stack` directory and running `docker compose up -d`.
- **Simplified Installation:** No more manual JDK, Logstash plugin, or service installations. Docker handles dependencies and runtimes.
- **Version Management:** Easily change component versions by updating the `image` tag in `docker-compose.yml`.
- **Cleanup:** Easily stop and remove the entire stack with a single `docker compose down` command.

This Dockerized setup is generally more robust and easier to manage for both development and production environments compared to manual installations on Windows.

ALTERNATIVE WAY:

The previous setups (Kafka/Grafana or FLLG stack) are standard for real-time log monitoring. If they don't work, it's often due to specific configuration issues, port conflicts, or permission problems. However, if you're looking for an *easier* alternative, especially for getting started quickly, the **Promtail + Loki + Grafana (PLG) stack** is arguably the simplest for log aggregation and visualization.

Why PLG is "Easier":

- **No Logstash:** Eliminates the need for Logstash, which can be complex to configure with its filtering language and Java dependency. Promtail is a simpler, lightweight agent specifically designed to send logs to Loki.[1]
- **No Kafka:** No need to manage Kafka or ZooKeeper, simplifying the architecture significantly.
- **Built for Logs:** Loki is designed specifically for logs, making it efficient and easy to query from Grafana.[2]
- **Simple Agent:** Promtail is a Go binary with a straightforward YAML configuration, similar to Filebeat but specifically for Loki.[3]

Easier Way (Without Docker): Promtail + Loki + Grafana (PLG)

Overall Architecture:

1. **Log Source:** Your Windows system generates logs.
2. **Promtail:** A lightweight agent that collects logs from files (and with recent versions, even Windows Event Logs) and pushes them directly to Loki.[4]
3. **Loki:** Stores the logs by indexing only their metadata (labels).[5]
4. **Grafana:** Visualizes the logs from Loki.[6]

Prerequisites:

- **Windows Administrator Privileges:** For installing Promtail and Loki as services (optional, but recommended for continuous operation).

- **No Java required for Loki/Promtail.**

1. Install and Configure Grafana

(Same as previous guides)

1. **Download Grafana MSI:** https://grafana.com/grafana/download
2. **Install Grafana:** Run the MSI, follow prompts. Default path: `C:\Program Files\GrafanaLabs\grafana`.
3. **Access Grafana:** `http://localhost:3000` (admin/admin).

2. Install and Configure Loki

(Same as previous guide for standalone Loki)

1. **Download Loki:**
    - Go to the Grafana Loki releases page: https://grafana.com/oss/loki/
    - Find `loki-windows-amd64.zip` and download.
2. **Extract Loki:**
    - Create `C:\Loki`.
    - Extract `loki-windows-amd64.exe` into it.
3. **Create `loki-local-config.yaml` in `C:\Loki`:**
4. YAML

```yaml
auth_enabled: false

server:
  http_listen_port: 3100
  grpc_listen_port: 9095

common:
  path_prefix: C:\Loki\data
  replication_factor: 1
  ring:
    instance_addr: 127.0.0.1
    kvstore:
```

```
      store: inmemory
  storage:
    filesystem:
      directory: C:\Loki\data\chunks
    boltdb_shipper:
      active_index_directory: C:\Loki\data\boltdb-shipper-active
      cache_location: C:\Loki\data\boltdb-shipper-cache
      resync_interval: 5s
    tsdb:
      dir: C:\Loki\data\tsdb

query_range:
  align_queries_with_step: true
  cache_results: true

schema_config:
  configs:
    - from: 2020-10-24
      store: boltdb-shipper
      object_store: filesystem
      schema: v11
      period: 24h
      index:
        prefix: index_
        period: 24h
```

```
memberlist:
  join_members: []

compactor:
  working_directory: C:\Loki\data\compactor
  compaction_interval: 10m
```

5.
    - **Crucial:** Create the directories: C:\Loki\data, C:\Loki\data\chunks, C:\Loki\data\boltdb-shipper-active, C:\Loki\data\boltdb-shipper-cache, C:\Loki\data\tsdb, C:\Loki\data\compactor.
6. **Start Loki:**
    - Open **Command Prompt as Administrator**.
    - Bash

```
cd C:\Loki
.\loki-windows-amd64.exe --config.file=loki-local-config.yaml
```

    -
    -
    - Keep this window open.

3. Install and Configure Promtail

Promtail is Loki's official log agent.[7]

1. **Download Promtail:**
    - Go to the Grafana Loki releases page: https://grafana.com/oss/loki/
    - Find promtail-windows-amd64.zip and download.
2. **Extract Promtail:**
    - Create a dedicated directory, e.g., C:\Promtail.
    - Extract promtail-windows-amd64.exe and promtail-config.yaml (from the archive) into C:\Promtail.
3. **Configure promtail-config.yaml:**
    - Open C:\Promtail\promtail-config.yaml in a text editor.

- Modify the `server`, `clients`, and `scrape_configs` sections.
- **For File-based Logs:**
- YAML

```yaml
server:
  http_listen_port: 9080
  grpc_listen_port: 0

positions:
  filename: C:\Promtail\positions.yaml # Promtail uses this to track what it's read

clients:
  - url: http://localhost:3100/loki/api/v1/push # Loki's ingest URL

scrape_configs:
  - job_name: system_logs
    static_configs:
      - targets:
          - localhost
        labels:
          job: windows_app_logs # A static label for this log source
          __path__: C:\your\application\logs\*.log # Path to your application logs
          host: your_windows_hostname # Replace with actual hostname
  # Another scrape config for another type of logs
  # - job_name: custom_logs
  #   static_configs:
```

```
  #     - targets:
  #         - localhost
  #       labels:
  #         job: custom_app_logs
  #         __path__: C:\another\log\directory\*.txt
  #         app: my_custom_app
  # log_format: json # If your logs are JSON, uncomment this
  # pipeline_stages: # Optional: for parsing log lines
  #   - json:
  #       expressions:
  #         level: level
  #         message: message
  #   - labels:
  #       level:
```

-
-
- **For Windows Event Logs (more recent Promtail versions support this):**
- YAML

```
server:
  http_listen_port: 9080
  grpc_listen_port: 0

positions:
  filename: C:\Promtail\positions.yaml
```

```
clients:
  - url: http://localhost:3100/loki/api/v1/push


scrape_configs:
  - job_name: windows_event_logs
    windows_events:
      channel: [ "Application", "System", "Security" ] # Event log channels to collect
      # If you need to filter specific events:
      # query: "*[System[(Level=2 or Level=3)]]" # Example: Error (2) or Warning (3) level
    relabel_configs: # Relabeling allows you to create labels from event fields
      - source_labels: ['__winlog_channel']
        target_label: 'channel'
      - source_labels: ['__winlog_level']
        target_label: 'level'
      - source_labels: ['__winlog_provider_name']
        target_label: 'provider'
    labels:
      job: windows_events # Base job label
      host: your_windows_hostname # Replace with actual hostname
```

-
-
- **Key Promtail Config Points:**
  - `positions.filename`: Promtail uses this file to keep track of what log lines it has already sent, preventing duplicates on restart.
  - `clients.url`: Points to your Loki instance.
  - `scrape_configs`: Define what logs to collect (`__path__` for files, `windows_events` for event logs) and what **labels** to attach to them. Labels are crucial for querying in Grafana.

        - `labels`: These are static or dynamically extracted metadata for your log streams. Choose low-cardinality values.
        - `pipeline_stages` (Optional): Allows you to parse log lines (e.g., JSON, Regex) and extract fields which can then become labels or just be included in the log message.[8] This replaces Logstash's filtering role for simple cases.
4. **Start Promtail:**
    - Open **another Command Prompt as Administrator**.
    - Bash

```
cd C:\Promtail

.\promtail-windows-amd64.exe -config.file=promtail-config.yaml
```

    -
    -
    - Keep this window open.

4. Integrate Loki with Grafana

(Same as previous guides)

1. **Add Loki Data Source in Grafana:**
    - Log in to Grafana (`http://localhost:3000`).
    - Go to "Connections" -> "Data sources" -> "Add data source" -> "Loki".
    - **URL:** `http://localhost:3100`
    - Click "Save & test".
2. **Create a Grafana Dashboard for Logs:**
    - Go to "Dashboards" -> "New dashboard" -> "Add a new panel".
    - Select `Loki` as data source.
    - **LogQL Query:** Use the labels you defined in Promtail.
        - File logs example: `{job="windows_app_logs", host="your_windows_hostname"}`
        - Event logs example: `{job="windows_events", channel="Application"}`
    - **Visualization:** Choose "Logs". Adjust time range and refresh interval.

Easier Way (With Docker): Promtail + Loki + Grafana (PLG)

This is arguably the cleanest and easiest way to run the stack, as Docker Compose manages all the interconnections and dependencies.

Overall Architecture (with Docker):

1. **Log Source:** Your Windows host system.
2. **Promtail (Container):** Collects logs from your Windows host (via volume mounts for files, or Docker Desktop's integration for event logs) and pushes directly to the Loki container.[9]
3. **Loki (Container):** Stores logs.
4. **Grafana (Container):** Visualizes logs.

Prerequisites:

- **Docker Desktop for Windows:** (See detailed installation steps from previous answer).
- **Windows Administrator Privileges:** For running Docker Desktop.

1. Project Setup and Directory Structure

```
C:\plg-stack\
├── docker-compose.yml
├── grafana\
│   └── provisioning\
│       ├── datasources\
│       │   └── loki.yml
│       └── dashboards\
│           └── default.yml
├── loki\
│   └── loki-local-config.yaml
├── promtail\
│   └── promtail-config.yaml
```

2. Create Configuration Files

(Mostly similar to the non-Docker versions, but adjust paths to container-internal paths and use service names for URLs).

2.1. loki/loki-local-config.yaml

(Identical to the Dockerized FLLG Loki config, as it's container-internal paths)

YAML

```yaml
auth_enabled: false

server:
  http_listen_port: 3100
  grpc_listen_port: 9095

common:
  path_prefix: /loki
  replication_factor: 1
  ring:
    instance_addr: 127.0.0.1
    kvstore:
      store: inmemory

  storage:
    filesystem:
      directory: /loki/chunks
    boltdb_shipper:
      active_index_directory: /loki/boltdb-shipper-active
      cache_location: /loki/boltdb-shipper-cache
      resync_interval: 5s
    tsdb:
      dir: /loki/tsdb
```

```
query_range:
  align_queries_with_step: true
  cache_results: true
schema_config:
  configs:
    - from: 2020-10-24
      store: boltdb-shipper
      object_store: filesystem
      schema: v11
      period: 24h
      index:
        prefix: index_
        period: 24h
memberlist:
  join_members: []
compactor:
  working_directory: /loki/compactor
  compaction_interval: 10m
```

2.2. promtail/promtail-config.yaml

- positions.filename: This will be inside the container, but we'll volume mount it for persistence.
- clients.url: Use the service name loki.
- __path__: These are paths *inside the container* that you will map from your host.

```
YAML

server:
```

```
  http_listen_port: 9080
  grpc_listen_port: 0

positions:
  filename: /tmp/positions.yaml # Path inside the container, mapped to a volume

clients:
  - url: http://loki:3100/loki/api/v1/push # Use the service name 'loki'

scrape_configs:
  # For file-based logs on your Windows host
  - job_name: windows_host_app_logs
    static_configs:
      - targets:
          - localhost
        labels:
          job: windows_app_logs
          __path__: /host_logs/my_app/*.log # This path MUST match your Docker volume mount!
          host: your_windows_hostname # Replace with your actual Windows hostname
    # Optional: If your app logs are JSON
    # pipeline_stages:
    #   - json:
    #       expressions:
    #         level: level
    #         message: message
```

```
  #   - labels:
  #       level:


# For Windows Event Logs
- job_name: windows_event_logs
  windows_events:
    channel: [ "Application", "System", "Security" ]
    # query: "*[System[(Level=2 or Level=3)]]" # Example filter
  relabel_configs:
    - source_labels: ['__winlog_channel']
      target_label: 'channel'
    - source_labels: ['__winlog_level']
      target_label: 'level'
    - source_labels: ['__winlog_provider_name']
      target_label: 'provider'
  labels:
    job: windows_events
    host: your_windows_hostname
```

2.3. grafana/provisioning/datasources/loki.yml

(Identical to the Dockerized FLLG Grafana datasource config)

```
YAML

apiVersion: 1


datasources:
```

```
  - name: Loki
    type: loki
    access: proxy
    orgId: 1
    url: http://loki:3100
    version: 1
    editable: true
    isDefault: true
```

2.4. grafana/provisioning/dashboards/default.yml

(Identical to the Dockerized FLLG Grafana dashboard provisioning)

```
YAML
apiVersion: 1

providers:
  - name: 'default'
    orgId: 1
    folder: ''
    type: file
    disableDeletion: false
    editable: true
    options:
      path: /etc/grafana/provisioning/dashboards
```

3. Create docker-compose.yml

This will be simpler than the FLLG one because there's no Logstash.

Create `docker-compose.yml` in your `C:\plg-stack\` directory.

YAML

```yaml
version: '3.8'

networks:
  logging_network:
    driver: bridge

volumes:
  grafana_data:
  loki_data:
  promtail_positions: # Volume for Promtail's positions.yaml

services:
  loki:
    image: grafana/loki:latest
    container_name: loki
    ports:
      - "3100:3100"
    volumes:
      - ./loki/loki-local-config.yaml:/etc/loki/local-config.yaml:ro
      - loki_data:/loki
    command: -config.file=/etc/loki/local-config.yaml
    networks:
```

```
    - logging_network
  restart: unless-stopped

promtail:
  image: grafana/promtail:latest
  container_name: promtail
  user: root # Needed for accessing system logs and host volumes
  command: -config.file=/etc/promtail/promtail-config.yaml
  volumes:
    - ./promtail/promtail-config.yaml:/etc/promtail/promtail-config.yaml:ro # Mount Promtail config
    - promtail_positions:/tmp # Mount volume for positions.yaml
    # Crucial: Mount host log directories into the container
    # For your application logs on Windows host:
    - type: bind
      source: C:/path/to/your/application/logs # Replace with your actual Windows host log path
      target: /host_logs/my_app # Path inside the Promtail container
      read_only: true
    # For Windows Event Logs, Promtail uses a special Windows API via Docker Desktop:
    # No explicit volume mount of C:\Windows\System32\winevt\Logs or similar is needed for winlog.
    # Docker Desktop handles this abstraction.
  networks:
    - logging_network
  depends_on:
    - loki # Promtail depends on Loki being up
  restart: unless-stopped
```

```
    cap_add: # Required for Windows Event Log collection
      - SYS_ADMIN
      - DAC_READ_SEARCH # Might be needed for some log access

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    ports:
      - "3000:3000"
    volumes:
      - grafana_data:/var/lib/grafana
      - ./grafana/provisioning:/etc/grafana/provisioning:ro
    environment:
      GF_SECURITY_ADMIN_USER: admin
      GF_SECURITY_ADMIN_PASSWORD: your_strong_password # CHANGE THIS PASSWORD!
    networks:
      - logging_network
    depends_on:
      - loki
    restart: unless-stopped
```

4. Deployment and Verification (Docker)

1. Open **PowerShell / Command Prompt as Administrator**.
2. Navigate to your project directory: `cd C:\plg-stack`.
3. **Start the stack:**
4. Bash

```
docker compose up -d
```

5.
6.
7. **Verify container status:**
8. Bash

```
docker ps
```

9.
10.
11. **Check logs (for debugging):**
12. Bash

```
docker compose logs -f loki

docker compose logs -f promtail

docker compose logs -f grafana
```

13.
14.
15. **Access Grafana:**
    - Go to http://localhost:3000.
    - Log in (admin/your_strong_password).
    - Go to "Explore", select "Loki" data source.
    - Query your logs: e.g., {job="windows_app_logs"} or {job="windows_events"}.

5. Managing the Stack (Docker)

- **Stop:** docker compose down
- **Stop and remove volumes:** docker compose down --volumes
- **Restart a service:** docker compose restart promtail

Why this is "Easier" with Docker:

- **Less Configuration Files:** No Logstash pipeline, simplifying transformation logic directly into Promtail's pipeline_stages if needed.
- **Fewer Moving Parts:** Direct push from Promtail to Loki, fewer services to manage.
- **Simplified Troubleshooting:** Fewer components mean fewer potential points of failure.

- **Official Tools:** Promtail is purpose-built by Grafana Labs to work seamlessly with Loki.[10]

This PLG stack, especially with Docker Compose, is an excellent choice for straightforward log monitoring and is often recommended for its simplicity and efficiency when you don't need the heavy-duty processing power of Logstash or the messaging queue capabilities of Kafka for your log pipeline.

POSTGRES

This is a comprehensive request! Setting up PostgreSQL, Kafka, and Grafana on Windows *without Docker* for data visualization involves several distinct installation and configuration steps. I'll guide you through each part.

Understanding the Flow:

1. **PostgreSQL:** Your primary data source. Data will reside here.
2. **Kafka:** An event streaming platform. We'll use Kafka Connect to move data:
    - **Source Connector:** To capture changes from PostgreSQL and push them to Kafka topics (for real-time).
    - **Sink Connector:** (Optional, but good for demonstrating the full cycle) To consume data from Kafka topics and write it back to another PostgreSQL table, or even the same one.
3. **Grafana:** A powerful visualization tool that will connect directly to PostgreSQL to query and display your data (both static and real-time data that has landed in PostgreSQL via Kafka).

Prerequisites:

- **Java Development Kit (JDK):** Kafka requires Java. Download and install a recent JDK (e.g., OpenJDK 11 or higher). Set your `JAVA_HOME` environment variable.
- **Administrator Privileges:** You'll need admin rights for some installations and firewall configurations.
- **Good Internet Connection:** For downloading software.

Part 1: PostgreSQL Installation and Setup

1. **Download PostgreSQL:**
    - Go to the official PostgreSQL download page: https://www.postgresql.org/download/windows/
    - Download the interactive installer for Windows. Choose the latest stable version.
2. **Install PostgreSQL:**
    - Run the downloaded installer.
    - Follow the on-screen instructions. Key points:

        - **Installation Directory:** Choose a suitable location (e.g., `C:\Program Files\PostgreSQL\<version>`).
        - **Components:** Ensure "PostgreSQL Server," "pgAdmin 4," and "Command Line Tools" are selected. You might not strictly need "Stack Builder" for this setup.
        - **Data Directory:** Choose where your database files will be stored (e.g., `C:\Program Files\PostgreSQL\<version>\data`).
        - **Password:** Set a strong password for the `postgres` superuser. **Remember this password!**
        - **Port:** The default is `5432`. Keep it if you don't have conflicts.
        - **Locale:** Choose your preferred locale.
    - The installer will complete the setup and start the PostgreSQL service.
3. **Verify PostgreSQL Installation:**
    - Open the "SQL Shell (psql)" from the Start Menu.
    - Press Enter for Server (`localhost`), Database (`postgres`), Port (`5432`), Username (`postgres`).
    - Enter the password you set during installation.
    - You should see the `psql` prompt (`postgres=#`).
    - Type `SELECT version();` and press Enter. You should see the PostgreSQL version information.
    - Type `\q` to exit psql.
4. **Create a Database and User for Grafana/Kafka:**
    - Open "SQL Shell (psql)" again.
    - Connect as `postgres` user.
    - Create a new database (e.g., `mydatabase`):
    - SQL

```
CREATE DATABASE mydatabase;
```

    -
    -
    - Create a new user (e.g., `myuser`) with a password (e.g., `mypassword`):
    - SQL

```
CREATE USER myuser WITH PASSWORD 'mypassword';
```

    -
    -
    - Grant privileges to the new user on the database:
    - SQL

```
GRANT ALL PRIVILEGES ON DATABASE mydatabase TO myuser;
```

    -
    -
    - Connect to your new database:
    - SQL

```
\c mydatabase;
```

    -
    -
    - Grant usage on the `public` schema to your user (important for table visibility):
    - SQL

```
GRANT USAGE ON SCHEMA public TO myuser;
```

    -
    -
    - For any tables you create later, you'll need to grant `SELECT` privileges to `myuser` if you want Grafana to read them. For simplicity, you can grant `SELECT` on all tables in `public` for now:
    - SQL

```
ALTER DEFAULT PRIVILEGES FOR ROLE myuser IN SCHEMA public GRANT SELECT ON TABLES TO myuser;
```

    -
    -
    - Type `\q` to exit psql.

Part 2: Apache Kafka Installation and Setup

Kafka requires ZooKeeper (or Kraft mode for newer versions, but for simplicity on Windows without Docker, ZooKeeper is often assumed).

1. **Download Kafka:**
    - Go to the Apache Kafka downloads page: `https://kafka.apache.org/downloads`

    - Download the latest binary release (choose the Scala 2.13 version, e.g., `kafka_2.13-3.x.x.tgz`).
2. **Extract Kafka:**
    - Use a tool like 7-Zip or WinRAR to extract the `.tgz` file. You might need to extract it twice: first to a `.tar` and then to a folder.
    - Extract the contents to a simple path, e.g., `C:\kafka_2.13-3.x.x`. Let's refer to this as `%KAFKA_HOME%`.
3. **Configure Kafka:**
    - Navigate to `%KAFKA_HOME%\config`.
    - **Edit `zookeeper.properties`:**
        - Open `zookeeper.properties` in a text editor.
        - Change `dataDir=/tmp/zookeeper` to a Windows-friendly path, e.g., `dataDir=C:/kafka_logs/zookeeper`.
    - **Edit `server.properties`:**
        - Open `server.properties` in a text editor.
        - Change `log.dirs=/tmp/kafka-logs` to a Windows-friendly path, e.g., `log.dirs=C:/kafka_logs/kafka-data`.
        - Ensure `listeners=PLAINTEXT://localhost:9092` (default, usually fine).
4. **Start ZooKeeper:**
    - Open a **new Command Prompt as Administrator**.
    - Navigate to your Kafka installation's `bin\windows` directory:
    - Bash

```
cd C:\kafka_2.13-3.x.x\bin\windows
```

    -
    -
    - Start ZooKeeper:
    - Bash

```
zookeeper-server-start.bat ..\..\config\zookeeper.properties
```

    -
    -
    - Keep this window open. ZooKeeper needs to be running for Kafka to function.
5. **Start Kafka Broker:**
    - Open **another new Command Prompt as Administrator**.
    - Navigate to your Kafka installation's `bin\windows` directory:
    - Bash

```
cd C:\kafka_2.13-3.x.x\bin\windows
```

-
-
- Start Kafka:
- Bash

```
kafka-server-start.bat ..\..\config\server.properties
```

-
-
- Keep this window open.

6. **Create a Kafka Topic (Optional, but good for testing):**
   - Open **another new Command Prompt as Administrator**.
   - Navigate to your Kafka installation's `bin\windows` directory:
   - Bash

```
cd C:\kafka_2.13-3.x.x\bin\windows
```

-
-
- Create a topic (e.g., `my_topic`):
- Bash

```
kafka-topics.bat --create --topic my_topic --bootstrap-server localhost:9092 --partitions 1
--replication-factor 1
```

-
-
- Verify the topic:
- Bash

```
kafka-topics.bat --list --bootstrap-server localhost:9092
```

-
-

Part 3: Kafka Connect for PostgreSQL Integration

Kafka Connect is a framework for connecting Kafka with other systems. We'll use JDBC connectors.

1. **Download Kafka Connect JDBC Plugin:**
    - You'll need the Confluent Community JDBC Connector.
    - Go to Confluent Hub: `https://www.confluent.io/hub/confluentinc/kafka-connect-jdbc`
    - Download the latest version as a ZIP file.
2. **Install JDBC Plugin:**
    - Create a directory for Kafka Connect plugins, e.g., `C:\kafka_plugins`.
    - Extract the contents of the downloaded JDBC connector ZIP file into `C:\kafka_plugins`. You should have a structure like `C:\kafka_plugins\confluentinc-kafka-connect-jdbc-<version>\lib`.
3. **Download PostgreSQL JDBC Driver:**
    - You need the PostgreSQL JDBC driver for Kafka Connect to interact with PostgreSQL.
    - Go to the PostgreSQL JDBC driver download page: `https://jdbc.postgresql.org/download/`
    - Download the latest stable JAR file (e.g., `postgresql-42.x.x.jar`).
4. **Place JDBC Driver in Plugin Path:**
    - Copy the `postgresql-42.x.x.jar` file into the `lib` directory of your JDBC connector plugin: `C:\kafka_plugins\confluentinc-kafka-connect-jdbc-<version>\lib`.
5. **Configure Kafka Connect Standalone:**
    - Navigate to `%KAFKA_HOME%\config`.
    - **Edit `connect-standalone.properties`:**
        - Open `connect-standalone.properties`.
        - Find and uncomment (or add if missing) `plugin.path=`. Set it to your plugin directory:
        - Properties

```
plugin.path=C:/kafka_plugins
```

        -
        - (Use forward slashes for paths in properties files on Windows for consistency).
6. **Create Kafka Connect Source Connector Configuration (`pg-source.properties`):**
    - Create a new file named `pg-source.properties` in `%KAFKA_HOME%\config` (or a dedicated `connectors` folder).
    - Paste the following configuration, adjusting placeholders:
    - Properties

```
name=postgresql-source-connector

connector.class=io.confluent.connect.jdbc.JdbcSourceConnector

tasks.max=1

connection.url=jdbc:postgresql://localhost:5432/mydatabase

connection.user=myuser

connection.password=mypassword

mode=timestamp+incrementing

topic.prefix=postgres_

table.whitelist=your_table_name_here # Replace with the actual table you want to monitor

timestamp.column.name=updated_at # Assuming you have an 'updated_at' column for changes

incrementing.column.name=id # Assuming you have an 'id' column that is incrementing

poll.interval.ms=5000 # Poll every 5 seconds for new data
```

- table.whitelist: **Crucial!** This should be the table you want to stream from PostgreSQL.
- timestamp.column.name: Recommended for real-time. This column in your source table should be a TIMESTAMP or TIMESTAMPTZ and update on every change.
- incrementing.column.name: An auto-incrementing INTEGER or BIGINT column in your source table.
- If you don't have timestamp.column.name and only want to capture new rows, you can use mode=incrementing and just incrementing.column.name.
- For a simple "static" load (one-time dump), you can use mode=bulk.

7. **Create a Sample Table in PostgreSQL:**
   - Open "SQL Shell (psql)", connect to mydatabase as myuser.
   - Create a table. Make sure to include id (primary key) and updated_at (timestamp for changes) columns:
   - SQL

```
CREATE TABLE sensor_data (

    id SERIAL PRIMARY KEY,
```

```
    temperature DECIMAL(5, 2),
    humidity DECIMAL(5, 2),
    location VARCHAR(100),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Grant select on this table to your user, if not already granted via default privileges
GRANT SELECT ON sensor_data TO myuser;
```

-
-
- Insert some initial data:
- SQL

```
INSERT INTO sensor_data (temperature, humidity, location) VALUES (25.5, 60.2, 'Living Room');
INSERT INTO sensor_data (temperature, humidity, location) VALUES (22.1, 55.0, 'Bedroom');
```

-
-
- Update `pg-source.properties` to `table.whitelist=sensor_data`.

8. **Start Kafka Connect (Source):**
   - Open **another new Command Prompt as Administrator**.
   - Navigate to `%KAFKA_HOME%\bin\windows`.
   - Run Kafka Connect in standalone mode with your source connector configuration:
   - Bash

```
connect-standalone.bat ..\..\config\connect-standalone.properties ..\..\config\pg-source.properties
```

-
-
- You should see logs indicating the connector starting and pulling data.
- **Verify Kafka Topic Data:** Open another CMD, navigate to `%KAFKA_HOME%\bin\windows`, and run:
- Bash

```
kafka-console-consumer.bat --topic postgres_sensor_data --bootstrap-server localhost:9092
--from-beginning
```

    -
    - (The topic name will be `topic.prefix` + `table.name`, so `postgres_sensor_data`). You should see the data from your PostgreSQL table appearing here.

Part 4: Grafana Installation and Setup

1. **Download Grafana:**
    - Go to the Grafana download page: `https://grafana.com/grafana/download?platform=windows`
    - Download the installer (usually `.msi` file).
2. **Install Grafana:**
    - Run the downloaded `.msi` installer.
    - Follow the on-screen instructions. The default installation path (e.g., `C:\Program Files\GrafanaLabs\grafana`) is usually fine.
    - Grafana will typically install as a Windows service and start automatically.
3. **Access Grafana:**
    - Open your web browser and navigate to `http://localhost:3000`.
    - The default username and password are `admin`/`admin`. You will be prompted to change the password on your first login.
4. **Add PostgreSQL Data Source in Grafana:**
    - After logging in, click the **gear icon (Configuration)** on the left-hand menu.
    - Click **Data Sources**.
    - Click **Add data source**.
    - Search for and select **PostgreSQL**.
    - Configure the data source settings:
        - **Name:** `PostgreSQL_DB` (or anything descriptive)
        - **Host:** `localhost:5432`
        - **Database:** `mydatabase`
        - **User:** `myuser`
        - **Password:** `mypassword`
        - **SSL Mode:** `disable` (for local development, for production, use `require` or `verify-ca` with proper certificates).
        - **PostgreSQL Version:** Select the version you installed.
        - **Min time interval:** (Optional, but good for real-time dashboards) Set this to match your `poll.interval.ms` in Kafka Connect, e.g., `5s`.
    - Click **Save & Test**. You should see "Database Connection OK".

Part 5: Creating Grafana Dashboards for Visualization

Now that everything is connected, let's create some dashboards.

Static Data Visualization:

This involves querying the data that is already in your PostgreSQL database.

1. **Create a New Dashboard:**
    - Click the **+ icon** on the left-hand menu.
    - Click **Dashboard**.
    - Click **Add new panel**.
2. **Configure Panel (e.g., Table for static view):**
    - In the **Query** tab, select your `PostgreSQL_DB` data source.
    - Enter a SQL query to retrieve data from your `sensor_data` table:
    - SQL

```
SELECT id, temperature, humidity, location, updated_at FROM sensor_data ORDER BY updated_at DESC;
```

    -
    -
    - In the **Visualization** tab, select `Table`.
    - You can customize columns, apply transformations, etc.
    - Click **Apply** to save the panel.
3. **Add More Static Panels:**
    - You can add panels for other types of static visualization:
        - **Stat:** To show a single value (e.g., `SELECT AVG(temperature) FROM sensor_data;`).
        - **Gauge:** Similar to Stat, but with thresholds.
        - **Bar Chart/Pie Chart:** For categorical data (e.g., `SELECT location, COUNT(*) FROM sensor_data GROUP BY location;`).

Real-time Data Visualization:

This relies on Kafka Connect continuously pushing data to PostgreSQL, and Grafana querying that updated data.

1. **Update Data in PostgreSQL:**
    - Go back to your psql shell (connected to `mydatabase` as `myuser`).
    - Insert new data or update existing data in the `sensor_data` table. For example:
    - SQL

```
INSERT INTO sensor_data (temperature, humidity, location) VALUES (26.0, 61.5, 'Kitchen');

UPDATE sensor_data SET temperature = 25.8, updated_at = NOW() WHERE id = 1;
```

    -
    -
    - Kafka Connect will pick up these changes and publish them to the `postgres_sensor_data` topic.
2. **Create a New Panel for Real-time (Time Series):**
    - On your Grafana dashboard, click **Add new panel**.
    - In the **Query** tab, select `PostgreSQL_DB`.
    - Enter a SQL query suitable for time-series data. Grafana's `$__timeGroup` and `$__timeFilter` macros are very useful here:
    - SQL

```
SELECT
  $__timeGroup(updated_at, '5s') AS time,
  AVG(temperature) AS avg_temperature,
  AVG(humidity) AS avg_humidity
FROM
  sensor_data
WHERE
  $__timeFilter(updated_at)
GROUP BY 1
ORDER BY 1
```

    -
        - This query groups data by 5-second intervals (adjust `'5s'` as needed) and calculates the average temperature and humidity within that interval. `$__timeFilter` automatically applies the time range selected in Grafana.
    - In the **Visualization** tab, select `Time Series`.
    - **Set Refresh Rate:** In the top right corner of the dashboard, you can set the auto-refresh interval (e.g., `5s`, `10s`). This is key for "real-time" updates.
    - Click **Apply**.
3. **Experiment with Real-time Updates:**
    - Keep your PostgreSQL psql window open, inserting/updating data.

    - Observe the Grafana dashboard. The `Time Series` panel should update automatically as new data arrives and the dashboard refreshes.

Important Considerations and Troubleshooting:

- **Firewall:** Ensure your Windows Firewall allows inbound connections for PostgreSQL (port 5432) and Grafana (port 3000) if you plan to access them from other machines. For local testing, it's usually fine, but if you have issues, check firewall rules.
- **Environment Variables:** Make sure `JAVA_HOME` is set correctly for Kafka. You might need to restart your command prompts after setting environment variables.
- **Logs:**
    - **Kafka/ZooKeeper:** Check the command prompt windows where you started them for error messages.
    - **Kafka Connect:** Check the `connect-standalone.bat` command prompt for errors related to the connector.
    - **Grafana:** Grafana logs are typically in `C:\Program Files\GrafanaLabs\grafana\data\log` (or your chosen data directory).
- **Kafka Connect Modes:**
    - `mode=timestamp+incrementing`: Best for capturing changes (updates and inserts) in a table with both a timestamp and an incrementing ID.
    - `mode=incrementing`: Only for new rows with an incrementing ID.
    - `mode=timestamp`: Only for new rows based on a timestamp column.
    - `mode=bulk`: Reads the entire table once at startup. Not suitable for real-time.
- **Primary Keys:** For Kafka Connect to correctly identify and update/insert records, your PostgreSQL tables should have a primary key.
- **Kafka Connect Sink (PostgreSQL -> Kafka -> PostgreSQL):**
    - If you want to complete the loop (PostgreSQL -> Kafka -> PostgreSQL), you'd configure a `JdbcSinkConnector`.
    - You'd create a *second* PostgreSQL table (e.g., `sensor_data_sink`) in the same or a different database.
    - The `pg-sink.properties` would look something like this:
    - Properties

```
name=postgresql-sink-connector

connector.class=io.confluent.connect.jdbc.JdbcSinkConnector

tasks.max=1

connection.url=jdbc:postgresql://localhost:5432/mydatabase
```

```
connection.user=myuser

connection.password=mypassword

topics=postgres_sensor_data # The topic from your source connector

auto.create=true # Automatically create table if it doesn't exist

pk.mode=record_value # Use a field in the record value as primary key

pk.fields=id # The field in the Kafka message that acts as primary key
```

    -
    -
    - You'd then run `connect-standalone.bat ..\..\config\connect-standalone.properties ..\..\config\pg-source.properties ..\..\config\pg-sink.properties` to run both connectors simultaneously.
- **Data Formats in Kafka:** The JDBC Source Connector by default outputs data in a JSON format with schema information. Grafana's PostgreSQL data source reads directly from PostgreSQL, so the Kafka data format is mostly relevant for the Kafka Connect part itself.

This detailed guide should help you get your PostgreSQL, Kafka, and Grafana setup running on Windows for both static and real-time data visualization! Remember to work through each step methodically and check for errors in your command prompts.

(TO CHANGE PASSWORD: ALTER USER myuser WITH PASSWORD 'MyNewStrongP@ssword';)

TROUBLESHOOTING IF PERMISSION DENIED:

\c mydatabase;

GRANT SELECT ON TABLE sensor_data TO myuser;

\dp sensor_data

\q

Restart Kafka Connect:

- Go to the command prompt where Kafka Connect is running.
- Press `Ctrl+C` to stop it.
- Re-run the command:

connect-standalone.bat ..\..\config\connect-standalone.properties ..\..\config\pg-source.properties

More queries:

Now that your Kafka Connect is successfully streaming data from PostgreSQL to Kafka, and your Grafana is connected to PostgreSQL, let's explore more queries to build and test various visualizations in Grafana.

We'll assume your `sensor_data` table looks like this (or similar):

```
SQL

CREATE TABLE sensor_data (

    id SERIAL PRIMARY KEY,

    temperature DECIMAL(5, 2),

    humidity DECIMAL(5, 2),

    location VARCHAR(100),

    updated_at TIMESTAMP NOT NULL DEFAULT NOW()

);
```

And you're actively inserting/updating data into it.

General Grafana Querying Tips:

- **Data Source:** Always select your PostgreSQL data source (e.g., `PostgreSQL_DB`) in the panel's Query editor.
- **Time Series Panels:** For "Time Series" visualizations, use the Grafana macros `$__timeGroup`, `$__timeFilter`, and `$__timeEpoch` (or `$__timeUnixEpoch`) for dynamic time range and grouping.
- **Table Panels:** For "Table" visualizations, simple `SELECT` statements are usually sufficient.
- **Variables:** Consider using Grafana variables for dynamic filtering (e.g., a dropdown to select a `location`).

Query Examples for Different Panel Types:

1. Time Series Panel (Line Graph, Area Graph)

This is the primary way to visualize real-time trends.

Query 1: Average Temperature and Humidity Over Time

Shows the trend of average temperature and humidity across all locations.

```
SQL

SELECT

  $__timeGroup(updated_at, '$__interval') AS time,

  AVG(temperature) AS avg_temperature,

  AVG(humidity) AS avg_humidity

FROM

  sensor_data

WHERE

  $__timeFilter(updated_at)

GROUP BY 1

ORDER BY 1

```

- **$__interval**: Grafana automatically replaces this with an appropriate time interval (e.g., 1m, 5m, 1h) based on your dashboard's time range and panel width, ensuring optimal data density.
- **Panel Type:** Time Series (or Graph)
- **Series:** avg_temperature, avg_humidity

Query 2: Temperature Trend per Location (Requires a Grafana Variable)

First, create a Grafana variable for location:

- Go to Dashboard settings (gear icon) -> Variables -> Add variable.
- **Name:** location

- **Type:** Query
- **Data source:** Your PostgreSQL_DB
- **Query:** SELECT DISTINCT location FROM sensor_data ORDER BY location;
- **Selection Options:** Multi-value and Include All option (optional, but good)

Then, use this query in your panel:

```
SQL

SELECT

  $__timeGroup(updated_at, '$__interval') AS time,

  AVG(temperature) AS avg_temperature

FROM

  sensor_data

WHERE

  $__timeFilter(updated_at) AND location IN ($location)

GROUP BY 1

ORDER BY 1
```

- **location IN ($location)**: This uses the Grafana variable. If All is selected, Grafana expands it into location IN ('Living Room', 'Bedroom', 'Kitchen') (or whatever values exist).
- **Panel Type:** Time Series

2. Stat Panel (Single Value)

For displaying a single, important metric.

Query 3: Current (Last Recorded) Temperature

```
SQL

SELECT
```

```
  temperature
FROM
  sensor_data
ORDER BY updated_at DESC
LIMIT 1
```

- **Panel Type:** Stat
- **Value options:**
  - **Show:** All values (select temperature)
  - **Calculation:** Last* (or Last not null)

Query 4: Average Temperature Over Last Hour (or selected time range)

```
SQL
SELECT
  AVG(temperature)
FROM
  sensor_data
WHERE
  $__timeFilter(updated_at)
```

- **Panel Type:** Stat
- **Value options:**
  - **Show:** All values
  - **Calculation:** Mean (or Average)

3. Gauge Panel

Similar to Stat, but with visual thresholds.

Query 5: Current Humidity with Thresholds

```
SQL

SELECT
  humidity
FROM
  sensor_data
ORDER BY updated_at DESC
LIMIT 1

```

- **Panel Type:** Gauge
- **Value options:**
  - **Show:** All values (select humidity)
  - **Calculation:** Last*
- **Thresholds:** Configure these in the "Thresholds" section of the panel options (e.g., 0-40 (green), 40-70 (yellow), 70-100 (red) for humidity).

4. Table Panel

For displaying raw or aggregated tabular data.

Query 6: Most Recent Sensor Readings (Top 10)

```
SQL

SELECT
  updated_at,
  location,
  temperature,
  humidity,
```

```
  id
FROM
  sensor_data
ORDER BY updated_at DESC
LIMIT 10
```

- **Panel Type:** Table
- You can use Transform options to organize columns, add value mappings, etc.

Query 7: Average Readings per Location (Over time range)

```
SQL
SELECT
  location,
  AVG(temperature) AS avg_temp,
  AVG(humidity) AS avg_hum,
  COUNT(*) AS data_points
FROM
  sensor_data
WHERE
  $__timeFilter(updated_at)
GROUP BY
  location
```

```
ORDER BY
  location
```

- **Panel Type:** Table

5. Bar Chart / Pie Chart Panel

For visualizing distributions or counts.

Query 8: Number of Readings per Location (Current Time Range)

```
SQL
SELECT
  location,
  COUNT(*) AS count
FROM
  sensor_data
WHERE
  $__timeFilter(updated_at)
GROUP BY
  location
ORDER BY
  count DESC
```

- **Panel Type:** Bar Chart (or Pie Chart)
- **Field settings:** Map `location` to "Category" and `count` to "Value".

6. Heatmap Panel (Requires specific data structure)

Good for showing how a metric changes over two dimensions (e.g., time and temperature range). Requires a `TIME`, `X-BUCKET`, `Y-BUCKET`, and `VALUE` structure.

Query 9: Temperature Distribution by Hour of Day

This is an advanced query and might require creating an additional column or using a subquery to extract the hour from `updated_at`.

```
SQL

SELECT

  $__timeGroup(updated_at, '$__interval') AS time,

  -- Extract hour of the day for the Y-axis (or another categorical bucket)

  EXTRACT(HOUR FROM updated_at) AS hour_of_day,

  -- Aggregate a value, e.g., average temperature for that hour bucket

  AVG(temperature) AS avg_temperature_per_hour

FROM

  sensor_data

WHERE

  $__timeFilter(updated_at)

GROUP BY 1, 2

ORDER BY 1, 2
```

- **Panel Type:** Heatmap
- **Data options:** Configure `Time`, `X-Bucket (hour_of_day)`, `Y-Bucket (this might need to be adjusted based on desired output)`, `Value`. This can be tricky to set up without precise bucket definitions.

How to Build and Test in Grafana:

1. **Add a new Dashboard:** Click the `+` icon -> `Dashboard`.
2. **Add a new Panel:** Click `Add new panel`.
3. **Select Data Source:** In the "Query" tab, ensure `PostgreSQL_DB` (or whatever you named your data source) is selected.
4. **Enter SQL Query:** Paste one of the queries above into the SQL editor.
5. **Choose Visualization:** Go to the "Visualization" tab and select the appropriate panel type (Time Series, Stat, Table, etc.).
6. **Configure Panel Options:** Adjust settings in the "Panel options," "Legend," "Axes," "Thresholds," etc., tabs as needed for each visualization.
7. **Apply Changes:** Click "Apply" to add the panel to your dashboard.
8. **Save Dashboard:** Click the save icon (floppy disk) at the top right to save your dashboard.
9. **Set Time Range & Refresh:** Use the time range picker at the top right of the dashboard to select a range (e.g., "Last 5 minutes") and enable auto-refresh (e.g., "5 seconds") to see your real-time data flow.

Experiment with these queries and different panel types. The more you play with them, the better you'll understand Grafana's capabilities for visualizing your streamed PostgreSQL data!