EC5207: DEVOPS ENGINEERING

CI-CD DIAGRAM

NAME	:	P A N D I G A M A Y . C
REG. NO	:	EG/2022/5232
SEMESTER	:	05
DATE	:	22/01/2026

Table of Contents

Part 1: CICD Design Diagram

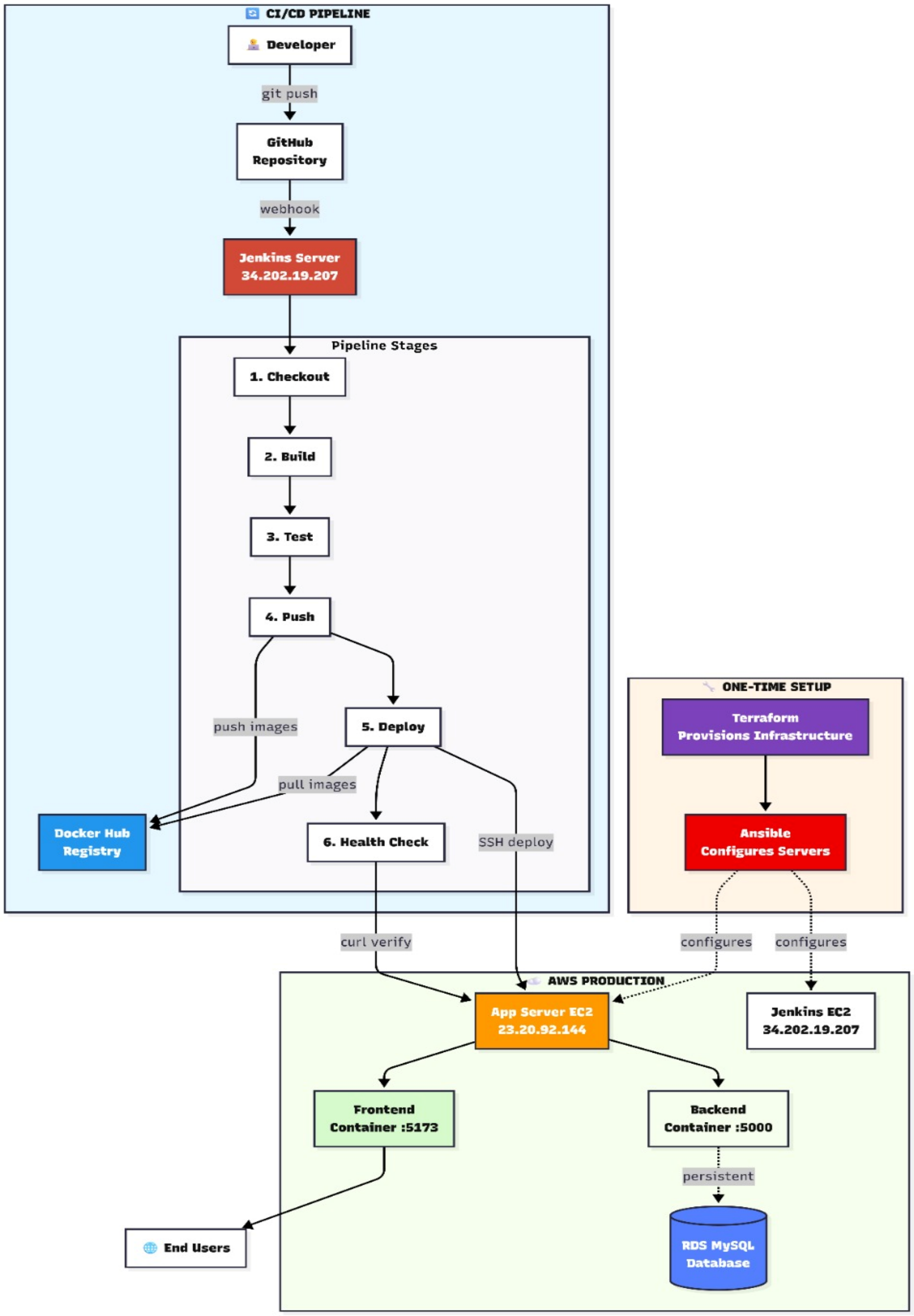

Figure 1: CICD Design Diagram

Overview

The CI/CD pipeline is a fully automated deployment workflow designed to remove manual intervention from code commit to production release. When developers push changes to the main branch on GitHub, a webhook immediately triggers Jenkins at http://34.202.19.207:8080/github-webhook/. This event-driven setup ensures that every code change is automatically picked up and processed without delays, providing fast and reliable feedback to the development team.

Once triggered, Jenkins executes a structured sequence of six stages. It first checks out the latest source code from GitHub, then builds Docker images for both the frontend and backend in parallel using multi-stage builds. After the build, the pipeline runs automated checks, including backend unit tests with Mocha, frontend linting with ESLint, and basic smoke tests to verify application stability. If all tests pass, Jenkins authenticates with Docker Hub and pushes the newly built images as yrcd27/travel-bucket-list-frontend:latest and yrcd27/travel-bucket-list-backend:latest.

In the deployment phase, Jenkins securely connects to the production EC2 instance at 23.20.92.144 via SSH, pulls the latest Docker images, and restarts the services using Docker Compose. The pipeline then performs health checks by sending curl requests to both frontend and backend endpoints to confirm successful deployment. If any stage fails, the process stops immediately, keeping the previous stable version running and ensuring zero downtime. This approach guarantees consistent deployments, supports multiple releases per day, and maintains high system reliability.

Part 2: Automation Approach

1. Briefly describe the DevOps tools you are using for the application deployment with their relevant version and purpose of using those tools

1. Git & GitHub (Version Control)

Version: Git 2.47.1
Purpose: Provides distributed version control for source code, infrastructure code, and configuration files. GitHub serves as the centralized repository and triggers automated CI/CD pipeline through webhooks on code commits.

2. Jenkins (CI/CD Server)

Version: Jenkins 2.x LTS
Purpose: Orchestrates the entire CI/CD pipeline by automating build, test, and deployment processes. Manages secure credentials for Docker Hub and EC2 access, integrates with Docker for containerization, and executes multi-stage pipelines triggered by GitHub webhooks.

3. Terraform (Infrastructure as Code)

Version: Terraform 1.13.4 with AWS Provider ~> 5.0
Purpose: Automates provisioning of AWS infrastructure including VPC, EC2 instances, RDS database, security groups, and networking components. Enables version-controlled, reproducible infrastructure deployments with easy rollback capabilities.

4. Ansible (Configuration Management)

Version: Ansible 2.16.3
Purpose: Automates configuration of provisioned EC2 instances by installing Docker, deploying application configurations, managing environment variables, and ensuring idempotent server state across deployments.

5. Docker (Containerization)

Version: Docker Engine 28.3.3
Purpose: Packages applications with all dependencies into isolated containers, ensuring consistency across development and production environments. Enables easy deployment, scaling, and version management of application components.

6. Docker Compose (Container Orchestration)

Version: Docker Compose v2.39.1
Purpose: Orchestrates multi-container deployments by defining and managing both frontend and backend containers, their networking, and dependencies through a single declarative configuration file.

7. Docker Hub (Container Registry)

Purpose: Serves as the centralized registry for storing versioned Docker images (yrcd27/travel-bucket-list-frontend:latest and backend:latest), enabling consistent image distribution to deployment servers.

8. Amazon Web Services (Cloud Infrastructure)

Services Used: EC2 (Elastic Compute Cloud), RDS (Relational Database Service), VPC (Virtual Private Cloud)
Purpose: Provides scalable cloud infrastructure for hosting application servers, Jenkins CI/CD server, and managed MySQL database with built-in security, backup, and networking capabilities.

2. Briefly describe other Application tools and dependencies with their relevant versions

Frontend Application Stack:

React- Modern JavaScript framework for building interactive user interfaces with component-based architecture
Vite- Fast build tool providing instant hot module replacement for rapid development
React Router DOM- Client-side routing library for single-page application navigation
Axios- Promise-based HTTP client for making API requests to backend
ESLint- Code quality tool for identifying and fixing JavaScript errors and enforcing coding standards
nginx: Alpine-latest - Lightweight web server serving production React build files with optimized configuration

Backend Application Stack:

Node.js- JavaScript runtime environment for executing server-side code
Express.js- Minimal web framework for building RESTful APIs and handling HTTP requests
MySQL2- MySQL client library for database connectivity with promise-based API
bcryptjs- Password hashing library for secure credential storage
jsonwebtoken - JWT implementation for secure user authentication and authorization
CORS: - Middleware enabling secure cross-origin resource sharing between frontend and backend
dotenv- Environment variable management for secure configuration handling
Mocha- Testing framework for writing and executing backend unit tests
nodemon- Development utility for automatic server restart on code changes
Database:

MySQL: Version 8.0 on AWS RDS (db.t3.micro instance)

Purpose: Relational database management system hosting the travelbucket database with users and destinations tables, providing reliable data persistence with automated backups
Operating System:

Ubuntu Server: Version 22.04 LTS (ARM64)

Purpose: Stable Linux distribution optimized for ARM-based Graviton2 processors, providing long-term support and wide package availability

3. Briefly describe how your application deployment has been automated. You can include your pipeline stages for comprehensive explanation.

The application deployment is fully automated through a Jenkins pipeline that executes six sequential stages without manual intervention. When developers push code to GitHub's main branch, a webhook triggers Jenkins at http://34.202.19.207:8080/github-webhook/, initiating the automated workflow.

Stage 1: Checkout
Jenkins receives the webhook notification and automatically fetches the latest code from the GitHub repository's main branch using the SCM plugin, making the codebase available in the Jenkins workspace for subsequent stages.

Stage 2: Build Docker Images (Parallel Execution)
The frontend undergoes a multi-stage Docker build starting with node:20-alpine as the builder, installing dependencies, building the production bundle with Vite, then copying artifacts to an nginx:alpine final image. Simultaneously, the backend is built using node:20-alpine with production-only dependencies installed via npm ci. Both images are tagged as yrcd27/travel-bucket-list-frontend:latest and backend:latest.

Stage 3: Test (Parallel Execution)
Three automated test suites run concurrently: backend unit tests execute using the Mocha framework to validate API endpoints and business logic; frontend code undergoes ESLint analysis to ensure React best practices and identify potential bugs; smoke tests verify Docker images built successfully. The pipeline stops immediately if any test fails, preventing faulty code from reaching production.

Stage 4: Push to Docker Hub
Jenkins retrieves stored Docker Hub credentials securely from its credential manager and authenticates using docker login. Both frontend and backend images are pushed to the Docker Hub registry (yrcd27/travel-bucket-list-frontend:latest and backend:latest), making them available for deployment while ensuring credentials never appear in logs.

Stage 5: Deploy to Production EC2
Jenkins connects to the App Server at 23.20.92.144 via SSH using the stored ec2-ssh-key credential. Remote commands are executed to pull the latest Docker images from Docker Hub, gracefully stop existing containers with docker compose down, and start updated containers using docker compose up -d --force-recreate. Container status is verified with docker ps to ensure successful deployment.

Stage 6: Health Check
After allowing containers to fully initialize, automated health checks execute curl requests to verify both the frontend endpoint (http://23.20.92.144:5173) and backend API health endpoint

(http://23.20.92.144:5000/api/health) return successful HTTP 200 responses. If health checks fail, the pipeline marks the deployment as failed and the previous version continues running, ensuring zero downtime. Successful verification confirms the application is live and operational.

Post-Build Actions:

Regardless of pipeline outcome, local Docker images are cleaned up from the Jenkins server to free disk space. On success, the build is marked with green status and the application URL is displayed. On failure, detailed error logs are captured and the build is marked red for developer investigation.