Deployment Document

Overview

The deployment of the Book Management System involves:

1. **Infrastructure setup** on AWS (or any other cloud provider).
2. **Containerization** with Docker.
3. **CI/CD Workflow** using GitHub Actions.

The deployed system will use FastAPI as the backend framework and PostgreSQL as the database, with an integrated Gemini generative AI model to handle text summarization.

Section 1: Prerequisites

1. **Cloud Provider Account**: AWS account set up with access to:
    - Amazon RDS for PostgreSQL.
    - Amazon ECS (Elastic Container Service) for container orchestration.
    - Amazon ECR (Elastic Container Registry) for container image storage.
    - Amazon S3 for any file storage needs.
    - AWS Secrets Manager (for securely managing environment variables).
2. **Docker**: Installed on the local machine for containerizing the application.
3. **GitHub Repository**: Repository to host the code with GitHub Actions enabled.

Section 2: Infrastructure Setup on AWS

Step 1: Set Up Amazon RDS for PostgreSQL

1. **Create a new PostgreSQL instance**:
    - Go to Amazon RDS → Create Database → Select PostgreSQL.
    - Choose an instance type (e.g., db.t3.micro for small projects).
    - Set storage (e.g., 20 GB).
    - Enable Publicly accessible if needed, or use a VPC for private networking.
    - Set up credentials for PostgreSQL (username and password).
2. **Configure Security**:
    - Set up security groups to allow inbound traffic on port 5432 from the ECS instances or VPC.

- Configure RDS to accept connections only from the EC2 or ECS network.

3. **Note the Endpoint**: Copy the RDS endpoint to include in the .env file for database configuration.

Step 2: Set Up Amazon ECS Cluster

1. **Create an ECS Cluster**:
   - Go to Amazon ECS → Create Cluster.
   - Choose the "Networking only" cluster option (for Fargate).
   - Name the cluster (e.g., book-management-cluster).
2. **Create an ECR Repository** for Docker images:
   - Go to Amazon ECR → Create Repository.
   - Name the repository (e.g., book-management-system).
   - Note the repository URL for pushing Docker images.
3. **Create an ECS Task Definition**:
   - Go to ECS → Task Definitions → Create new Task Definition.
   - Choose Fargate as the launch type.
   - Define the container details:
     - Set the container name (e.g., book-management-container).
     - Image: Use the ECR repository URL (updated via GitHub Actions).
     - Port Mappings: Expose port 80.
   - Define environment variables, including database URL and Gemini API key.
   - Set memory and CPU limits.
4. **Create an ECS Service**:
   - Go to Services → Create Service.
   - Select the cluster, task definition, and set desired number of tasks.
   - Configure networking: Set the VPC and subnets to match RDS setup, ensuring connectivity between RDS and ECS.
5. **Set Up Load Balancer (Optional)**:
   - Create an Application Load Balancer (ALB) in the ECS service for better scaling and fault tolerance.
   - Configure ALB to forward requests to ECS tasks.

Step 3: Set Up AWS Secrets Manager

- Store sensitive data (e.g., GEMINI_API_KEY and DATABASE_URL) in AWS Secrets Manager.
- Use IAM roles to allow ECS tasks to retrieve these secrets.

Section 3: Application Configuration

Step 1: Update Environment Variables

1. **.env file**:
    - Store the following variables in your local .env file for testing and in AWS Secrets Manager for production:
        - GEMINI_API_KEY: Gemini API key.
        - DATABASE_URL: PostgreSQL connection string.
    - Configure FastAPI to read these secrets during deployment.

Step 2: Dockerize the Application

1. **Create Dockerfile** in the root directory of the application:

```
# Start from the official Python image
FROM python:3.9-slim

# Set working directory
WORKDIR /app

# Copy requirements.txt and install dependencies
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copy the application code
COPY . .

# Expose the port FastAPI will run on
EXPOSE 80
```

```
# Run the FastAPI app with Uvicorn
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]
```

2. **Build Docker Image**:

```
docker build -t book-management-system .
```

3. **Push to ECR**:
    1. Authenticate Docker with ECR.
    2. Tag the image with the ECR repository URI.
    3. Push the Docker image to ECR.

Section 4: CI/CD Pipeline Using GitHub Actions

Step 1: Create GitHub Actions Workflow

1. **File Structure**: Create the workflow file at .github/workflows/deploy.yml.
2. **Sample Workflow File** (deploy.yml):

```
name: Deploy to AWS

on:
  push:
    branches:
      - main

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Code
```

```
      uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.9'

    - name: Install Dependencies
      run: |
        pip install -r requirements.txt

    - name: Login to ECR
      env:
        AWS_REGION: 'us-west-2'
      run: |
        aws ecr get-login-password --region $AWS_REGION | docker login --username AWS --password-stdin <ecr-repo-url>

    - name: Build and Push Docker Image
      env:
        ECR_REPOSITORY: '<ecr-repo-url>'
      run: |
        docker build -t $ECR_REPOSITORY:latest .
        docker tag $ECR_REPOSITORY:latest $ECR_REPOSITORY:latest
        docker push $ECR_REPOSITORY:latest

    - name: Deploy to ECS
      uses: aws-actions/amazon-ecs-deploy-task-definition@v1
      env:
        AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
        AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

```
        AWS_REGION: 'us-west-2'
      with:
        task-definition: 'ecs-task-def.json'
        service: 'book-management-service'
        cluster: 'book-management-cluster'
        wait-for-service-stability: true
```

3. **GitHub Secrets**:
    1. Store sensitive information (like AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY) in GitHub Secrets.
    2. For accessing GEMINI_API_KEY and DATABASE_URL, reference AWS Secrets Manager in ECS.

Step 2: Configure ECS Task Definition for GitHub Actions

1. **Generate ECS Task Definition JSON**:
    - Create an ecs-task-def.json file:

```
{
  "family": "book-management-task",
  "containerDefinitions": [
    {
      "name": "book-management-container",
      "image": "<ecr-repo-url>:latest",
      "memory": 512,
      "cpu": 256,
      "essential": true,
      "environment": [
        {
          "name": "DATABASE_URL",
          "value": "<database-url>"
        },
        {
```

```
        "name": "GEMINI_API_KEY",
        "valueFrom": "<secret-manager-arn>"
      }
    ],
    "portMappings": [
      {
        "containerPort": 80,
        "hostPort": 80
      }
    ]
  }
 ]
}
```

Step 3: Test Deployment

1. **Push Changes to GitHub**:
    - Push code to the main branch, triggering the GitHub Actions workflow.
    - Verify that GitHub Actions builds and pushes the Docker image to ECR and deploys it to ECS.
2. **Verify ECS Service**:
    - Go to the ECS dashboard, check service logs, and confirm the application is running.
    - Test endpoints to ensure they respond as expected.

Section 5: Monitoring and Scaling

1. **AWS CloudWatch**:
    - Configure ECS to log to CloudWatch for error monitoring.
    - Set up alerts for CPU, memory usage, and application errors.
2. **Auto-Scaling**:
    - Set up auto-scaling rules based on CPU and memory metrics to handle traffic spikes.
3. **Load Testing**:
    - Use tools like Locust or k6 to load test endpoints and monitor performance.