# SEMINAR DOCUMENTATION

## 1. Introduction & Background

- Software deployment has evolved from **manual, error-prone processes** to **automated, high-velocity pipelines** that define modern delivery — reflecting not just technical change, but a **cultural shift** in how teams build and release software.

### 1.1. The Evolution: From Manual Friction to Automated Flow

- In the early days, deploying web apps was **manual, fragile, and error-prone** — developers often used tools like **FileZilla** to upload files directly to live servers.
- **Key Pain Points of Manual Deployment**
  - **Human Error:** Missing files, outdated versions, or misconfigurations caused frequent failures.
  - **No Rollback:** No version control; failed deployments required manual fixes.
  - **Environment Mismatch:** Inconsistent local and production setups.
  - **Deployment Fear:** High risk made teams deploy rarely, in risky batches.
  - **Security Bottlenecks:** Manual reviews slowed releases.
- **The Turning Point**
  - The adoption of **version control systems** such as Git began to solve reproducibility issues.
  - The rise of **Platform-as-a-Service (PaaS)** platforms like Heroku revolutionized deployment simplicity.

  ⇒ This innovation paved the way for **CI/CD** — emphasizing small, rapid, automated releases.

### 1.2. The Primacy of Automation in a DevOps Culture

- **DevOps** unites development and operations through shared responsibility and automation.

- **CI/CD** is the **technical engine** that enables **DevOps** — automating code integration, testing, and deployment.
- **Benefits of Automation**
  - **Speed & Efficiency:** Enables fast feedback and frequent releases.
  - **Reduced Risk:** Ensures consistent, repeatable deployments.
  - **Improved Quality:** Automated tests quickly detect and isolate issues.
  - **Continuous Feedback:** Supports rapid, customer-focused iteration.
- **Stages of CI/CD Automation**
  - **Continuous Integration (CI):** Code merges trigger automated builds and tests.
  - **Continuous Delivery (CD):** Software stays deployable; staging is automatic.
  - **Continuous Deployment (CD):** Every passing change ships to production**;** relies heavily on test automation.

## 1.3. Architecting the Modern, Decoupled Pipeline

- Modern CI/CD pipelines are designed to be decoupled, portable, and modular, allowing each stage (build, test, deployment) to use the best-suited tool without vendor lock-in.
- **Chosen Stack Overview**
  - **Frontend** code is deployed to **Vercel**, providing fast, global delivery.
  - **Backend** services are containerized with **Docker** and deployed to **Render**, ensuring consistency and scalability.
  - **GitHub Actions** orchestrates builds, tests, and deployments automatically whenever code changes are pushed.

| Pain Point (Manual Era) | Modern CI/CD Solution |
|---|---|
| **High Risk of Human Error** | An automated, repeatable pipeline runs the same process, error-free, every time. |
| **No Reproducibility or Rollback** | Immutable, version-controlled Docker images and Git-based history allow instant, reliable rollbacks. |
| **Deployment Fear / Infrequency** | Automation enables a “fail fast" culture, making small, frequent deployments safe and routine. |

| Pain Point (Manual Era) | Modern CI/CD Solution |
| --- | --- |
| **Environment Mismatch** | Docker containers create a uniform, consistent environment, isolating software from its environment. |
| **Security & Compliance Bottlenecks** | Security is shifted left — automated vulnerability scanning and compliance checks are integrated directly into the pipeline. |
| **Opaque Configuration** | Infrastructure as Code (IaC), such as a `render.yaml` file, makes all environment configuration declarative and version-controlled. |

# 2. Objectives

- **Automate the Full Pipeline:** Build a complete, automated **build–test–deploy** workflow using **GitHub Actions**, turning CI/CD concepts into hands-on practice.
- **Master Containerized Deployment:** Package applications with all dependencies using an optimized **multi-stage Dockerfile**, creating a secure, minimal, production-ready backend image.
- **Implement Decoupled Hosting:** Deploy a **React frontend to Vercel** and a **Node.js backend to Render**, leveraging a decoupled architecture that avoids vendor lock-in.
- **Showcase CI/CD Best Practices:** Apply real-world DevOps techniques including **monorepo path filtering**, **dependency caching**, and **secure secret management** to enhance reliability and maintainability.

# 3. Tools and Technologies

## 1. GitHub Repository

- **Description:** A centralized, cloud-based storage location for your project's code and files, managed by the Git version control system.
- **Role:** The **single source of truth** for your entire application. It tracks every change, manages different versions (branches), and is the central hub for team collaboration.
- **Usage:** Developers `git push` new code to the repository and `git pull` updates from it. It's where all the automated workflows (like GitHub Actions) are triggered from.

## 2. GitHub Actions

- **Description:** A powerful automation and CI/CD platform built directly into GitHub. It allows you to define custom workflows that run in response to events in your repository (like a `push` or `pull request`).
- **Role:** The **automation engine** or "conductor" of your development pipeline. It takes your code, runs commands on it (like testing, building), and then tells other services what to do.
- **Usage:** You create YAML files in your repository's `.github/workflows` directory. These files define a series of "jobs" and "steps," such as `npm install`, `npm test`, `docker build`, and `deploy`.

## 3. Docker

- **Description:** A platform that enables you to "containerize" applications. A container is a lightweight, standalone package that includes application's code, runtime, and all its dependencies (libraries, settings, etc.).
- **Role:** The **portable environment** for your backend. It solves the "it works on my machine" (incongruency in environments and OSes) problem by ensuring your backend API runs in the *exact same* environment in development, testing, and production.
- **Usage:** You create a `Dockerfile` (a text file with instructions) in your repository. **GitHub Actions** uses this file to build a "Docker image" (a template) and then pushes that image to a container registry. **Render** then pulls and runs this image.

## 4. Vercel

- **Description:** A cloud PaaS (Platform-as-a-Service) optimized for deploying static websites and frontend frameworks (like React, Next.js, Vue, Svelte).
- **Role:** The **hosting platform for your frontend** (the user interface). It's known for its high performance, global CDN (Content Delivery Network), and seamless integration with GitHub.
- **Usage:** You connect your GitHub repository's frontend directory to Vercel. When you push new code, Vercel *automatically* detects it, builds your frontend application, and deploys it, often providing a unique preview URL for every push.

## 5. Render

- **Description:** A flexible cloud platform (PaaS) designed to host a wide variety of services, including backend APIs, databases, and containerized applications.
- **Role:** The **hosting platform for your backend** (the API). It's where your **Docker** container (containing your backend code) lives, runs, and responds to requests from your Vercel-hosted frontend.
- **Usage:** You configure Render to watch your Docker container registry (or your GitHub repo directly). When a new image is pushed by **GitHub Actions**, Render automatically pulls it and deploys it as a "new version" of your backend service with zero downtime.

Here is a step-by-step example of how these tools work together in a modern web app pipeline (e.g., a React frontend on Vercel and a Node.js/Express backend on Render).

1. **Code Push:** A developer finishes a new feature (e.g., a new API endpoint and a new UI button) and pushes the code for both the frontend and backend to the **GitHub Repository**.
2. **Workflow Trigger:** This `push` event automatically triggers your workflow in **GitHub Actions**.
3. **CI (Continuous Integration) Phase:**
    - **GitHub Actions** spins up a temporary virtual machine.
    - It checks out the new code.
    - It runs automated tests for both the frontend and backend (e.g., `npm test`). If any tests fail, the pipeline stops, and the developer is notified. This prevents broken code from being deployed.
4. **CD (Continuous Deployment) Phase (if tests pass):** The workflow splits into two parallel jobs:
    - **Frontend Deployment (to Vercel):**
        - **Vercel** has a native integration with the **GitHub Repository**. It detects the push to the main branch *at the same time* as GitHub Actions.
        - It automatically pulls the frontend code, builds the static application, and deploys it to its global CDN.

    - Within seconds, the new UI is live for users.
- **Backend Deployment (to Render):**
    - **GitHub Actions** (continuing its job) uses the `Dockerfile` in the repository to build a new **Docker** image of the backend API.
    - After the image is built, **GitHub Actions** pushes this new, versioned image to a container registry (like Docker Hub or GitHub Container Registry).
    - **Render** (which is configured to "watch" that container registry) detects the new image.
    - Render automatically pulls the new container image and deploys it, replacing the old running container with the new one. Your backend API is now updated.

# 4. System Architecture Overview

## 4.1 Overall Concept

In our system, we apply a **modern DevOps pipeline** that connects the entire process — from source code changes to automatic deployment — across both backend and frontend services.

The idea is simple: **every time a developer pushes new code to the GitHub repository, the whole build, test, and deployment process happens automatically**, ensuring fast and reliable delivery.

This pipeline relies on three main technologies working together:

- **GitHub Actions** for automation and CI/CD,
- **Docker** for containerization and consistent runtime,
- **Render and Vercel** as cloud hosting platforms for backend and frontend.

## 4.2 System Workflow Diagram

```
Developer Push → GitHub Actions CI/CD → Docker Build →
Render (Backend API) + Vercel (Frontend)
```

## 4.3 Workflow Stages Explained

### 4.3.1 Source Control & CI Trigger

- All source code (both frontend and backend) is stored in a **GitHub repository**.
- The repository contains a `.github/workflows` directory, where the **GitHub Actions workflow file** (`ci-cd.yml`) is defined.
- Whenever a developer pushes code to the `main` branch, GitHub Actions automatically triggers the workflow — this marks the start of the **CI/CD pipeline**.

### 4.3.2 Build & Test Stage

- In this stage, GitHub Actions pulls the latest code and installs all dependencies for both the **frontend (React)** and the **backend (Node.js or ASP.NET)**.
- Automated tests (if available) are run to check code integrity.
- If everything passes, the workflow proceeds to the next stage.

### 4.3.3 Dockerization

- Both backend and frontend are containerized using **Docker**.
- Each service has its own `Dockerfile` — describing how the image is built, what dependencies are installed, and what command runs the app.
- The pipeline can use **multi-stage builds** to reduce image size and speed up deployments.

Example:

- The backend's Dockerfile includes building and exposing the API.
- The frontend's Dockerfile builds the static React files ready for production.

### 4.3.4 Deployment

Once the Docker images are successfully built, GitHub Actions deploys them automatically to the respective platforms:

- **Backend → Render:**

  The Docker image or source code is pushed to Render, where it is hosted as a live backend API.

  Environment variables (like database URLs or API keys) are injected securely via Render's settings.

- **Frontend → Vercel:**

  The frontend build output is automatically deployed to Vercel.

  Vercel detects the React framework and configures optimal settings for production hosting.

  Once completed, a live production URL is generated.

### 4.4 Data & Process Flow Summary

| Step | Component | Description |
|---|---|---|
| 1 | Developer | Pushes code to GitHub |
| 2 | GitHub Actions | Detects push → runs workflow |
| 3 | Docker | Builds container images |
| 4 | Render | Deploys backend API |
| 5 | Vercel | Deploys frontend website |

### 4.5 Key Benefits of This Architecture

- **Fully automated deployment:** No manual steps required.
- **Consistency:** Same environment from development to production using Docker.
- **Speed:** Every commit can trigger a quick build-test-deploy cycle.
- **Scalability:** Easily extendable to more services or environments.
- **Reliability:** Each deployment is versioned and traceable through GitHub Actions logs.

## 5. Extracted CI/CD Configuration

**Architecture Overview — Triggers & Monorepo Logic**

- **What:** The GitHub Actions workflow triggers on `push` and `pull_request` targeting `main` and `dev` branches.
- **Why:** Standard CI triggers to validate code on PRs and on branch updates.
- **Notes:** There are no path filters in the workflow (no `paths:` clauses), so the workflow runs for every push/PR to those branches rather than being limited by monorepo subpaths.

```
name: CI / CD

on:
    push:
        branches: [main, dev]
    pull_request:
        branches: [main, dev]
```

- What the block does: declares workflow triggers.
- Why it exists: ensures CI runs on changes to primary branches and PRs against them.
- How it contributes: starts the CI pipeline (lint, test, deploy jobs).
- Dependencies/assumptions: Assumes team uses `main` / `dev` branches; no monorepo path filtering means all pushes/prs will run the workflow.

**CI Stage — Linting & Tests (GitHub Actions jobs)**

The workflow defines server lint and server test jobs. The client lint/job is present but commented out.

```
server-lint:
    name: Lint (server)
    runs-on: ubuntu-latest
    steps:
        - name: Checkout
          uses: actions/checkout@v4
        - name: Setup Node.js
          uses: actions/setup-node@v4
          with:
              node-version: 21
        - name: Install server dependencies
          run: |
              cd server
              npm ci
        - name: Install ESLint for CI
          run: |
              cd server
              npm i -D eslint @typescript-eslint/parser @typescript-esli
```

```
nt/eslint-plugin
            - name: Run server lint
              run: |
                  cd server
                  npx eslint src --ext .ts || true

server-test:
    name: Server Unit Tests
    runs-on: ubuntu-latest
    steps:
        - name: Checkout
          uses: actions/checkout@v4
        - name: Setup Node.js
          uses: actions/setup-node@v4
          with:
              node-version: 22
        - name: Install server dependencies
          run: |
              cd server
              npm ci --include=dev
        - name: Run server tests
          run: |
              cd server
              npm test
```

- What the blocks do: run lint and unit tests for the `server` package.
- Why they exist: to catch style and functional regressions before deployment.
- How they contribute: they gate the `server-deploy` job (see `needs:`). The `|| true` on the lint step prevents lint failures from failing the job — likely intentional to avoid blocking.
- Dependencies/assumptions: relies on Node.js (v21/v22), `npm ci`, project scripts (`npm test`), and `eslint` being compatible with the server code.
- Notable omissions: no `actions/cache` usage for `node_modules` or package caching — dependency install runs every CI execution.

**CD Stage — Render & Vercel Deploy**

The workflow triggers a Render deployment by invoking a Render deploy hook. The client deploy step to Vercel is included but commented out.

```
server-deploy:
    name: Trigger Render Deploy for Server
    needs: [server-lint, server-test]
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4
      - name: Trigger Render deployment (via Deploy Hook)
        env:
          RENDER_DEPLOY_HOOK: ${{ secrets.RENDER_DEPLOY_H
OOK }}
        run: |
          if [ -z "${RENDER_DEPLOY_HOOK}" ]; then
            echo "RENDER_DEPLOY_HOOK is not set. Skipping Ren
der deploy.";
            exit 1;
          fi
          curl -fsS -X POST "${RENDER_DEPLOY_HOOK}" || true

# client-deploy (commented): Trigger Vercel using `VERCEL_TOKEN` with `
npx vercel --prod`
```

- What the block does: posts to a Render deploy hook to trigger Render to build/deploy the service; client deployment via Vercel is present but disabled in the workflow.
- Why it exists: keeps CI focused on tests and uses each host's native deploy mechanisms (Render's hook) rather than building/pushing container images from Actions.
- How it contributes: automates remote deployment after successful CI. Because the job calls a webhook, the actual build/publish is performed by Render (not Actions).
- Dependencies/assumptions: requires a `RENDER_DEPLOY_HOOK` secret configured in GitHub; assumes Render is connected to the repository or to a container registry that will be updated by Render.

- Secrets usage: `secrets.RENDER_DEPLOY_HOOK` (required). Commented client-deploy references `secrets.VERCEL_TOKEN`.

**Docker — Multi-stage backend build (server/Dockerfile)**

The repository contains a multi-stage `Dockerfile` for the backend. It separates a `builder` stage (install + build) from a `runner` stage (runtime image).

```
# Multi-stage Dockerfile for backend
# Builder: install deps and build TypeScript
FROM node:20-alpine AS builder

WORKDIR /app

# Copy package files and install (includes devDependencies for build)
COPY package*.json ./
RUN npm ci --silent

# Copy source and build
COPY tsconfig.json ./
COPY src ./src
RUN npm run build

# Runtime image: lightweight, only runtime files
FROM node:20-alpine AS runner

WORKDIR /app
ENV NODE_ENV=production

# Copy package.json (for metadata) and production node_modules and buil
t dist
COPY --from=builder /app/package.json ./package.json
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/dist ./dist

# Expose the port your app uses (adjust if different)
EXPOSE 3000
```

```
# Start the server
CMD ["node","dist/index.js"]
```

- What the file does: builds TypeScript in a builder stage and produces a minimal runtime image containing `node_modules` and compiled `dist`.
- Why it exists: multi-stage reduces final image size and avoids shipping dev dependencies in production.
- How it contributes: provides a production-ready container image that can be built locally, by Render, or by a CI job; suitable for container registries.
- Dependencies/assumptions: `npm run build` emits `dist/` and installs build-time dependencies; Node 20 alpine base is used; app listens on port 3000.

**docker-compose (runtime vs. developer override)**

`server/docker-compose.yml` defines the production container mapping; `docker-compose.override.yml` changes the service for local development (volumes, command, NODE_ENV). Extracts below.

```
services:
  server:
    container_name: roads-to-rome-server
    build: .
    ports:
      - "3000:3000"
    env_file:
      - .env
    environment:
      NODE_ENV: production
    restart: unless-stopped
```

```
services:
  server:
    container_name: roads-to-rome-server-dev
    command: npm run dev
    env_file:
      - .env
    environment:
      NODE_ENV: development
```

```
    ports:
      - "3000:3000"
    volumes:
      - ./:/app:delegated
      - /app/node_modules
```

- What the files do: `docker-compose.yml` expresses how to build and run the server container; the override file enables a dev workflow (mounted volume, `npm run dev`).
- Why they exist: make it easy to run the service locally (hot-reload) and to define container runtime settings for deployments or staging.
- How it contributes: local developer experience and a simple container runtime definition for environments that accept docker-compose deployments.
- Dependencies/assumptions: expects an `.env` file for environment variables; `npm run dev` is defined for developer mode.

**Vercel config (client/vercel.json)**

```
{
  "rewrites": [
    { "source": "(.*)", "destination": "/index.html" }
  ]
}
```

- What the file does: instructs Vercel to rewrite all routes to `index.html` (single-page-app routing).
- Why it exists: ensure client-side routing works for deep links when deployed to Vercel's static hosting.
- How it contributes: prevents 404s for client-side routes by serving the SPA entrypoint.
- Dependencies/assumptions: client is a single-page app (React/Vite) and expects client-side routing.

**Summary of notable CI/CD properties observed**

- Triggers: `push` and `pull_request` on `main` and `dev` branches.

- Monorepo path filters: none present — workflows run for all changes to those branches.
- Caching: no `actions/cache` usage for Node modules; each CI run installs dependencies anew.
- Docker build/push: the repository contains a multi-stage `Dockerfile`, but GitHub Actions does not build/push images to a registry in the current workflow — instead a Render deploy hook is used.
- Secrets: `secrets.RENDER_DEPLOY_HOOK` is required to trigger Render; `secrets.VERCEL_TOKEN` is referenced but the Vercel deploy job is commented out.
- Render usage: deploys are triggered via webhook (deploy hook), so Render handles the actual build/deploy step.