**Real-Time Log Analytics Service**

**The Architect Phase:**

**Problem Statement:** Application teams currently lack a centralized, real-time system to analyze high-volume log data. This fragmentation delays incident response, makes performance monitoring difficult, and prevents effective analysis of user behavior.

**Define the Requirements**:

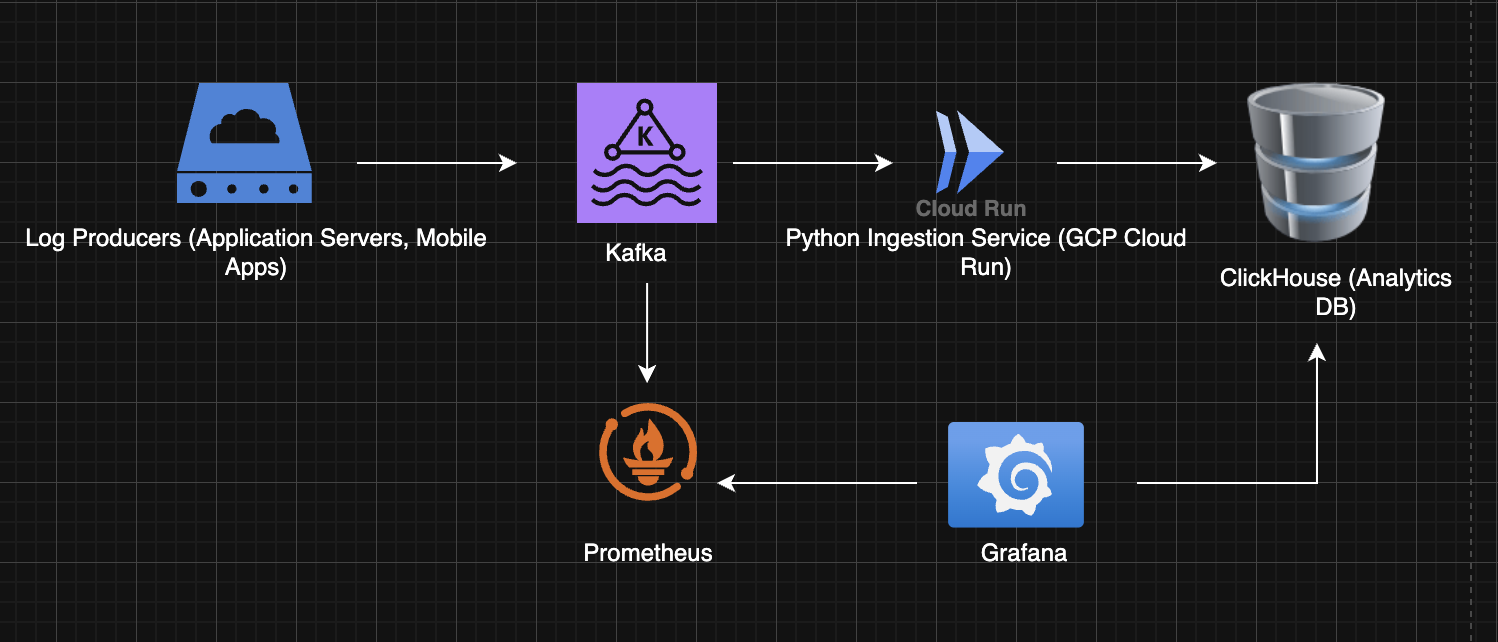
**Functional Requirements (What it does):**

* Ingest log events from multiple sources.
* Store log data efficiently.
* Provide an interface for fast, real-time queries.
* Visualize key metrics on a dashboard.

**Non-Functional Requirements (How well it does it):**

* **Throughput:** Must process over 1 million events per day.
* **Performance:** P99 query latency must be under 400ms.
* **Scalability:** The ingestion service must scale automatically with load.
* **Durability:** The system must guarantee no data loss during ingestion.

**The Architecture Diagram:**



**Technology Justification:**

1. **GCP Cloud Run:**Chosen for its serverless, auto-scaling capabilities. It allows us to handle variable log traffic cost-effectively without managing any underlying server infrastructure
2. **Apache Kafka:**Selected as a fault-tolerant message bus. It decouples our log producers from the consumer service, providing a durable buffer that prevents data loss during service outages or traffic spikes
3. **ClickHouse:**Its columnar storage engine is purpose-built for high-speed analytical queries (OLAP). This is critical for meeting our requirement of sub-400ms query performance on terabytes of log data.
4. **Prometheus & Grafana:**Chosen as the industry-standard open-source stack for monitoring and visualization. Prometheus provides robust time-series data collection, while Grafana offers a powerful and flexible platform for creating insightful dashboards.

**The Engineer Phase:**

**Python Ingestion Service**:

1. Create database schema
2. Build Your Local Environment : pinning up the core backend services (Kafka + ClickHouse) on your own laptop using Docker.
3. **Docker Compose:**

* Docker Compose is like a recipe book.
* It lets you describe multiple services (Kafka, ClickHouse, Zookeeper) in one YAML file and start them all together with a single command (docker-compose up).

1. **Zookeeper**

* Required by **Kafka**.
* Manages Kafka brokers, keeps track of who’s the leader, coordinates distributed state.
* You don’t use Zookeeper directly — it’s just plumbing for Kafka.

1. **Kafka (Message Broker)**

* Think of Kafka as your **“data bus.”**
* Producers (apps, servers) send logs → Kafka stores them temporarily → Consumers (like your Python service) read them.
* It’s designed for high-throughput, fault-tolerant streaming.

1. **ClickHouse (Analytics Database)**
   * A super-fast database optimized for **analytics and logs**.
   * Instead of traditional row-based storage (like Postgres), it uses **columnar storage** — making queries over billions of rows very fast.
   * This is where your logs will ultimately be stored for querying and dashboards.
2. **Docker Volumes**  
    volumes:

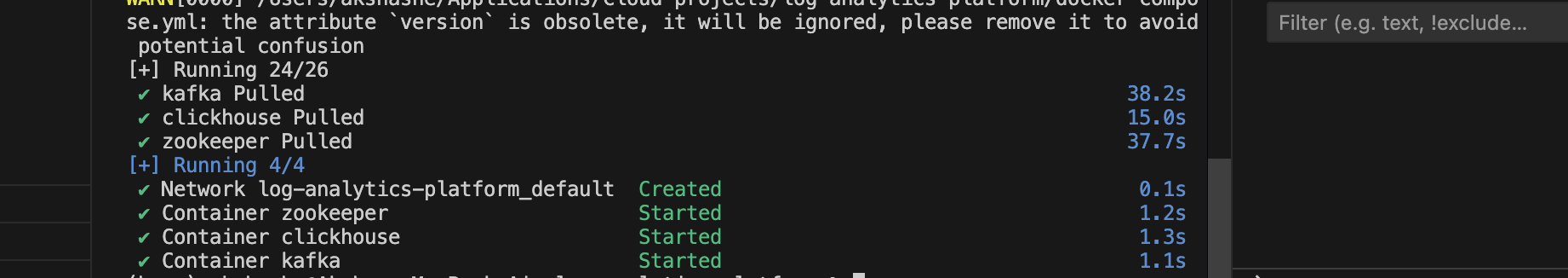
- ./clickhouse-data:/var/lib/clickhouse

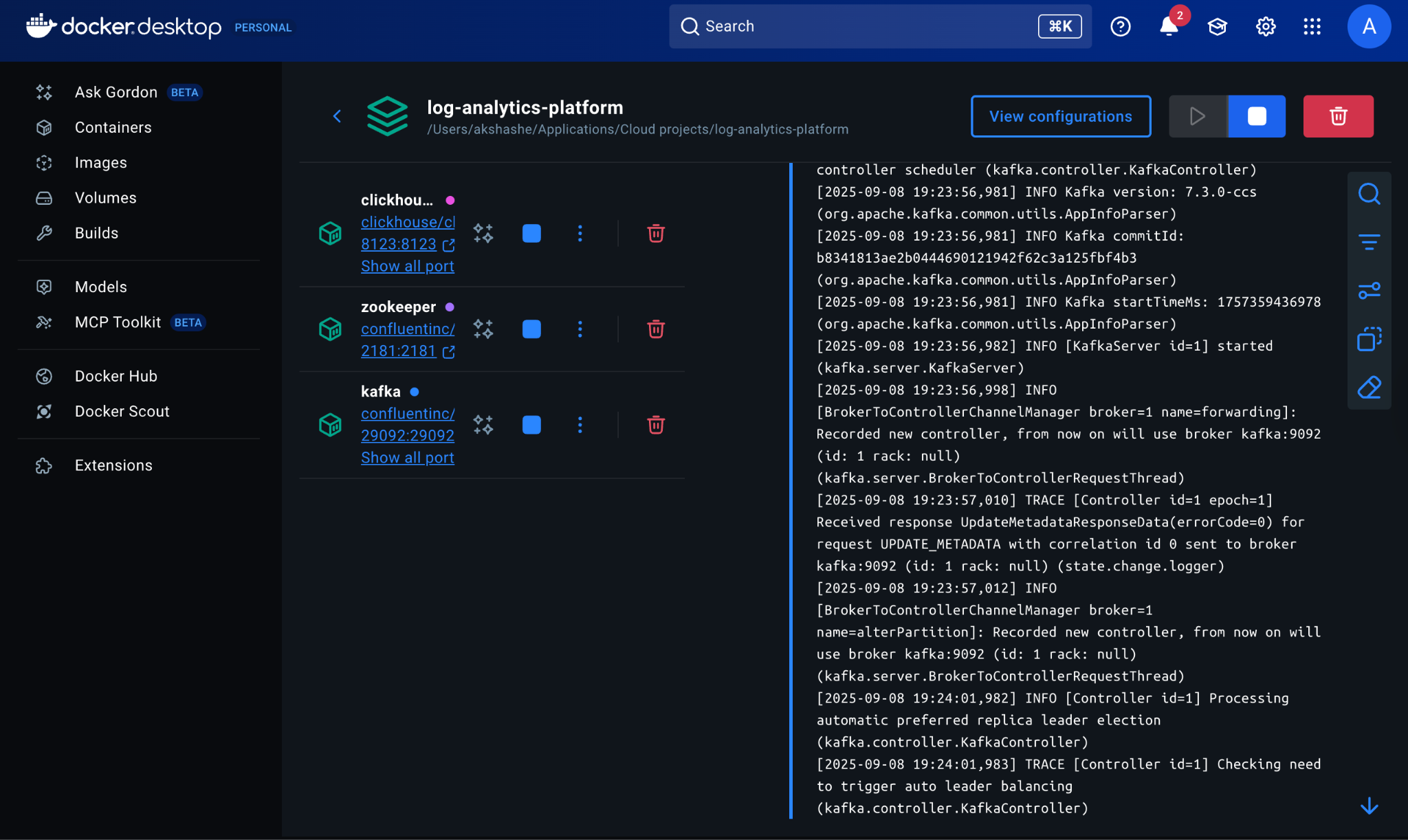
* + That means: “Save database files to my local folder clickhouse-data.”
  + Without this, data would vanish every time you stopped the container.

The flow:

**Producers** ( your test apps) → send logs → **Kafka**

**Python ingestion service** → consumes logs → stores them into **ClickHouse  
Grafana** → queries ClickHouse + Prometheus → shows dashboards





Zookeeper: The Coordinator

Of course. Let's break down what each service in your docker-compose.yml file does. Think of this file as a blueprint for creating a mini, virtual data center on your computer.

## **Zookeeper: The Coordinator 🐘**

**What it is:** Apache Zookeeper is a centralized service for maintaining configuration information, naming, and providing distributed coordination.

**Its role in our project:** Zookeeper acts as the **manager or "brain" for Kafka**. In older Kafka versions, it was responsible for keeping track of critical metadata, such as:

* Which brokers (Kafka servers) are currently alive.
* Who is the "leader" for a specific data topic.
* Configuration settings for all topics.

In short, Kafka brokers talk to Zookeeper to coordinate their work. Without it, the Kafka cluster wouldn't know how to function.

Of course. Let's break down what each service in your docker-compose.yml file does. Think of this file as a blueprint for creating a mini, virtual data center on your computer.

## **Zookeeper: The Coordinator 🐘**

**What it is:** Apache Zookeeper is a centralized service for maintaining configuration information, naming, and providing distributed coordination.

**Its role in our project:** Zookeeper acts as the **manager or "brain" for Kafka**. In older Kafka versions, it was responsible for keeping track of critical metadata, such as:

* Which brokers (Kafka servers) are currently alive.
* Who is the "leader" for a specific data topic.
* Configuration settings for all topics.

In short, Kafka brokers talk to Zookeeper to coordinate their work. Without it, the Kafka cluster wouldn't know how to function.

**Key Configuration:** The line KAFKA\_ZOOKEEPER\_CONNECT: 'zookeeper:2181' in the Kafka service is the explicit instruction telling Kafka where to find its manager.

**Note:** Newer versions of Kafka can now run without Zookeeper in a mode called KRaft, but using Zookeeper is still very common in many production environments and is great for learning.

## **Kafka: The Data Superhighway**

## **What it is:** Apache Kafka is a distributed event streaming platform. It's designed to move huge amounts of data from a source to a destination, reliably and in real-time.

**Its role in our project:** Kafka is our **central message bus**. It acts like a highly organized post office.

1. **Log Producers** (your applications) send log messages (letters) to a specific Kafka **Topic** (a P.O. Box).
2. Kafka stores these messages durably for a configured amount of time.
3. **Your Python Service** (the consumer) connects to Kafka and picks up the messages from the topic to process them.

It creates a buffer that separates the application writing the logs from the service processing them. If your Python service goes down for 5 minutes, Kafka holds onto the logs, and the service can pick them right up when it comes back online.

## **ClickHouse: The Analytics Engine**

**What it is:** ClickHouse is a high-performance, columnar database management system.

**Its role in our project:** This is our **specialized warehouse and query engine**. While a normal database stores data in rows (like a spreadsheet), ClickHouse stores it in columns. This structure makes it incredibly fast for analytical queries that scan and aggregate large amounts of data, like:

* "Count all ERROR logs from the payment-service in the last hour."
* "Find the average response time for user\_id '123'."

This is why it can meet our sub-400ms query performance goal.