Design Document: CRS Deployment Architecture

---

1. Introduction

This document provides a detailed design for deploying the Course Recommendation System (CRS), utilizing Minikube and Kubernetes for local development, and Docker for containerization. The system comprises three main components:

- Django Backend: Handles business logic and provides API endpoints.
- React Frontend: The user interface that interacts with the backend.
- PostgreSQL Database: Stores and manages the data related to courses and users.

The system will use Kubernetes for orchestration and scaling in development and production environments, ensuring the application is easily deployable and scalable. Minikube will be used for local development to simulate a cloud-native infrastructure, while Docker will allow for quick testing and containerized deployment.

---

2. System Architecture

The CRS architecture is designed to be modular, with each component running in its own containerized environment. Kubernetes will handle the orchestration, ensuring high availability, scalability, and fault tolerance.

2.1. Application Components

1. **Django Backend:**
    - The Django backend will serve as the API layer, handling user requests and interacting with the PostgreSQL database.
    - It will be deployed as a Kubernetes Deployment, ensuring that multiple instances can be run for scalability and redundancy.
    - Persistent Storage: Any necessary persistent storage (e.g., logs) will be managed by Kubernetes using Persistent Volumes.

2. **React Frontend:**
    - The React application will provide a web interface for users, which communicates with the Django API to fetch and display data.
    - It will be deployed in a separate Kubernetes Deployment and accessed via a Kubernetes Service for internal communication.
    - Ingress will be used to expose the frontend to the outside world.
3. **PostgreSQL Database:**
    - PostgreSQL will store all necessary data for the application (user information, courses, recommendations).
    - It will be deployed using a Kubernetes StatefulSet, ensuring that data persists even when the database pod is recreated.
    - Persistent Volumes will ensure that database data is stored securely and survives pod restarts.

2.2. Kubernetes Communication

- The React frontend communicates with the Django backend through REST API calls.
- Both the frontend and backend services are exposed through Kubernetes Services to handle internal communication within the cluster.

- Ingress is used to expose the frontend to external users, making it accessible via a public IP.

---

3. Database Design

3.1. Database Choice

- PostgreSQL will be used for production, providing robust performance and reliability.
- For local development or small-scale deployments, SQLite can be used as an alternative to simplify setup.

3.2. PostgreSQL Setup in Kubernetes

- PostgreSQL will be deployed in a StatefulSet to ensure persistence across pod restarts.
- Persistent Volumes will be used to store the database data.
- Kubernetes Secrets will manage database credentials securely.

3.3. Data Models

- User Model: Contains user details like name, email, and preferences.
- Course Model: Stores information about available courses (course names, descriptions, etc.).
- Recommendations Model: Stores the generated course recommendations for each user.

---

4. Deployment Process

4.1. Local Development Setup with Minikube

1. **Minikube Setup:**

    - Install Minikube and configure kubectl to point to the local Minikube cluster.
    - Start Minikube with `minikube start` to create a local Kubernetes cluster.
    - Use `kubectl apply -f` to deploy the application to the cluster, using YAML files for Kubernetes resources like Deployments, Services, and Ingress.
2. **Docker Setup:**
    - Use Docker for containerizing each component (Django, React, PostgreSQL).
    - Build Docker images for each service and run them locally for testing, without the need for Kubernetes.

4.2. Kubernetes Resources

- Deployments for the Django and React services.
- StatefulSet for PostgreSQL, ensuring data persistence.
- Services to allow communication between the frontend, backend, and database.
- Ingress to expose the frontend to external traffic.
- ConfigMaps and Secrets for application configuration and secure credential management.

4.3. Scaling

- Horizontal Scaling: Kubernetes can automatically scale the number of pods for the backend and frontend services based on traffic.
- Vertical Scaling: The amount of CPU and memory resources can be adjusted dynamically for each pod to meet the demands of the application.

---

5. Security

5.1. Secrets Management

- Kubernetes Secrets will be used to store sensitive information like database credentials and API keys.
- RBAC (Role-Based Access Control) will enforce secure access to Kubernetes resources, limiting permissions based on the roles assigned to users.

5.2. Database Security

- Only trusted services and pods will have access to the PostgreSQL database. Access will be controlled through Kubernetes Network Policies and Secrets.

---

6. High Availability and Fault Tolerance

6.1. Fault Tolerance

- Multiple Pod Replicas: The Django backend and React frontend services will be replicated to ensure that if one pod fails, Kubernetes can automatically spin up a new one.
- StatefulSet for PostgreSQL ensures that data is not lost if the database pod is restarted.

6.2. Load Balancing

- Kubernetes Services will load balance traffic to the backend and frontend pods, ensuring even distribution of requests.
- Ingress Controller will handle external HTTP requests, forwarding them to the appropriate services within the Kubernetes cluster.

---

7. Continuous Integration and Deployment (CI/CD)

Testing

- Unit Tests: Django unit tests will ensure that the backend logic works correctly.
- End-to-End Tests: The React frontend will undergo integration tests to ensure that it functions as expected with the Django API.

The Course Recommendation System (CRS) is designed to be a cloud-native, scalable, and highly available system. By using Kubernetes for orchestration, Docker for containerization, and Minikube for local development, the system is flexible and easy to deploy in various environments. This architecture ensures that the CRS can scale as needed while providing a robust and secure platform for recommending courses to users.