CI/CD and Design of a Distributed System

Tomas Conti

January 19, 2026

Abstract

This document details the design, implementation, and validation of a distributed system engineered for high availability and fault tolerance. The architecture integrates a Spring Boot microservices framework with Apache Kafka for asynchronous event streaming and MongoDB for resilient persistent storage. A significant portion of this work is dedicated to modern DevOps methodologies, focusing on the establishment of reliable CI/CD pipelines and a containerized deployment workflow designed to ensure system stability in production-like environments.

1 Introduction

Distributed systems offer substantial advantages in scalability and modularity; however, they introduce inherent complexities regarding inter-service communication, data consistency, and operational resilience. Addressing these challenges necessitates the adoption of specific architectural patterns—such as the Producer-Consumer model and Message-Oriented Middleware (MOM)—alongside robust coordination mechanisms.

Developed as the final project for the Distributed Systems course, this work bridges the gap between theoretical distributed computing principles and practical engineering. The core of the application leverages a three-node Kafka cluster to guarantee data integrity and partition tolerance. A primary objective is the exploration of the software's full lifecycle: from containerization with Docker to orchestration via Docker Compose. By simulating real-world failure scenarios, the project validates the system's "self-healing" capabilities and its ability to maintain service continuity under stress.

The complete source code, infrastructure-as-code configurations, and extended documentation are hosted on GitHub at: TomasConti02/DistributedSystemsProject.

2 Technologies

In this chapter, I will provide a quick overview of the main technologies and frameworks used to build, containerize, and orchestrate the distributed application. Each tool has been selected to address specific challenges such as modularity, asynchronous communication, and environment consistency.

2.1 SpringBoot

Spring Boot is the core framework used for developing the microservices within this system. It provides a robust ecosystem for building stand-alone, production-grade Spring-based applications. In this project, it is utilized to simplify the bootstrapping and development of RESTful APIs, leveraging its dependency injection and auto-configuration capabilities to maintain a clean and modular codebase. For the project i decide to import Kafka, and MonogoDB extantion to semplify the configuration of spring into the cluster.

The following list describes the key **listener components** and the architectural layers of the application:

- **Tutorial Entity:** This class represents the MongoDB document mapped to the `tutorials` collection. It consists of a unique identifier (`id`) and a `title` (String), acting as the Data Transfer Object for the persistence layer.
- **TutorialRepository:** An interface that abstracts data access to MongoDB. Although MongoDB is a NoSQL database, this component follows the **Repository Pattern** (provided by Spring Data), enabling standard CRUD operations and seamless database interaction without boilerplate code.
- **TutorialService:** This service layer encapsulates the core business logic. It handles the instantiation of `Tutorial` objects and coordinates data persistence by interfacing between the entry points (Kafka/REST) and the repository.
- **KafkaConsumerService:** A dedicated message listener for the `tutorials-topic` Kafka topic. Upon message ingestion, it invokes the service layer to persist data into MongoDB, facilitating asynchronous communication within the distributed system.
- **TutorialController:** A RESTful controller that exposes the `/api/test` endpoint. It processes incoming HTTP GET requests and returns a JSON response containing the full dataset retrieved from the database.

The **Producer** module is designed to handle data ingestion and message dispatching through the following components:

- **ProducerController:** This RESTful controller serves as the system's entry point, exposing a `POST` endpoint at `/api/publish`.
- **KafkaProducerService:** Its primary role is to encapsulate the logic required to publish string-based payloads to the `"tutorials-topic"`. By doing so, it effectively bridges the gap between the synchronous execution of the HTTP request and the asynchronous nature of the Kafka message broker.

2.2 Apache Kafka

Apache Kafka serves as a distributed event streaming platform, enabling asynchronous communication between the microservices of this architecture. By implementing a publish-subscribe messaging pattern, Kafka ensures high throughput, fault tolerance, and loose coupling between components.

In this project, Kafka acts as the backbone for data ingestion. It is utilized to decouple the **Producer** module, which receives tutorial data via REST API, from the **Consumer** module, which manages persistence. Specifically, the `tutorials-topic` is used to stream tutorial titles in real-time. This ensures that the system remains highly responsive: the Producer can acknowledge the receipt of data immediately after publishing it to the broker, while the Consumer processes and stores it in MongoDB independently, effectively managing potential spikes in load without affecting the user experience.

2.3 Docker

Docker is employed to containerize each architectural component, ensuring environmental consistency and reproducibility. By encapsulating the Spring Boot applications, the Kafka brokers, and the MongoDB instance into immutable containers, Docker effectively mitigates the "it works on my machine" phenomenon.

In this project, Docker serves as the foundational layer for portability. Each microservice (Producer and Consumer) is built into a lightweight image using a multi-stage build process, optimizing the final size and security. This containerization strategy allows for the identical execution of the entire stack across the development team's local machines and the final deployment environment.

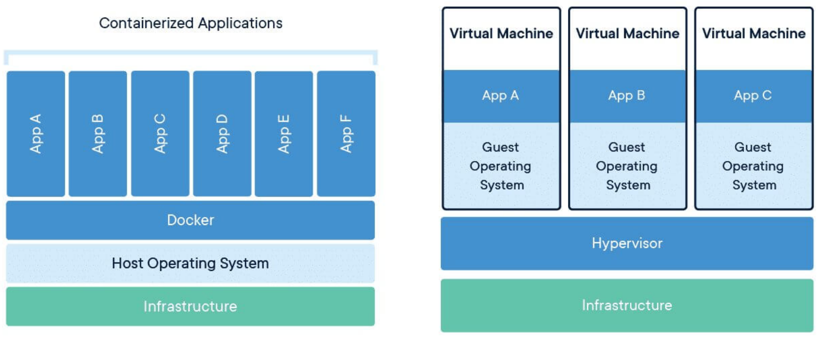


Figure 1: Containerization of the microservices and infrastructure components.

2.4 Kubernetes

Kubernetes (K8s) is the orchestration platform adopted to manage the lifecycle and scaling of the containerized services. While Docker handles the individual containers, Kubernetes coordinates the cluster, automating deployment and ensuring high availability.

Within the scope of this work, Kubernetes is leveraged to transform the application into a resilient distributed system. It manages the **three-node Kafka cluster**, ensuring the cluster identity . Furthermore, K8s provides **self-healing** capabilities: if a Spring Boot pod or a Kafka broker fails, the orchestrator automatically reinstantiates the component to maintain the desired state. Service discovery and internal load balancing are also utilized to allow the Producer to communicate with the Kafka cluster without hardcoding specific IP addresses.

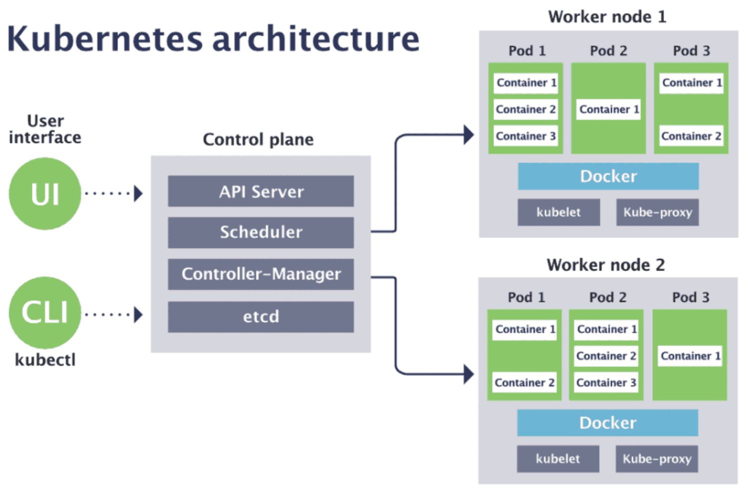
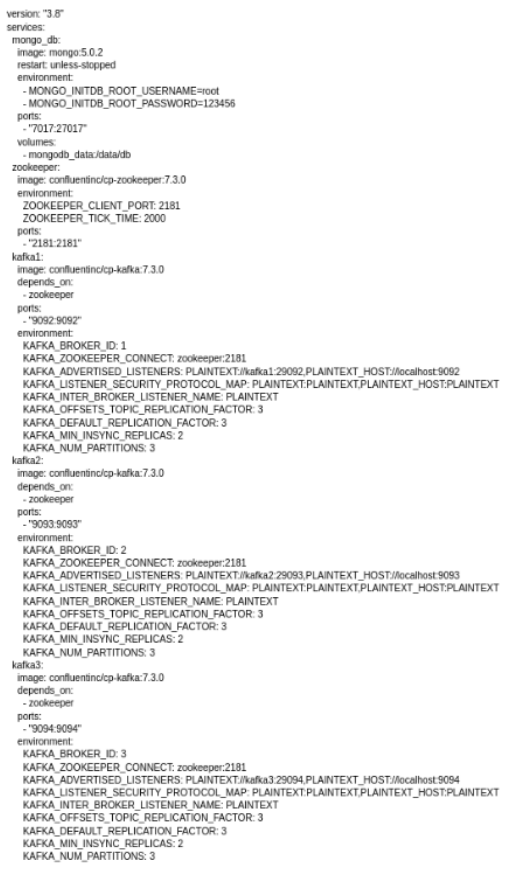


Figure 2: Kubernetes architecture and pod orchestration.

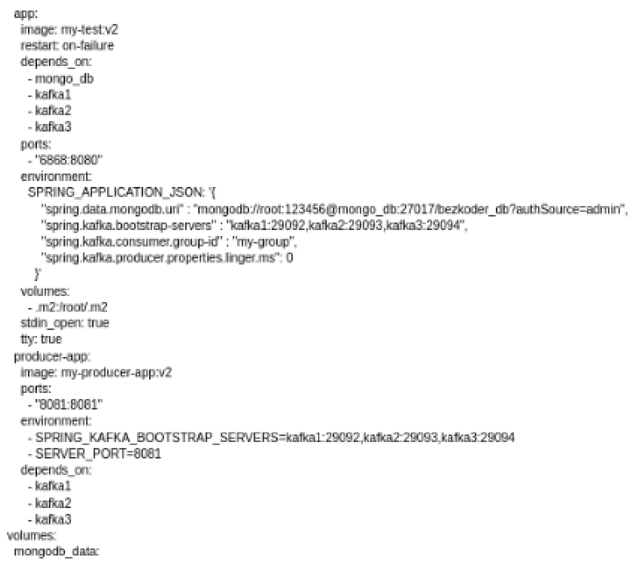
3 System Architecture

The interaction begins when a client sends a request to the Sender service via a REST endpoint. Instead of processing the request synchronously, the Sender produces a message into a Kafka topic. The Receiver service, which subscribes to the relevant topic, consumes the message and performs the final write operation to the MongoDB instance. This design ensures high availability and scalability by decoupling the ingestion layer from the persistence layer.



```
version: "3.8"
services:
  mongo_db:
    image: mongo:5.0.2
    restart: unless-stopped
    environment:
      - MONGO_INITDB_ROOT_USERNAME=root
      - MONGO_INITDB_ROOT_PASSWORD=123456
    ports:
      - "7017:27017"
    volumes:
      - mongodb_data:/data/db
  zookeeper:
    image: confluentinc/cp-zookeeper:7.3.0
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - "2181:2181"
  kafka1:
    image: confluentinc/cp-kafka:7.3.0
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka1:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 3
      KAFKA_DEFAULT_REPLICATION_FACTOR: 3
      KAFKA_MIN_INSYNC_REPLICAS: 2
      KAFKA_NUM_PARTITIONS: 3
  kafka2:
    image: confluentinc/cp-kafka:7.3.0
    depends_on:
      - zookeeper
    ports:
      - "9093:9093"
    environment:
      KAFKA_BROKER_ID: 2
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka2:29093,PLAINTEXT_HOST://localhost:9093
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 3
      KAFKA_DEFAULT_REPLICATION_FACTOR: 3
      KAFKA_MIN_INSYNC_REPLICAS: 2
      KAFKA_NUM_PARTITIONS: 3
  kafka3:
    image: confluentinc/cp-kafka:7.3.0
    depends_on:
      - zookeeper
    ports:
      - "9094:9094"
    environment:
      KAFKA_BROKER_ID: 3
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka3:29094,PLAINTEXT_HOST://localhost:9094
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 3
      KAFKA_DEFAULT_REPLICATION_FACTOR: 3
      KAFKA_MIN_INSYNC_REPLICAS: 2
      KAFKA_NUM_PARTITIONS: 3
```

(a) Kafka Broker Cluster.



```
  app:
    image: my-test:v2
    restart: on-failure
    depends_on:
      - mongo_db
      - kafka1
      - kafka2
      - kafka3
    ports:
      - "6868:8080"
    environment:
      SPRING_APPLICATION_JSON: '{
        "spring.data.mongodb.uri" : "mongodb://root:123456@mongo_db:27017/bezkoder_db?authSource=admin",
        "spring.kafka.bootstrap-servers" : "kafka1:29092,kafka2:29093,kafka3:29094",
        "spring.kafka.consumer.group-id" : "my-group",
        "spring.kafka.producer.properties.linger.ms": 0
      }'
    volumes:
      - .m2:/root/.m2
    stdin_open: true
    tty: true
  producer-app:
    image: my-producer-app:v2
    ports:
      - "8081:8081"
    environment:
      - SPRING_KAFKA_BOOTSTRAP_SERVERS=kafka1:29092,kafka2:29093,kafka3:29094
      - SERVER_PORT=8081
    depends_on:
      - kafka1
      - kafka2
      - kafka3
volumes:
  mongodb_data:
```

(b) Receiver and Listener.

Figure 3: Docker compose.

This architecture relies on a three-node Kafka cluster to achieve fault tolerance through replication. By maintaining a majority quorum, the system ensures single-fault tolerance. Each topic is partitioned and replicated across all broker instances, ensuring high reliability and data consistency. To guarantee that messages are committed safely, Kafka is configured to require at least two acknowledgments (the majority). In the event of a broker failure, the cluster maintains a consistent state. Consequently, this configuration prioritizes Consistency and Partition Tolerance (CP); while it maintains availability under normal conditions, it sacrifices total availability if a majority of nodes are unreachable, strictly adhering to the single-fault assumption.

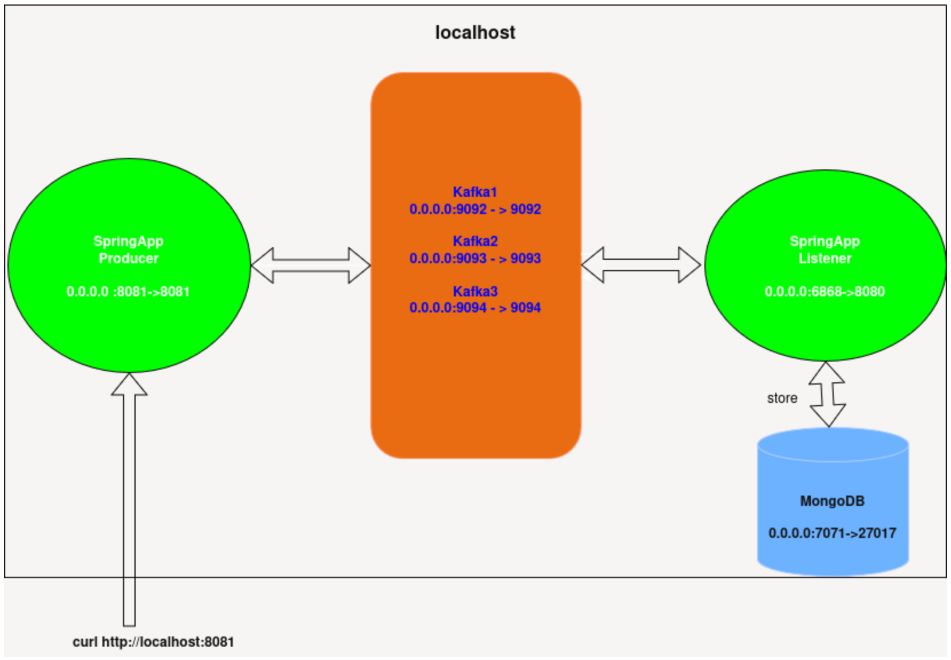
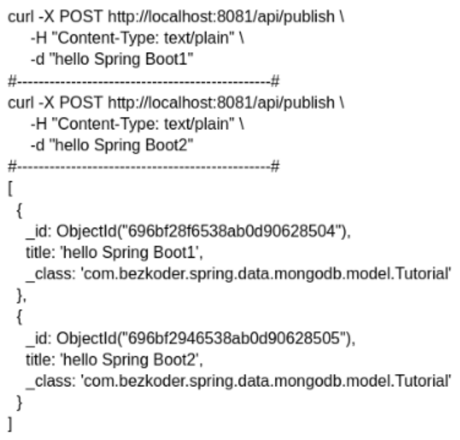


Figure 4: Distributed architecture schema.

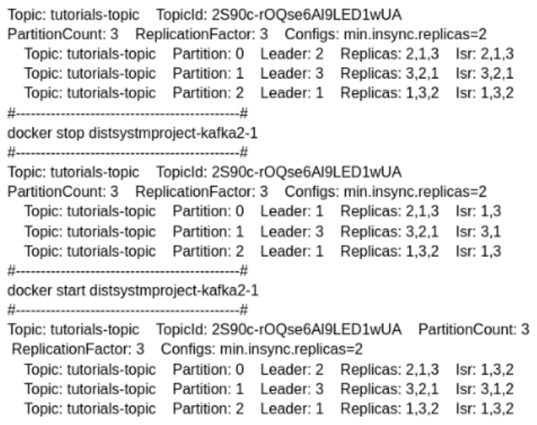
3.1 System testing and validation

I have implemented a local test environment using multi-stage Docker builds to optimize the Spring Boot images. Using Docker Compose, I can deploy the entire application stack locally to validate business logic and functional tasks. Furthermore, I performed resilience testing by simulating a node failure; the system successfully demonstrated its ability to maintain consistency and re-synchronize replicas once the faulted node was restored.



```
curl -X POST http://localhost:8081/api/publish \
    -H "Content-Type: text/plain" \
    -d "hello Spring Boot1"
#------------------------------------------------#
curl -X POST http://localhost:8081/api/publish \
    -H "Content-Type: text/plain" \
    -d "hello Spring Boot2"
#------------------------------------------------#
[
  {
    _id: ObjectId("696bf28f6538ab0d90628504"),
    title: 'hello Spring Boot1',
    _class: 'com.bezkoder.spring.data.mongodb.model.Tutorial'
  },
  {
    _id: ObjectId("696bf2946538ab0d90628505"),
    title: 'hello Spring Boot2',
    _class: 'com.bezkoder.spring.data.mongodb.model.Tutorial'
  }
]
```

(a) Message saving test.



```
Topic: tutorials-topic    TopicId: 2S90c-rOQse6AI9LED1wUA
PartitionCount: 3    ReplicationFactor: 3    Configs: min.insync.replicas=2
    Topic: tutorials-topic    Partition: 0    Leader: 2    Replicas: 2,1,3    Isr: 2,1,3
    Topic: tutorials-topic    Partition: 1    Leader: 3    Replicas: 3,2,1    Isr: 3,2,1
    Topic: tutorials-topic    Partition: 2    Leader: 1    Replicas: 1,3,2    Isr: 1,3,2
#------------------------------------------#
docker stop distsystmproject-kafka2-1
#------------------------------------------#
Topic: tutorials-topic    TopicId: 2S90c-rOQse6AI9LED1wUA
PartitionCount: 3    ReplicationFactor: 3    Configs: min.insync.replicas=2
    Topic: tutorials-topic    Partition: 0    Leader: 1    Replicas: 2,1,3    Isr: 1,3
    Topic: tutorials-topic    Partition: 1    Leader: 3    Replicas: 3,2,1    Isr: 3,1
    Topic: tutorials-topic    Partition: 2    Leader: 1    Replicas: 1,3,2    Isr: 1,3
#------------------------------------------#
docker start distsystmproject-kafka2-1
#------------------------------------------#
Topic: tutorials-topic    TopicId: 2S90c-rOQse6AI9LED1wUA    PartitionCount: 3
 ReplicationFactor: 3    Configs: min.insync.replicas=2
    Topic: tutorials-topic    Partition: 0    Leader: 2    Replicas: 2,1,3    Isr: 1,3,2
    Topic: tutorials-topic    Partition: 1    Leader: 3    Replicas: 3,2,1    Isr: 3,1,2
    Topic: tutorials-topic    Partition: 2    Leader: 1    Replicas: 1,3,2    Isr: 1,3,2
```

(b) Fault tolerance of the cluster.

Figure 5: Validation tests.

The first test verifies the push/pull messaging mechanism through the message broker, ensuring that data is correctly persisted into the database NoSql. The second test evaluates fault tolerance confirms that after a system failure, Kafka restarts and successfully restores the cluster state, including all partitions and replicas, to its original configuration. Another significant advantage of this configuration is decoupling. In the event of a Listener crash, Kafka keeps the messages in the queue and redelivers them once the component has restarted. This behavior is crucial for maintaining system consistency.

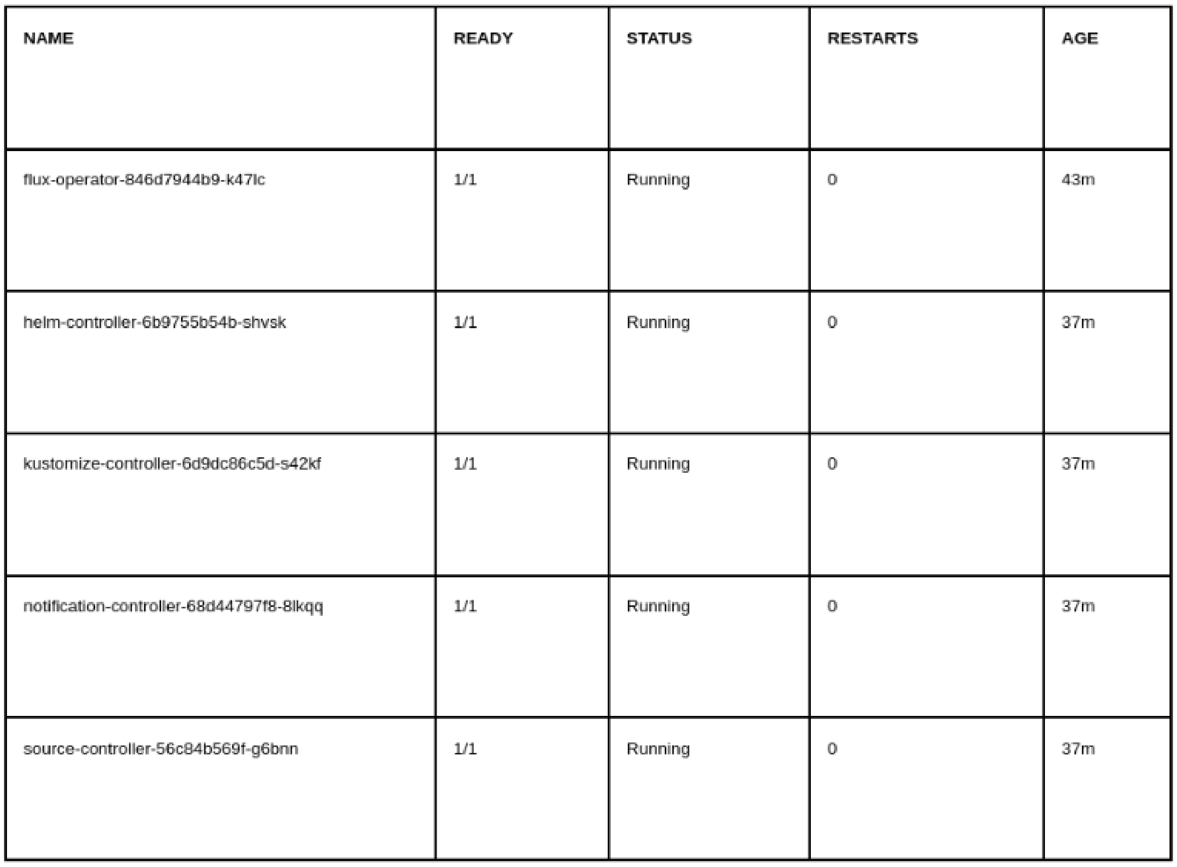
4 System in Production on k8s

After local validation, I provisioned a Kubernetes cluster using the Kind tool, consisting of three worker nodes and a control plane for system deployment. Using FluxCD, I established a link between the Kubernetes cluster and the GitHub repository to enable GitOps-based synchronization. This setup ensures that the cluster state automatically aligns with any commit changes made within the repository.

4.1 Deployment State

Once the Spring Boot images were published to Docker Hub and the Kubernetes manifests were finalized, the system was ready for deployment. FluxCD facilitates seamless synchronization, ensuring the cluster configuration remains consistent with the "source of truth" defined in the GitHub repository.

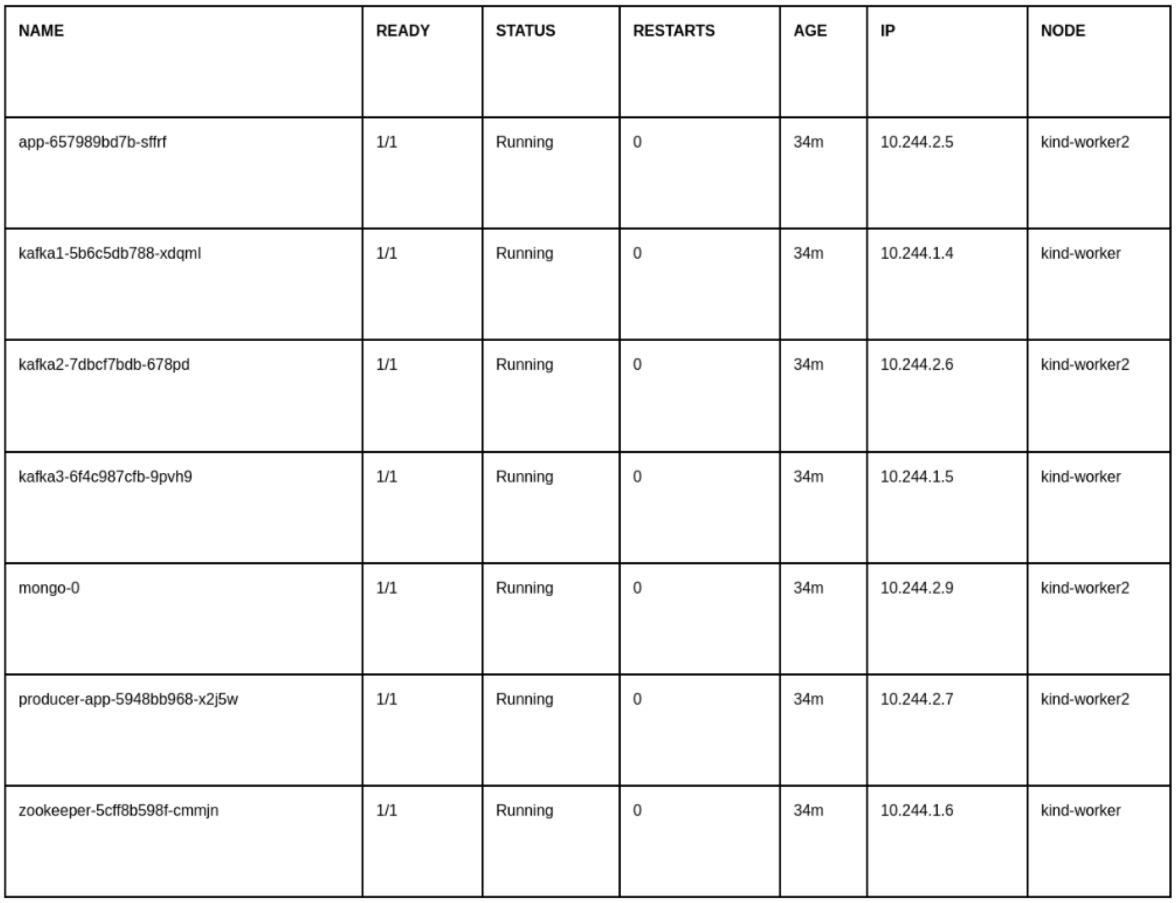
After applying the bootstrap configuration, I verified the operational status of the Flux controllers to ensure the GitOps engine was ready to manage the workloads:



NAME	READY	STATUS	RESTARTS	AGE
flux-operator-846d7944b9-k47lc	1/1	Running	0	43m
helm-controller-6b9755b54b-shvsk	1/1	Running	0	37m
kustomize-controller-6d9dc86c5d-s42kf	1/1	Running	0	37m
notification-controller-68d44797f8-8lkqq	1/1	Running	0	37m
source-controller-56c84b569f-g6bnn	1/1	Running	0	37m

Figure 6: FluxCD controllers operational status.

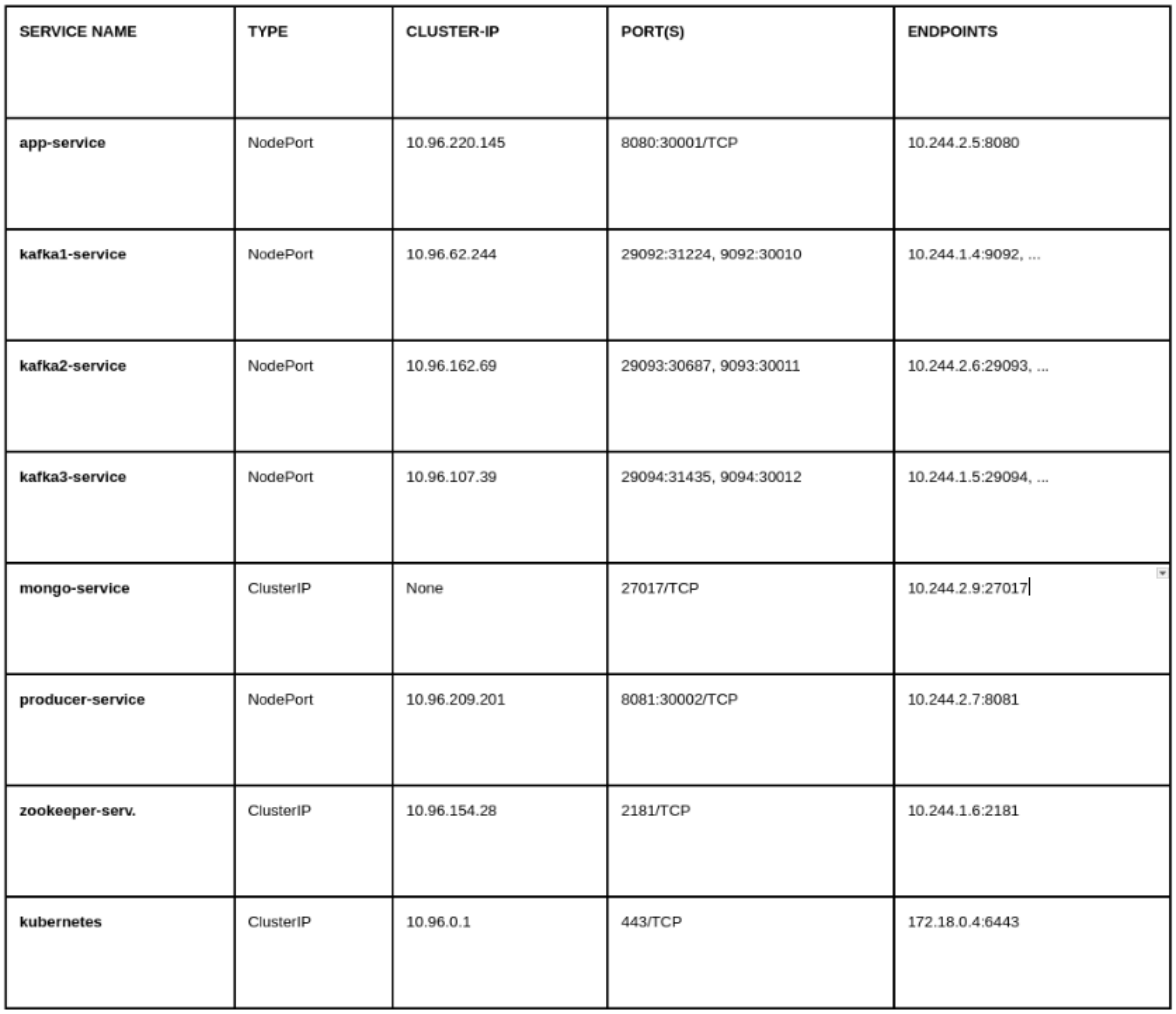
With the controllers active, Flux automatically reconciled the cluster state with the repository. It pulled the architecture definitions and orchestrated the deployment of all application components:



NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
app-657989bd7b-sffrf	1/1	Running	0	34m	10.244.2.5	kind-worker2
kafka1-5b6c5db788-xdqml	1/1	Running	0	34m	10.244.1.4	kind-worker
kafka2-7dbcf7bdb-678pd	1/1	Running	0	34m	10.244.2.6	kind-worker2
kafka3-6f4c987cfb-9pvh9	1/1	Running	0	34m	10.244.1.5	kind-worker
mongo-0	1/1	Running	0	34m	10.244.2.9	kind-worker2
producer-app-5948bb968-x2j5w	1/1	Running	0	34m	10.244.2.7	kind-worker2
zookeeper-5cff8b598f-cmmjn	1/1	Running	0	34m	10.244.1.6	kind-worker

Figure 7: Application pods and services successfully deployed.

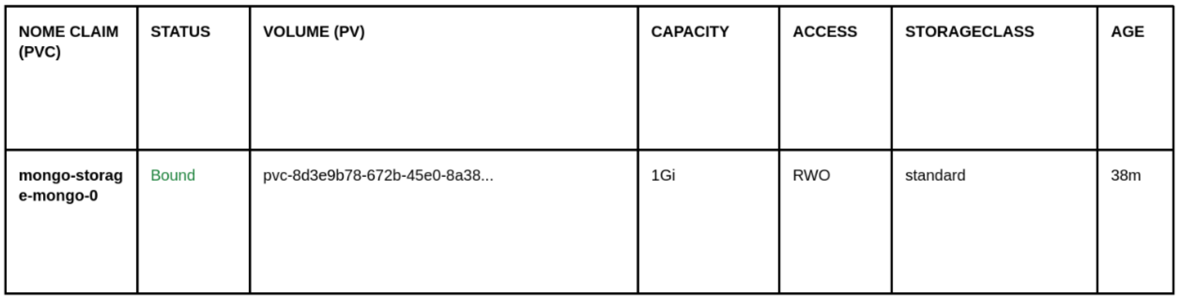
I then verified the network layer, ensuring that NodePort services and Endpoints were correctly mapped to allow both internal communication and external access to the microservices:



SERVICE NAME	TYPE	CLUSTER-IP	PORT(S)	ENDPOINTS
app-service	NodePort	10.96.220.145	8080:30001/TCP	10.244.2.5:8080
kafka1-service	NodePort	10.96.62.244	29092:31224, 9092:30010	10.244.1.4:9092, ...
kafka2-service	NodePort	10.96.162.69	29093:30687, 9093:30011	10.244.2.6:29093, ...
kafka3-service	NodePort	10.96.107.39	29094:31435, 9094:30012	10.244.1.5:29094, ...
mongo-service	ClusterIP	None	27017/TCP	10.244.2.9:27017
producer-service	NodePort	10.96.209.201	8081:30002/TCP	10.244.2.7:8081
zookeeper-serv.	ClusterIP	10.96.154.28	2181/TCP	10.244.1.6:2181
kubernetes	ClusterIP	10.96.0.1	443/TCP	172.18.0.4:6443

Figure 8: Network configuration and service discovery mapping.

Finally, I confirmed the storage provisioning. The binding of Persistent Volume Claims (PVC) to Persistent Volumes (PV) ensures data consistency and persistence for the MongoDB instance across pod restarts:



NOME CLAIM (PVC)	STATUS	VOLUME (PV)	CAPACITY	ACCESS	STORAGECLASS	AGE
mongo-storage-mongo-0	Bound	pvc-8d3e9b78-672b-45e0-8a38...	1Gi	RWO	standard	38m

Figure 9: Persistence layer: PVC and PV binding status.

5 Conclusions

The objective of this document is to demonstrate the application of DevOps patterns to transition a system from a testing environment to a production-ready state. By implementing a set of industry best practices, I achieved a high degree of automation, ensuring a robust and efficient architecture suitable for enterprise-level deployments.

A key component of this architecture is Apache Kafka, utilized as a Message-Oriented Middleware (MOM). This choice reflects the requirements of large-scale enterprise environments, where high throughput, reliability, and asynchronous communication are essential for system scalability.