Project - High Level Design

on

Deploy

Hospitality App

to Kubernetes

Course Name: DEVOPS FOUNDATION

Institution Name: Medicaps University – Datagami Skill Based Course

Student Name(s) & Enrolment Number(s):

Sr no	Student Name	Enrolment Number
1	Uddhava Mishra	EN22CS303056
2	Keyura Joshi	EN22CS306024
3	Samiksha Agrawal	EN22CS306045
4	Vedant Yadav	EN22CS304064

Group Name : Group 02D11

Project Number:DO-28

Industry Mentor Name: Vaibhav

University Mentor Name: Shyam Patel

Academic Year:2026

Table of Contents

1 INTRODUCTION :

In the modern hospitality industry, digital services are no longer just support systems; they are the primary interface for guest interactions. From booking reservations to contactless check-ins, the demand for 24/7 availability is non-negotiable. Any service disruption or downtime during updates can lead to failed transactions, frustrated guests, and direct revenue loss.

To address these challenges, this project implements a cloud-native deployment strategy for the Hospitality .NET application. By moving away from traditional monolithic server hosting to a containerized architecture, we aim to achieve higher agility, scalability, and resilience.

This implementation leverages Docker for creating lightweight, portable application artifacts and Kubernetes (Minikube) for orchestration. The core focus of this project is not merely deployment, but the assurance of business continuity. We utilize Kubernetes' advanced Rolling Update capabilities to ensure that new features and patches can be deployed to production in real-time without interrupting active user sessions or causing service downtime.

This document details the end-to-end engineering process, from the Continuous Integration (CI) pipeline that builds the software, to the specific configuration of the Kubernetes cluster that hosts it on a resource-optimized AWS infrastructure.

1.1 Scope of Document:

- This document outlines the architectural design, implementation strategy, and operational procedures for deploying a containerized Hospitality .NET application onto a Kubernetes cluster.
- Specifically, this document covers:
  Infrastructure Provisioning: Configuration of a cost-effective Kubernetes environment using Minikube on a resource-constrained AWS EC2 instance.
- CI/CD Pipeline Design: A detailed breakdown of the automated build and deployment pipeline using GitHub Actions and Docker Hub.
- Zero-Downtime Implementation: The technical methodology for achieving seamless "Rolling Updates" using Readiness/Liveness probes to ensure continuous availability during deployment cycles.

1.2 Intended Audience:

This document is designed for technical and semi-technical stakeholders involved in the software delivery lifecycle:

- DevOps Engineers & System Administrators: To understand the infrastructure constraints (Swap memory configuration), pipeline workflows, and cluster management commands.
- Backend Developers: To understand how their application is containerized and the health-check requirements (Probes) needed for successful orchestration.
- Project Managers/Product Owners: To gain high-level visibility into the release process, deployment frequency capabilities, and availability guarantees (Zero Downtime).

1.3 System Overview

The proposed system utilizes a modern, cloud-native architecture designed to decouple the Continuous Integration (CI) process from the Continuous Deployment (CD) environment. This separation allows for high-performance builds without overloading the hosting infrastructure.

The architecture consists of three primary stages:

A. The Build Plane (External CI) We leverage GitHub Actions as an external build plane. Upon a code commit, this SaaS-based runner handles the resource-intensive tasks of compiling the C# .NET code, running unit tests, and building the Docker container image. This artifact is then pushed to Docker Hub, a centralized container registry.

B. The Hosting Plane (AWS + Minikube) The application resides on a single AWS EC2 (t2.micro) instance. To circumvent the hardware limitations of the Free Tier, the instance is configured with swap memory to support Minikube, a lightweight Kubernetes implementation. This cluster acts as the orchestration engine, managing the application pods.

C. The Deployment Logic (Rolling Updates) The deployment uses a Rolling Update Strategy. When a new version is detected:

1. Kubernetes spins up a new pod alongside the old ones.

2. Readiness Probes verify the new pod is fully functional.

3. Traffic is gradually shifted to the new pod.

4. Only after confirmation, the old pod is terminated.

This ensures that at no point in time is the service unavailable to the end-user.

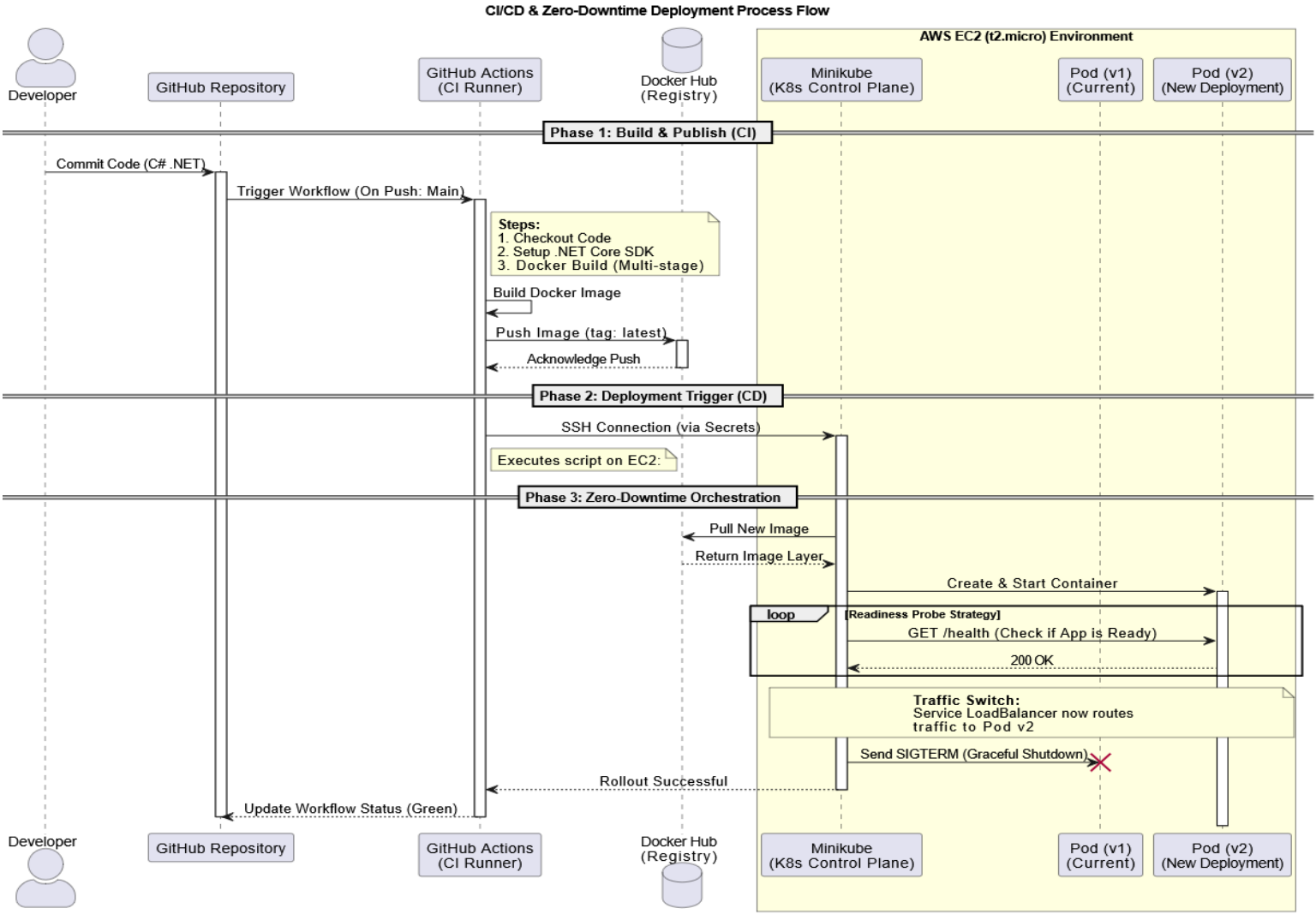
2 System Design:

2.1 Application Design

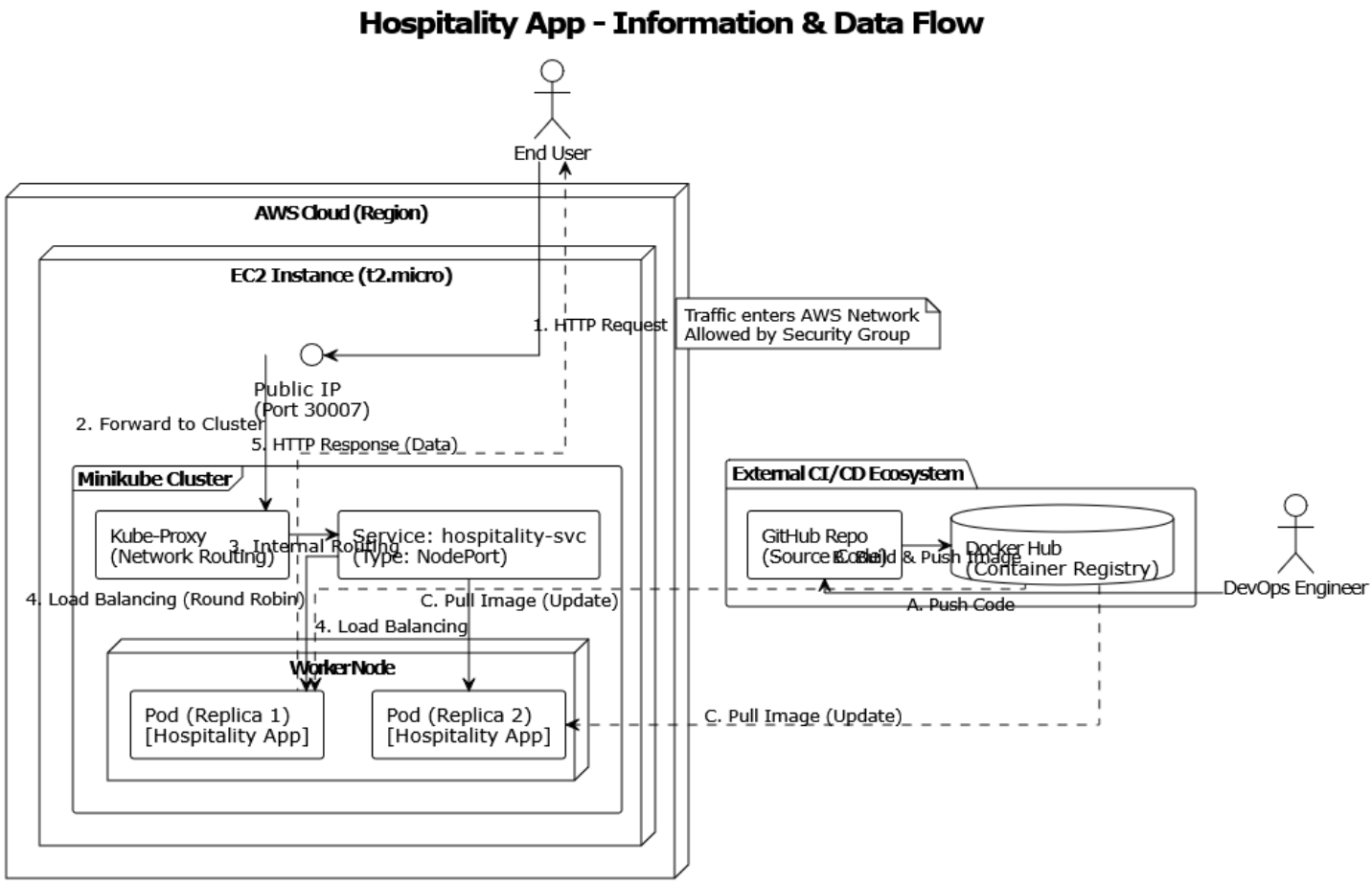
The Hospitality Application is engineered as a Cloud-Native, Container-First service. It adheres to the Twelve-Factor App methodology to ensure portability, scalability, and resilience when orchestrated by Kubernetes.
Architectural Pattern
The application utilizes a modular architecture designed for stateless execution.
Statelessness: The application does not store client session data (state) in the local process memory. All state is offloaded to external backing services (e.g., a database or distributed cache like Redis).
Significance: This stateless design is critical for the "Zero Downtime" requirement. It allows Kubernetes to destroy old pods and spin up new ones indiscriminately without data loss or interrupting active user sessions.

Technology Stack
Framework: .NET 8 (LTS) - Chosen for its cross-platform capabilities and high performance on Linux.
Web Server: Kestrel - A lightweight, cross-platform web server optimized for running inside Docker containers.
Base Image: mcr.microsoft.com/dotnet/aspnet:8.0-alpine - We utilize the Alpine Linux variant to minimize the attack surface and reduce the image size (approx. 100MB), ensuring faster deployment times on the resource-constrained EC2 instance.

2.2 Process Flow Diagram


CI/CD & Zero-Downtime Deployment Process Flow
Developer
GitHub Repository
GitHub Actions (CI Runner)
Docker Hub (Registry)
AWS EC2 (t2.micro) Environment
Minikube (K8s Control Plane)
Pod (v1) (Current)
Pod (v2) (New Deployment)
Phase 1: Build & Publish (CI)
Commit Code (C# .NET)
Trigger Workflow (On Push: Main)
Steps:
1. Checkout Code
2. Setup .NET Core SDK
3. Docker Build (Multi-stage)
Build Docker Image
Push Image (tag: latest)
Acknowledge Push
Phase 2: Deployment Trigger (CD)
SSH Connection (via Secrets)
Executes script on EC2:
Phase 3: Zero-Downtime Orchestration
Pull New Image
Return Image Layer
Create & Start Container
loop
[Readiness Probe Strategy]
GET /health (Check if App is Ready)
200 OK
Traffic Switch:
Service LoadBalancer now routes traffic to Pod v2
Send SIGTERM (Graceful Shutdown)
Rollout Successful
Update Workflow Status (Green)

2.3 Information Flow



2.4 Component Design

The system architecture is modular, composed of loosely coupled components managed by the Kubernetes control plane. Each component is designed to be autonomous, ensuring that the failure of one unit (like a Pod) does not bring down the entire system.

Infrastructure Layer Components

These components provide the foundational compute resources required to host the application.

- Host Node (AWS EC2 t2.micro)
    - Role: The physical (virtualized) server acting as the single-node cluster.
    - Operating System: Ubuntu 22.04 LTS (Chosen for stability and low overhead).
    - Swap Memory Module:
        - Function: A 2GB virtual memory file created on the disk.
        - Design Rationale: Critical for this project. Since t2.micro only has 1GB RAM, the swap file acts as an "overflow" buffer. Without this component, the Minikube cluster would crash immediately upon deployment.
- Container Runtime (Docker Engine)
    - Role: The engine responsible for running the actual application containers.
    - Configuration: Configured to use the overlay2 storage driver, which optimizes disk usage—essential for the limited storage on

the Free Tier instance.

Orchestration Layer (Kubernetes Objects)

These are the logical components defined in your YAML manifests that manage the application.

- The Deployment Controller (hospitality-deployment)
    - Role: The "Brain" of the application. It constantly monitors the state of the system to ensure the desired number of pods are running.
    - Key Design Feature: RollingUpdate Strategy.
        - maxSurge: 1: Allows the system to create one extra pod during updates.
        - maxUnavailable: 0: Ensures no existing pods are killed until the new one is ready. This is the core component that guarantees Zero Downtime.
- The Application Pods
    - Role: The atomic units of deployment. Each pod encapsulates one running instance of the .NET Hospitality Application.
    - Resource Guards:
        - Requests: 128Mi RAM (Guaranteed minimum).
        - Limits: 256Mi RAM (Maximum allowed).
        - Design Rationale: These hard limits prevent a "noisy neighbor" scenario where one memory leak could crash the entire server.
    - Health Probes:
        - Liveness Probe: Restarts the pod if the .NET process deadlocks.

- Readiness Probe: Blocks traffic until the app is fully booted.

- The Service Abstraction (hospitality-service)
    - Type: NodePort.
    - Role: Acts as the internal Load Balancer. It provides a stable networking endpoint (IP and Port) for the application.
    - Function: Even though Pods are created and destroyed frequently (during rolling updates), the Service component remains static, ensuring users never lose their connection point. It routes traffic only to "Ready" pods.

2.5 Key Design Considerations

The architecture of the Hospitality Application was driven by four primary design pillars: Resource Efficiency, Service Continuity, Security, and Automation. Given the specific constraint of deploying to a single-node, resource-limited environment (AWS t2.micro), specific trade-offs and optimizations were implemented.

Resource Optimization (The "Constraint" Strategy)
Swap Memory Implementation: The standard t2.micro instance provides only 1GB of RAM, which is insufficient for running both the Kubernetes Control Plane (Minikube) and the .NET application.
Design Decision: A 2GB Swap file was provisioned on the host disk.
Trade-off: While disk-based memory is slower than physical RAM, this trade-off was necessary to prevent Out-Of-Memory (OOM) crashes and ensure system stability during peak loads.
Lightweight Base Images: We utilized Alpine Linux for the .NET runtime image.
Impact: This reduced the container image size from ~500MB (standard Debian) to ~100MB. Smaller images consume less storage and bandwidth, resulting in faster pull times during updates.
Hard Resource Limits: Kubernetes Resource Quotas were strictly defined (limits.memory: 256Mi). This ensures that a single misbehaving pod cannot consume all available host memory and crash the critical system processes (like the Kubelet or Docker daemon).

Availability & Resilience (The "Zero Downtime" Strategy)
Rolling Update Configuration: The deployment strategy was explicitly tuned for seamless transitions.
Configuration: maxUnavailable was set to 0.

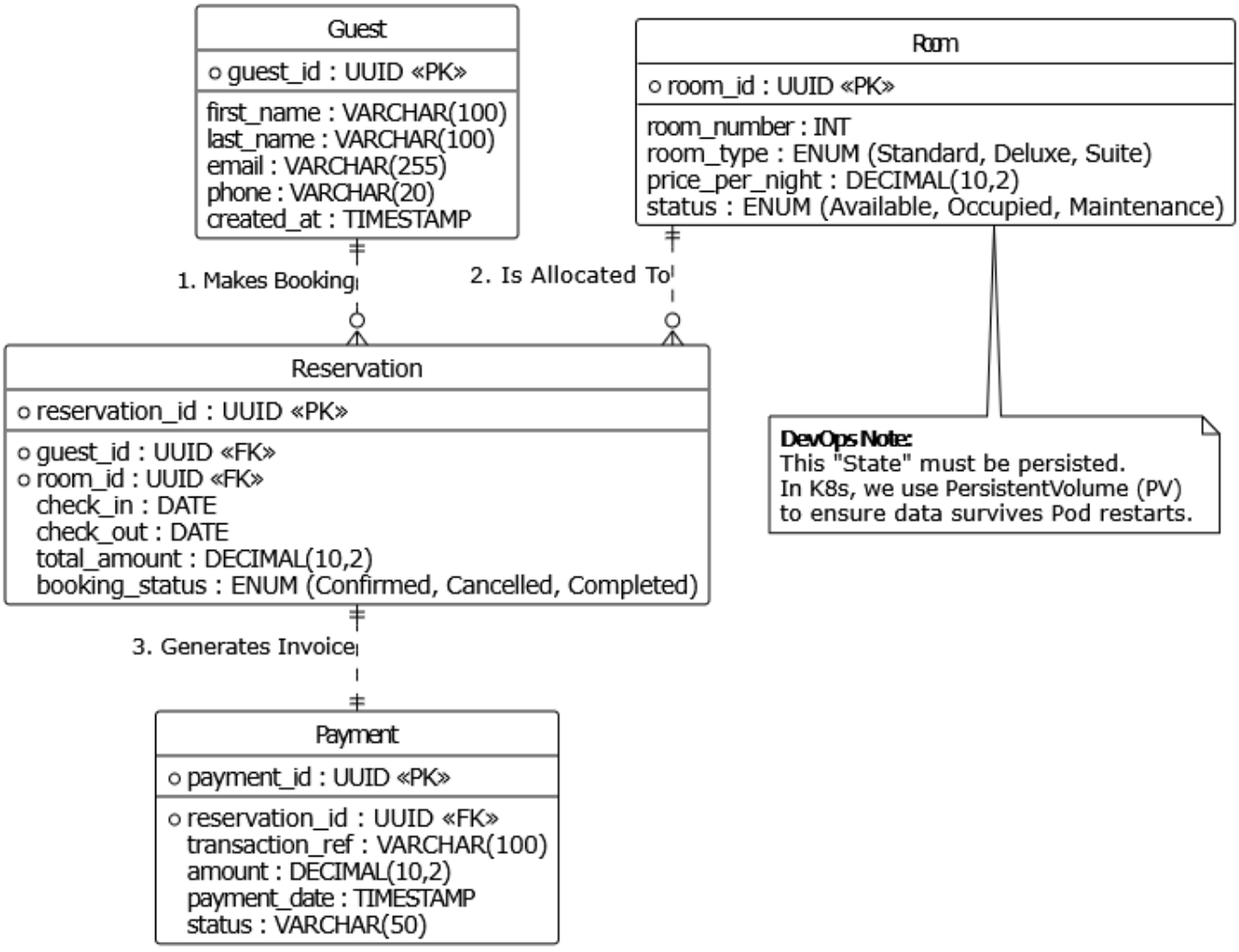
Reasoning: This guarantees that Kubernetes will never terminate an old version of the application until the new version is confirmed running and healthy. This is the technical enforcement of the "Zero Downtime" requirement.
Probes for Self-Healing:
Readiness Probes: We implemented a specific /health/ready endpoint. The Load Balancer (Service) is configured to never route traffic to a container until this probe returns a 200 OK status, preventing users from hitting "booting" applications.
Liveness Probes: If the application enters a deadlocked state, this probe detects the failure and automatically restarts the container without human intervention.

Security & Isolation
Secret Management: Sensitive information (such as database connection strings or API keys) was decoupled from the source code and the Docker image.
Implementation: We utilized Kubernetes Secrets objects. These are mounted into the pods as environment variables at runtime, ensuring credentials are never exposed in the Git repository.
Network Isolation: The application is not exposed directly to the internet. Access is mediated through the Kubernetes Service abstraction, which acts as an internal gateway, masking the dynamic IP addresses of the individual pods.

CI/CD Efficiency
Off-Cluster Building: Building Docker images is CPU-intensive.
Design Decision: We offloaded the build process to GitHub Actions (SaaS) rather than running a Jenkins server on the same EC2 instance.
Benefit: This preserves the limited CPU credits of the EC2 instance solely for serving user traffic and running the application, rather than wasting them on compilation tasks.

3 Data design:

3.1 Data Flow Diagram



3.2 Data Access Mechanism :

This section defines how the application connects to, reads from, and writes to the stateful database layer. It bridges the gap between the application's internal code and the external Kubernetes infrastructure.
~Object-Relational Mapping (ORM)
Framework: Entity Framework (EF) Core.
Purpose: EF Core acts as the data access layer, translating C# LINQ queries into the native SQL dialect of the underlying database (e.g., PostgreSQL or SQL Server).

Migration Strategy: Database schema migrations are executed via a dedicated Kubernetes Job prior to the main application pods starting. This ensures the database schema is always in sync with the new application version before any user traffic is served.

~ Secure Connection String Injection (The DevOps Approach)
Hardcoding connection strings in C# code or appsettings.json is a severe security vulnerability. For this deployment, we utilize a cloud-native configuration management strategy.

Kubernetes Secrets: The raw connection string (containing usernames and passwords) is stored securely as a base64-encoded Kubernetes Secret object in the cluster.

Environment Variables: In the deployment.yaml, this Secret is injected directly into the application container as an environment variable at runtime.

ASP.NET Core Override: The .NET configuration builder is designed to automatically override appsettings.json values with environment variables.

DevOps Note (Linux Compatibility): Because Linux environment variables do not support the colon (:) character used in JSON hierarchies, we use the double underscore (__) syntax. To override the "ConnectionStrings":
{ "DefaultConnection": "..." } value, the Kubernetes environment variable is

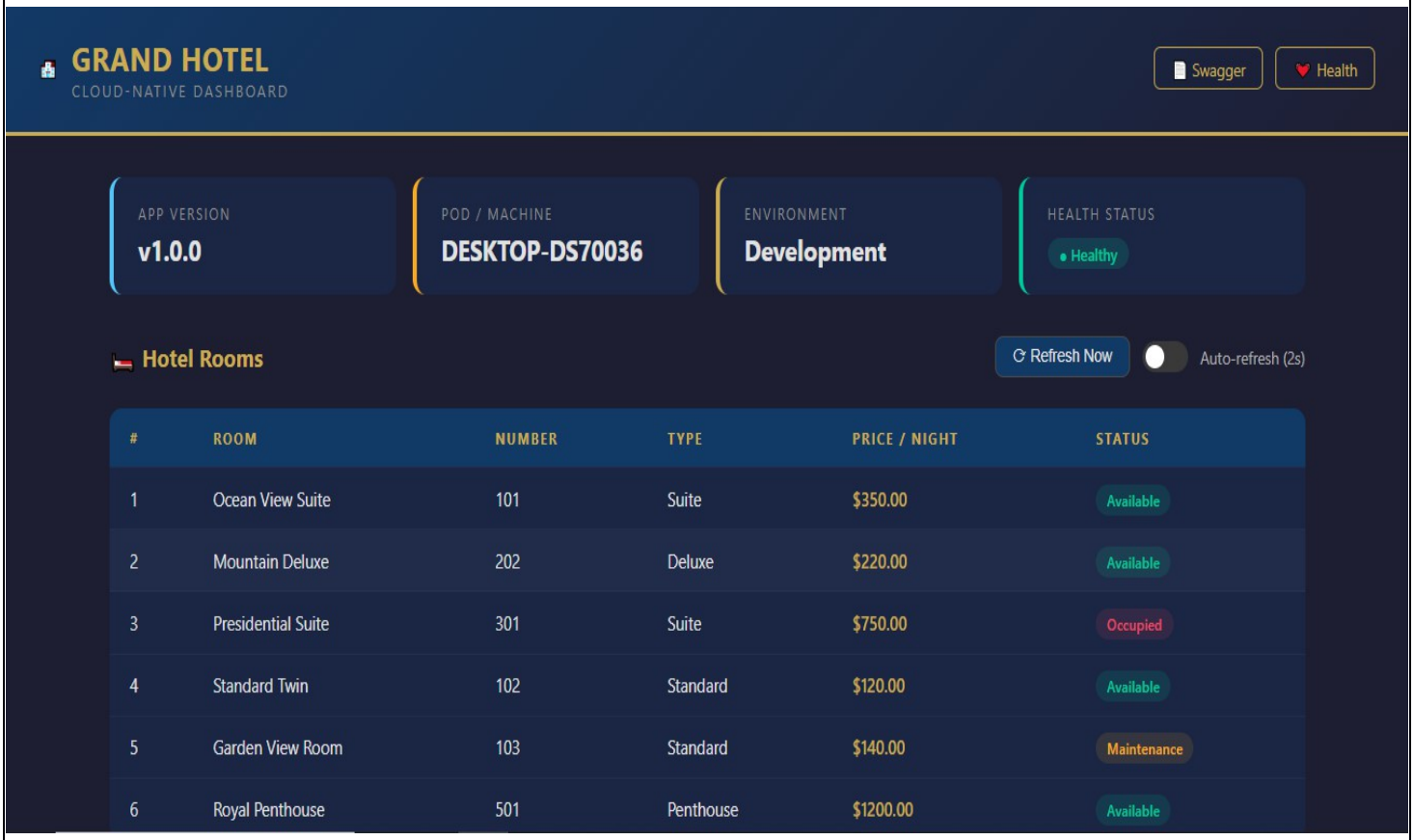
named ConnectionStrings__DefaultConnection.

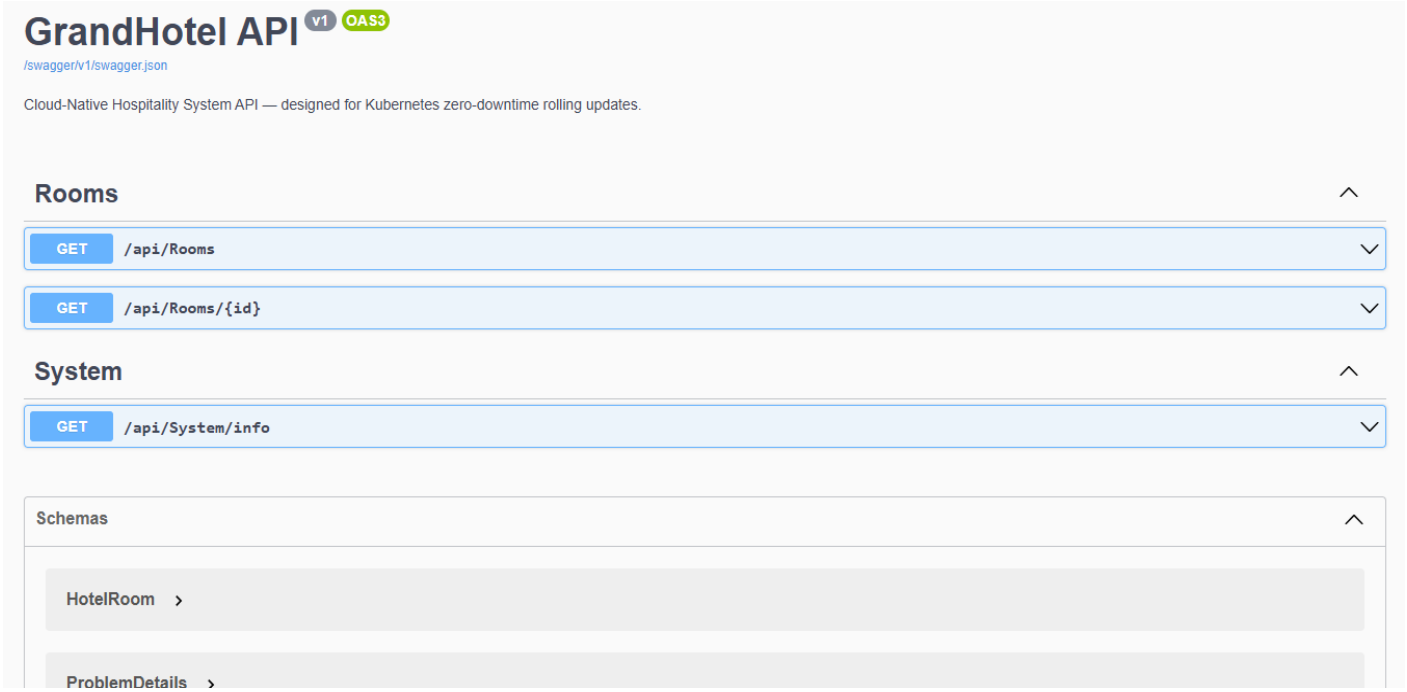
~In-Cluster Service Discovery
The application does not rely on static IP addresses to locate the database. Instead, it utilizes the Kubernetes internal DNS system.

Mechanism: If the database is hosted within the cluster, the connection string targets the database's Kubernetes Service name (e.g., Server=hospitality-db-svc.default.svc.cluster.local;...).

Benefit: If the database Pod crashes and is rescheduled with a completely new IP address, the Service DNS name remains constant. Kube-proxy seamlessly handles the internal routing, meaning the .NET application never loses its connection pathway.

4 INTERFACE :


GRAND HOTEL
CLOUD-NATIVE DASHBOARD
Swagger
Health
APP VERSION
v1.0.0
POD / MACHINE
DESKTOP-DS70036
ENVIRONMENT
Development
HEALTH STATUS
Healthy
Hotel Rooms
Refresh Now
Auto-refresh (2s)
ROOM NUMBER TYPE PRICE / NIGHT STATUS
1 Ocean View Suite 101 Suite $350.00 Available
2 Mountain Deluxe 202 Deluxe $220.00 Available
3 Presidential Suite 301 Suite $750.00 Occupied
4 Standard Twin 102 Standard $120.00 Available
5 Garden View Room 103 Standard $140.00 Maintenance
6 Royal Penthouse 501 Penthouse $1200.00 Available


GrandHotel API
v1
OAS3
/swagger/v1/swagger.json
Cloud-Native Hospitality System API — designed for Kubernetes zero-downtime rolling updates.
Rooms
GET /api/Rooms
GET /api/Rooms/{id}
System
GET /api/System/info
Schemas
HotelRoom
ProblemDetails

5 State And Session Management:

In a Kubernetes environment, application instances (Pods) are strictly ephemeral. During the implemented "Rolling Update" process, Kubernetes systematically shuts down older version pods and spins up newer versions. If the Hospitality Application relies on default In-Memory (In-Proc) session management, any active user session residing on a terminating pod will be permanently lost, resulting in a disruptive user experience (e.g., unexpected logouts, lost cart data, or interrupted booking flows).To ensure high availability and support seamless rolling updates, the application was designed to be stateless. This means that the .NET application containers do not store any persistent user data or session identifiers locally on the pod's file system or RAM.

6 Non-Functional Requirements:

6.1 Security Aspects:

1. Infrastructure and Network Security (AWS EC2)

The foundational layer of the application is hosted on an AWS EC2 instance. Security at this level focuses on minimizing the attack surface from the public internet.

Principle of Least Privilege (Security Groups): The AWS Security Group acts as

a strict virtual firewall. Inbound rules are explicitly limited to:
Port 22 (SSH) - Restricted to the administrator's specific IP address rather than 0.0.0.0/0.
Port 80/443 (HTTP/HTTPS) - Open to the web for end-user traffic.
Port 30007 (NodePort) - Explicitly opened only for the application's Kubernetes Service.
Key-Pair Authentication: Password authentication for SSH is disabled. Access to the server is strictly enforced via an RSA .pem key pair.

2. CI/CD and Secrets Management (GitHub Actions)
Automated pipelines are frequent targets for supply chain attacks. The CI/CD workflow was hardened to prevent unauthorized access and credential leakage.
Secure Credential Storage: Sensitive data, such as DockerHub credentials, AWS Host IPs, and SSH private keys, are never hardcoded in the repository. They are stored securely using GitHub Actions Secrets, which encrypts the data and masks it in all pipeline execution logs.
Ephemeral Build Environments: Every GitHub Actions workflow runs on a fresh, isolated ubuntu-latest virtual machine that is destroyed immediately after the pipeline finishes, ensuring no residual data is left behind.

3. Container Security (Docker & Image Build)
The container image is the immutable artifact running our application. If the image is vulnerable, the entire cluster is at risk.
Multi-Stage Builds: The Dockerfile utilizes a multi-stage build process. The SDK (build tools, source code, and package managers) is left behind in the build stage. The final runtime image contains only the compiled .dll files and the minimal ASP.NET runtime required to execute the app. This drastically reduces the container's attack surface.
Minimal Base Images: By using official, minimal Microsoft Linux base images, we avoid packaging unnecessary OS utilities (like curl or bash in the final layer), making it much harder for an attacker to execute arbitrary commands if a vulnerability is found.

6.2 Performance Aspects

Deploying a compiled framework like .NET alongside a Kubernetes orchestrator (Minikube) on a strictly constrained environment (AWS t2.micro) requires aggressive performance tuning. The strategy focused on minimizing footprint, managing memory aggressively, and ensuring stable rolling updates without node crashes.

1. Infrastructure Trade-offs: The Swap Memory Compromise
A standard Minikube installation requires a minimum of 2 CPUs and 2GB of RAM. To operate within the 1GB RAM constraint of the AWS Free Tier, a 2GB disk-based Swap file was provisioned.
The Trade-off: While Swap prevents the EC2 instance from suffering fatal Out-Of-Memory (OOM) kernel panics, it introduces high disk I/O latency.
The Result: Application startup times and image pulling are inherently slower. The Kubernetes architecture had to be tuned to tolerate this latency rather than fight it.

2. Kubernetes Resource Management (Requests vs. Limits)
To prevent the .NET pods from aggressively consuming the host node's limited memory (and triggering an OOMKilled status), strict resource boundaries were defined in the deployment manifest:

Resource Requests: Set to 128Mi Memory and 100m CPU. This guarantees the pod has the bare minimum resources required to boot the ASP.NET runtime.
Resource Limits: Capped at 256Mi Memory and 200m CPU. This creates a hard ceiling, ensuring no single pod can monopolize the t2.micro's CPU, leaving enough resources for the Kubernetes Control Plane to function.

7 REFERENCE

1. Kubernetes Orchestration and Deployment Strategies

- Kubernetes Official Documentation: Managing Deployments
    - *Topic:* Pod lifecycle, ReplicaSets, and creating declarative deployment manifests.
    - *Link:* kubernetes.io/docs/concepts/workloads/controllers/deployment/
- Kubernetes Official Documentation: Performing a Rolling Update
    - *Topic:* Zero-downtime strategies, configuring maxSurge and maxUnavailable parameters.
    - *Link:* kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/

- Kubernetes Official Documentation: Resource Management for Pods and Containers
    - *Topic:* Setting CPU and Memory requests and limits to prevent Out-Of-Memory (OOM) node crashes.
    - *Link:* kubernetes.io/docs/concepts/configuration/manage-resources-containers/

2. State and Session Management

- Microsoft Learn: Distributed Caching in ASP.NET Core
    - *Topic:* Architectural overview of externalizing state and session data to avoid local memory dependency.
    - *Link:* learn.microsoft.com/en-us/aspnet/core/performance/caching/distributed
- Microsoft Learn: Microsoft.Extensions.Caching.StackExchangeRedis
    - *Topic:* Official implementation guide for integrating Redis as a distributed cache provider in C# .NET 8.
    - *Link:* learn.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.rediscacheservicecollectionextensions

3. Application Performance and Tuning

- Microsoft Learn: Garbage Collection (GC) in .NET
    - *Topic:* Understanding "Workstation GC" vs. "Server GC" and memory footprint optimization for low-spec (t2.micro) container environments.
    - *Link:* learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals
- Docker Documentation: Multi-Stage Builds

- *Topic:* Securing and shrinking container image sizes using alpine-chiseled runtimes.

- *Link:* docs.docker.com/build/building/multi-stage/

4. CI/CD and Infrastructure

- GitHub Actions Documentation: Building and Testing .NET
    - *Topic:* Automating the CI pipeline and securely managing GitHub Secrets.

    - *Link:* docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-net

- Minikube Documentation: Resource Requirements
    - *Topic:* Minimum system specifications and overriding constraints using Swap memory on AWS EC2.

    - *Link:* minikube.sigs.k8s.io/docs/start/