**Master-Slave Job Execution Architecture**

**Technical Design Document**

**1. Problem Statement**

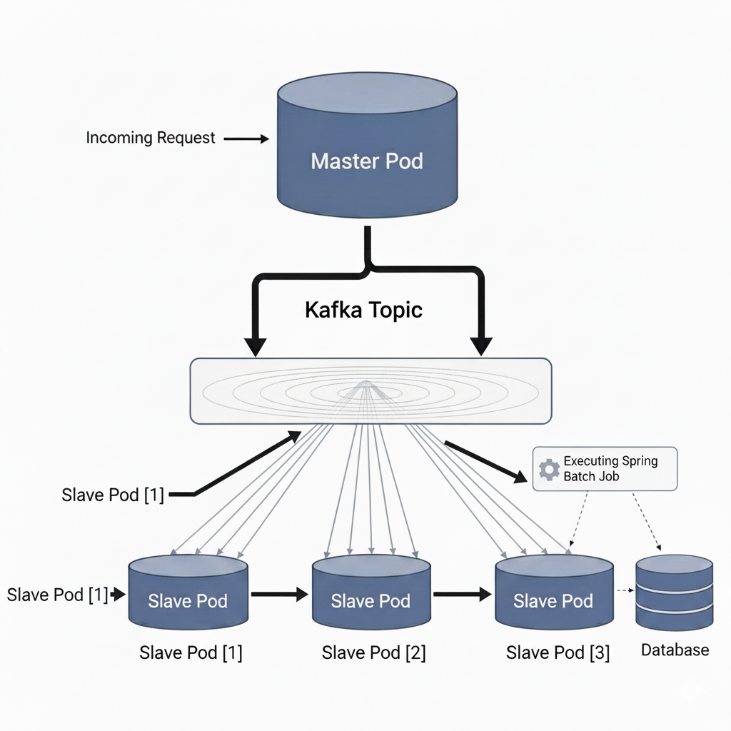
In our production environment, a Spring Boot application runs multiple Spring Batch jobs deployed across multiple Kubernetes pods. Currently, the load balancer routes all incoming requests to a single pod, while the remaining pods remain idle. This leads to:

* **Poor Resource Utilization**: Multiple pods are inactive despite being available.
* **Single Point of Failure**: The active pod becomes a bottleneck and a potential failure point.
* **Limited Scalability**: Inability to leverage the full capacity of the clustered environment.
* **Inefficient Load Handling**: No mechanism for parallel job processing.

**2. Solution Overview**

We implemented a **master-slave architecture** using **Spring Profiles** and **Apache Kafka** to decouple job triggering from job execution. This ensures optimal utilization of all pods and improves scalability and fault tolerance.

**2.1 Architecture Diagram**



**2.2 Key Components**

* **Master Pod**:
  + Handles HTTP requests.
  + Publishes job names to a Kafka topic.
  + Activated using Spring Profile master.
* **Worker Pods**:
  + Listen to the Kafka topic for job requests.
  + Execute jobs upon receiving messages.
  + Activated using Spring Profile worker.

**2.3 Key Benefits**

* **Optimal Resource Utilization:** All pods actively participate in job processing
* **Automatic Load Balancing:** Kafka inherently distributes messages across consumer instances
* **High Availability:** Multiple workers ensure continuous job processing capability
* **Scalability:** Can easily scale workers without affecting the master component
* **Loose Coupling**: Masters and workers operate independently without direct dependencies

**2.4 Core Components**

**Master Component (Profile-based Controller):**

@Profile("master")

public class POAJobTriggerController {

kafkaTemplate.send("job-launch-request", jobName);

}

**Slave Component (Kafka Listener):**

@Profile("worker")

public class JobLaunchKafkaListener {

@KafkaListener(topics = "job-launch-request")

public void listen(String jobName) { triggeredJob(jobName); }

}

**2.5 How It Works**

1. **Request Reception:** The master pod (with 'master' profile) receives HTTP requests through its REST controller
2. **Message Publication:** Instead of executing jobs directly, the master publishes job names to a Kafka topic
3. **Message Consumption:** All worker pods (with 'worker' profile) listen to the Kafka topic
4. **Job Execution:** One worker pod consumes each message and executes the corresponding Spring Batch job
5. **Load Distribution:** Kafka's consumer group mechanism automatically distributes messages among available workers

**3. Deployment Strategy**

**3.1 Kubernetes Deployment**

* **Master Deployment**:
  + Single replica with profile master.
  + Handles HTTP requests and publishes to Kafka.
* **Worker Deployment**:
  + Multiple replicas (3–4) with profile worker.
  + Listens to Kafka and executes jobs.

**3.2 Profile Activation**

The solution uses Spring Profiles to differentiate between master and worker components:

* Master pods activate the 'master' profile
* Worker pods activate the 'worker' profile
* Same application codebase used for both deployment types
* Profile-specific components automatically enabled/disabled

**3.3 Build Application**:

bash

./gradlew clean build -x test

**Run Single Master Profile**:  
The same application codebase can be run in either master or worker mode by activating the corresponding Spring profile:

bash

*# Run as MASTER (API controller active, Kafka listener disabled)*

./gradlew bootRun --args='--spring.profiles.active=master'

**Running Multiple Worker Instances**:  
For local development and testing, you can simulate multiple worker pods by running the worker profile in multiple terminal sessions (3-4 times):

bash

*# Terminal 1 - First worker instance*

./gradlew bootRun --args='--spring.profiles.active=worker --server.port=8081'

*# Terminal 2 - Second worker instance*

./gradlew bootRun --args='--spring.profiles.active=worker --server.port=8082'

*# Terminal 3 - Third worker instance*

./gradlew bootRun --args='--spring.profiles.active=worker --server.port=8083'

*# Terminal 4 - Master instance (API endpoint)*

./gradlew bootRun --args='--spring.profiles.active=master --server.port=8080'

This approach allows you to test the complete master-worker architecture locally, demonstrating how job requests are distributed across multiple worker instances via Kafka.

**4. Monitoring and Maintenance**

* Health checks for master and workers.
* Kafka consumer lag monitoring.
* Automated scaling based on queue depth.
* Comprehensive logging for job tracking.

**6. Conclusion**

This architecture transforms an underutilized cluster into a distributed, efficiently balanced system. It maximizes resource investment, provides robust fault tolerance, and ensures scalable performance.