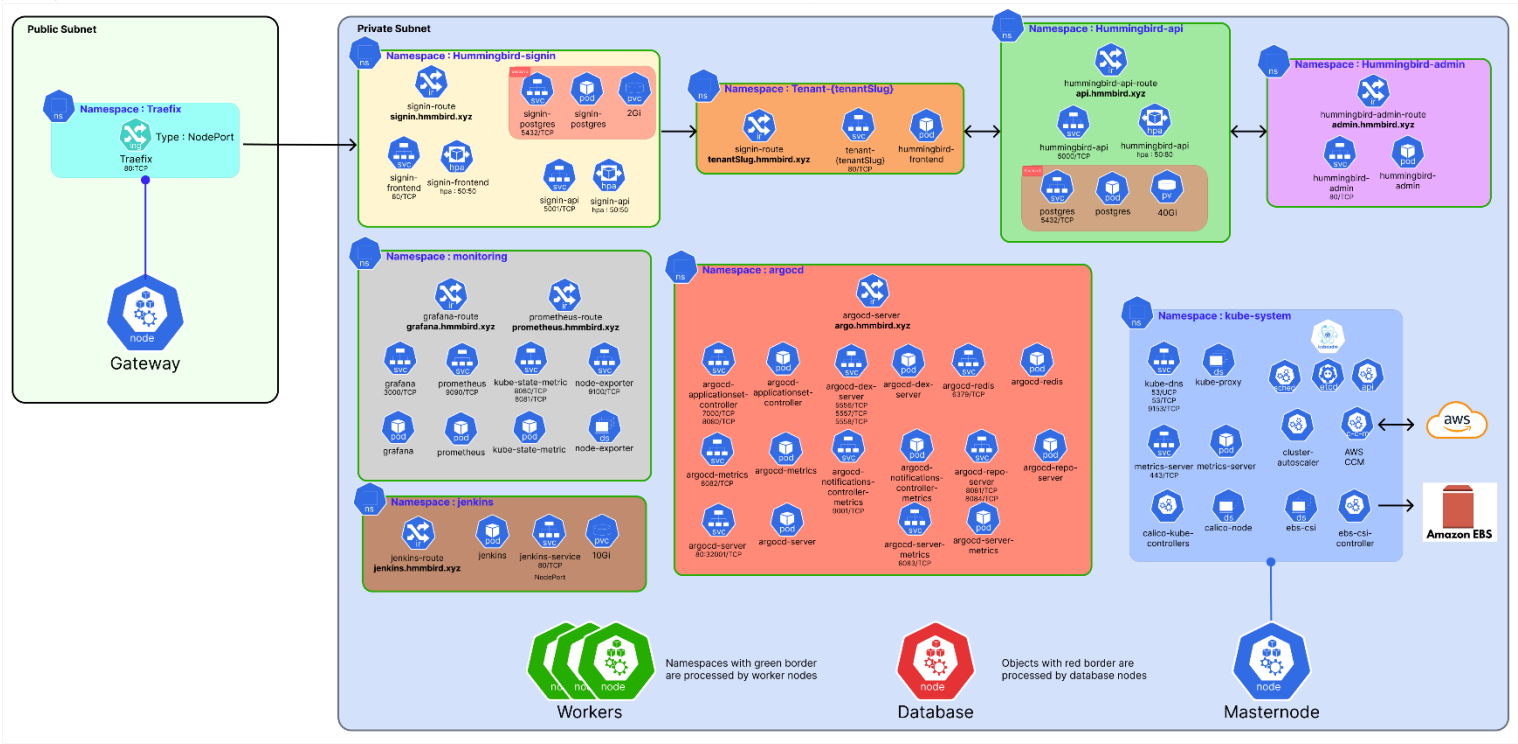
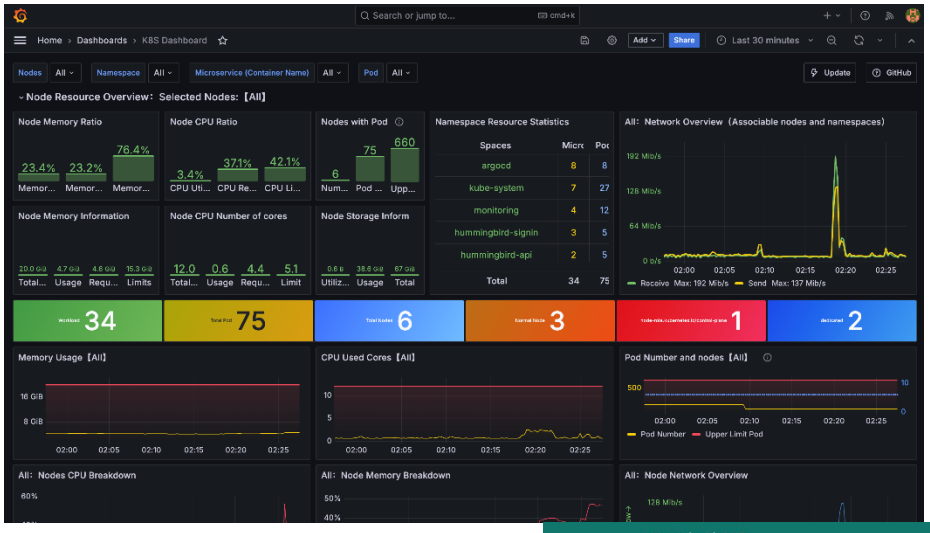
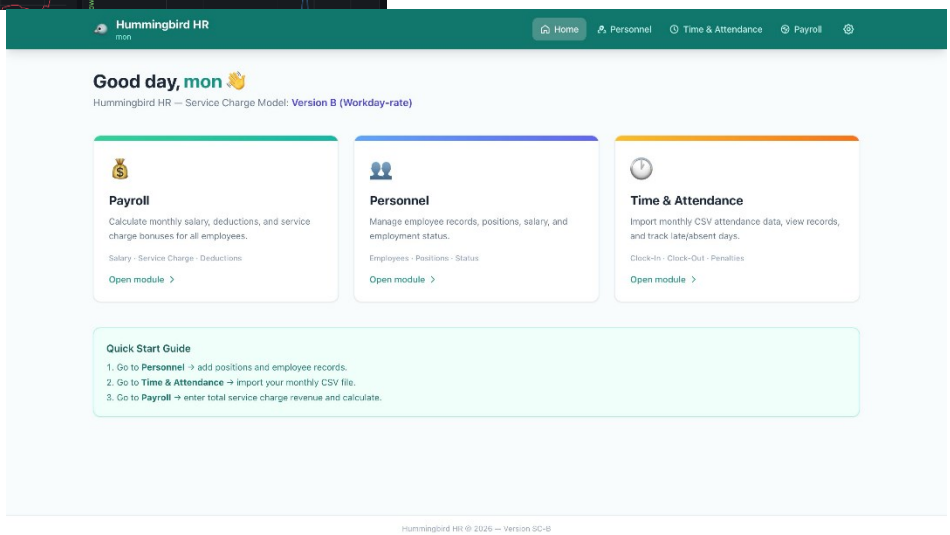
Multi-Tenant Cloud Architecture for SaaS Hotel Human Resource Management Application







CHULA INTERNATIONAL SCHOOL OF ENGINEERING
The ingenuity of CHULA ENGINEERING

Engineering Project Final Report

Multi-Tenant Cloud Architecture for SaaS Hotel Human Resource Management Application

Submitted by

Panupong Phamornbupachart	6538146021
Pemapol Sripratipbundit	6538149021

Approved by

Project Advisor : Asst. Prof. Kunwadee Sripanidkulchai, Ph.D.

Project Committee Member : Aung Pyae, Ph.D.

Project Committee Member : Asst. Prof. Sukree Sinthupinyo, Ph.D.

ABSTRACT

This report showcases the design and evaluation of a multi-tenant cloud architecture. This architecture is demonstrated through a Software-as-a-Service Human Resource Management (HRM) application specifically for hotels. The system was designed around a Kubernetes-based infrastructure with tenant namespace separation, a single centralized sign-in service, frontend customization for each tenant, and modular API layer which supports basic HR functions. Infrastructure features include autoscaling and deployment automation to handle demand spikes such as month-end payroll runs, tenant onboarding, and seasonal staffing events. The resulting design demonstrates how cloud-native architecture can support a secure, adaptable, and cost-effective SaaS platform. It also acts as a reference for other multi-tenant systems that need a sudden performance leap.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The hotel and hospitality industry faces growing pressure to update and modernize their digital infrastructure, including their Human Resource Management (HRM) systems, to keep up with industry standards, reduce costs, and improve operational efficiency. Typical existing on-premises HR solutions require high capital investments into dedicated IT infrastructure and specialized personnel for implementation, integration, and maintenance. As new cloud technologies develop, the industry is starting to shift towards cloud-based solutions like the emergence of Software-as-a-Service (SaaS) models, which offer a compelling alternative to on-premises. This model provides a cost-effective, and scalable solution accessible from on-site and remote locations. This project addresses a critical gap for multi-tenant SaaS HR applications where hotel operations differ from other organizations in terms of usage, feature requirements, and compliance.

SaaS has transformed the business software industry by allowing providers to deliver their product to their entire user base from shared infrastructure while at the same time reducing operational overhead. Every design decision from network topology to the application data model must make sure that each tenant's data is completely isolated from each other while still sharing computation and storage resources. This project presents a full design implementation and validation of such a system.

This project, named Hummingbird, designs, implements, and validates a multi-tenant cloud architecture for a SaaS Hotel Human Resource Management application targeting Thailand's hospitality industry. The system is built on a self-managed Kubernetes cluster deployed on Amazon Web Services (AWS) EC2 instances, with namespace-per-tenant isolation, a database-per-tenant PostgreSQL model, and a HRM API covering core functional modules for demonstration purposes. The system is designed to support up to five hotel tenants with 15,000 total employees and 10,000 concurrent API calls. Infrastructure automation is handled through a Terraform infrastructure-as-code configuration, a Jenkins and ArgoCD CI/CD pipeline, and Kubernetes Horizontal Pod Autoscaling (HPA) to manage demand spikes such as month-end payroll runs and seasonal staff onboarding events.

The project produces the following software deliverables: SaaS HRM Application, AWS Infrastructure Code, and Database Schema. The SaaS application is a web-based multi-tenant application with a centralized sign-in portal, per-tenant frontend, and three API modules for demonstration and infrastructure testing. The AWS Infrastructure Code is a Terraform-based infrastructure-as-code configuration (main.tf and variables.tf) for automated provisioning of all cloud computing resources including the VPC, subnets, Security Groups, and EC2 instances. Database Schema is a PostgreSQL schema optimized for the database-per-tenant isolation model and the HRM data requirements.

The project also produces the following documentation deliverables: Technical Architecture and Testing Report. The Technical Architecture document covers the entire design specification of the infrastructure, including Kubernetes cluster topology, namespace-per-tenant isolation strategy, AWS network design, and Security Group hardening policy. The Testing Report presents the results from load testing scenarios simulated using Grafana k6, covering steady-state normal usage, bulk month-end payroll processing, new tenant onboarding, and seasonal staff

onboarding and offboarding, which serve as the primary validation of the architecture against its design requirements.

2 BACKGROUND

The original concept of multi-tenancy originated in the 1960s with machines like the IBM OS/360 mainframe, which allows multiple users to use a single system through virtual memory and interrupt scheduling (IBM, n.d.). The modern SaaS era began as the founding of Salesforce in 1999 and its launch in 2000. Salesforce popularized multi-tenant architecture by delivering their Customer Relationship Management (CRM) application as one shared application instance to all of their customers (Mizono, 2022).

Around the mid-2000s, the adoption of SaaS rapidly expanded as cloud providers such as Amazon Web Services (AWS) and Microsoft Azure transformed and improved reliability through commodity hardware and automated failover, allowing for "cattle not pets" architectures where stateless application instances could scale horizontally (Wiegand, 2022). Multi-tenancy eventually evolved into three main tenant isolation strategies, primary involving database isolation: shared database and schema, which offers lowest isolation but highest density; shared database but separate schemas, which offers a balanced approach; and separate databases per tenant, which offers the highest isolation but also at a great cost (Ali, 2026).

In the hotel industry specifically, SaaS helps address the common complex requirements such as variable shift payroll, seasonal staffing, tip and service charge processing, and multi-property (chain) compliance. A case study from Switzerland shows how utilizing a SaaS HR platform increased manager productivity by 33% and improved employee satisfaction by 24% by streamlining workflows (Darly Solutions, n.d.). Hospitality workloads also typically introduce unique user spikes such as month-end payroll runs across chains, rapid onboarding of temporary staff during tourism peaks, and high-volume attendance imports.

Kubernetes is an open-source container orchestration platform originally developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF). It automates the deployment, scaling, and lifecycle management of containerized applications across clusters of machines. A Kubernetes cluster consists of a control plane, which manages cluster state and scheduling, and worker nodes, which host the application workloads in units called pods. Kubernetes supports multi-tenancy with the use of "namespaces", which provides logical isolation for Application Programming Interfaces (APIs), Role-Based Access Control (RBAC), and workloads within a single Kubernetes cluster. Namespaces allows for soft multi-tenancy by permitting object name overlap across tenants and security policies like NetworkPolicies and resource quotas to prevent resource contention (Kubernetes, n.d.). Red Hat's reference architecture describes namespace-per-tenant isolation for SaaS applications, where all of one tenant's deployment and services operate within one dedicated namespace. Tenant boundaries are further enforced using network policies using Calico or Cilium (Reselman, 2022). Kubernetes documentation also describes "hybrid" and "bridges" models, the usage of shared control planes, and one namespace for one tenant, as a balanced method for Business-to-Business (B2B) SaaS since it combines the benefits of cost efficiency while maintaining strong logical isolation. To

enable more customized tenant-specific control planes, virtual clusters (vCluster) can be used on top of shared worker nodes.

Current commercial offerings available overseas serve small and medium-sized businesses (SMB) payroll and benefits processing through cloud-based HR platforms, while multi-tenant solutions usually implement isolation using EF Core with PostgreSQL, relying on patterns like database-level isolation (e.g. row-level security) or database-per-tenant (Dhandala, 2026). Open-source projects with Kubernetes patterns such as Loft's vCluster and Spectro Cloud leverage namespaces for logical isolation, Calico network policies for pod-level traffic control, and Horizontal Pod Autoscaler (HPA) to handle workload spikes dynamically (vCluster, n.d.).

Enterprise HR SaaS solutions such as Workday, SAP SuccessFactors, ADP Workforce Now, and Zoho People offer many features but are subsequently priced for large enterprise budgets and, without costly modifications, not optimized specifically for the Thai hotel industry, including pay structures and shift timings. Smaller hotels have limited options that are both affordable and locally compliant. This project specifically targets this market gap, optimized specifically for the Thai hotel industry.

3 DESIGN REQUIREMENTS

The design requirements for this project were derived from the target use case of hotel human resource management in Thailand, the constraints of a SaaS architecture model serving multiple hotel clients, and the limited budget and timeline constraints of the project. Infrastructure requirements (Section 3.1) cover the multi-tenancy isolation model, automated tenant provisioning, and horizontal scaling. Application requirements (Section 3.2) cover the functional features and user-facing performance targets.

3.1 Infrastructure Requirements

The infrastructure must satisfy three primary requirements: multi-tenancy with strict tenant isolation, automated tenant provisioning, and dynamic horizontal scaling. The quantified targets for these properties are presented in Table 3.1. The following sections below describe the design intent behind each requirement.

3.1.1 Multi-Tenancy and Noisy Neighbor Isolation

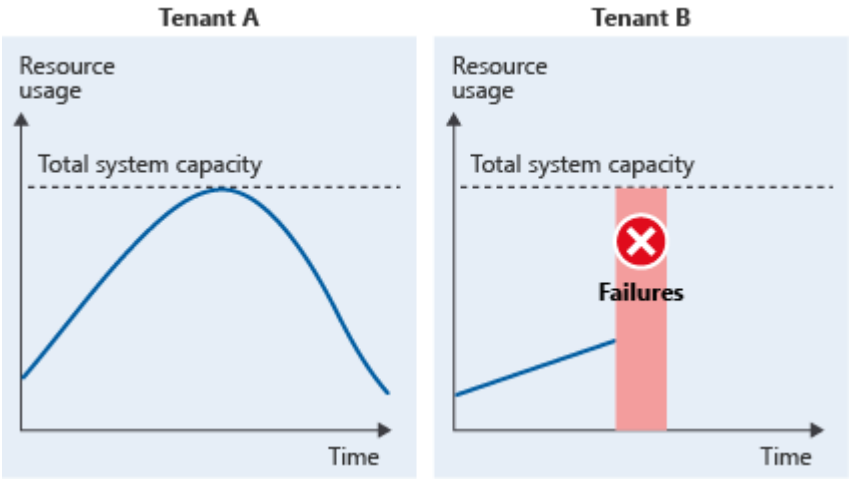


Figure 3.1: Noisy Neighbor Antipattern (Downs et al., n.d.)

The system must support multiple independent hotel organizations (tenants) on a single shared infrastructure without any tenant's own workload degrading the performance or exposing the data of another tenant. This type of problem is usually known as the "Noisy Neighbor" problem in multi-tenancy-based systems (Downs et al., n.d.). The architecture addresses this issue through three mechanisms: Kubernetes namespace-per-tenant isolation which provides logical separation of all workloads, services, and RBAC policies per tenant; Calico NetworkPolicies, which enforce pod-level traffic rules so that pods in one tenant namespace cannot initiate connections to pods in another; and the database-per-tenant model, which ensures that no SQL query from one tenant can ever target another tenant's PostgreSQL instance. Combined, these layers form a "defense-in-depth" boundary that isolates and prevents both accidental and deliberate attempts of cross-tenant data access.

Tenant Data Isolation is a binary pass/fail requirement and is non-negotiable from a data privacy and regulatory standpoint. Employee payroll and HR data is an extremely sensitive category of personal information, and any cross-tenant exposure would render the system unsuitable for production deployment.

3.1.2 Tenant Provisioning

The system must be capable of onboarding new hotel tenants automatically and within an acceptable time window from the moment the admin API call is made to the moment that the new tenant's environment is live and usable. Automated provisioning is crucial to the SaaS commercial model as a platform that requires manual administrator configuration of the infrastructure for each new tenant client is impractical and cannot scale. Provisioning time is measured as the total wall-clock duration from the admin API call that initiates provisioning to the moment the new tenant's frontend pod is confirmed reachable. This end-to-end duration is decomposed into three sequential steps: the infrastructure provisioning step (namespace creation, frontend deployment, IngressRoute configuration, and migration), the employee data seeding step, and the attendance data seeding step. The combined completion time of all three steps represents the full onboarding duration from an administrator's perspective.

3.1.3 Scaling

Horizontal scaling refers to the handling of increased load by adding more instances of a service rather than upgrading the hardware specification of a single machine. In contrast to vertical scaling, which increases the CPU or RAM of an existing server and requires downtime to resize, horizontal scaling adds or removes identical copies of an application pod, distributing the workload across them. This approach is well-suited to containerized, stateless services because each pod is interchangeable and can be started or stopped in seconds. For the Hummingbird system, the stateless HRM API is the primary scaling target: additional API pod replicas can be spun up within the existing Worker Nodes to absorb bursts of incoming requests without touching the underlying infrastructure.

The primary mechanism for automated horizontal scaling is the Kubernetes Horizontal Pod Autoscaler (HPA), which is configured to monitor the CPU utilization of the HRM API deployment and trigger additional pod replicas when average CPU usage exceeds 50% or memory usage exceeds 80%. The HPA operates between a minimum of 8 replicas and a maximum of 50 replicas, allowing the system to absorb workload bursts such as month-end payroll runs or seasonal staff onboarding events while returning to the minimum footprint during idle periods. This elasticity directly supports the monthly infrastructure cost target by avoiding persistent over-provisioning.

When pod-level scaling alone is insufficient to meet demand, specifically when all existing Worker Nodes are fully saturated and no schedulable capacity remains, the Kubernetes Cluster Autoscaler is responsible for scaling at the node level. The Cluster Autoscaler monitors for pods that remain in a Pending state due to insufficient cluster resources and responds by provisioning additional EC2 Worker Node instances through the AWS Auto Scaling Group. Conversely, when

node utilization falls below the configured threshold for an extended period, the Cluster Autoscaler drains and terminates underutilized nodes to reduce infrastructure costs.

Table 3.1: Infrastructure Design Requirements

Requirement	Units / Criteria	Marginal Value	Ideal Value
Tenant Data Isolation	Pass/Fail: no cross-tenant data access	Pass	Pass
Tenant Onboarding Time	Minutes from API call to live tenant	10 minutes	5 minutes
Steady-State Throughput	Requests per second without HPA firing	50 RPS	75 RPS
Tenant User Capacity	Total employees across all tenants	10,000	15,000–30,000
Monthly Infrastructure Cost	USD per month	< $350	< $300

3.2 Application Requirements

For the purposes of infrastructure demonstration, and user feature and load testing, an application with basic minimal functions is required. The following functional requirements define what the system must do from the user's perspective:

- Multi-tenant application supporting complete data isolation between hotel tenant clients
- Employee information database with full lifecycle management (add, update, remove)
- Payroll calculation supporting Thailand's hotel industry compensation structures including hourly wages, tips, and service charges in two configurable models (Version A: fixed service charge rate per position; Version B: workday-weighted distribution)
- Time and attendance tracking via CSV import from physical attendance machines

Non-functional requirements are presented in Table 3.2. These targets reflect infrastructure-observable behavior on the test cluster hardware (t3.medium Worker Nodes, t3.xlarge Database Node). Absolute response latency targets are deferred to production-class hardware, as t3.medium burstable instances have a 20% baseline CPU allocation that makes latency-based SLAs non-representative of production conditions. The requirements instead focus on error rate thresholds and scaling behavior, which are hardware-independent indicators of whether the architecture is functioning correctly. API error rate is measured at two distinct load levels, steady-state and spike, to capture both normal-day and peak-event behavior.

Table 3.2: Application Design Requirements

Requirement	Units / Criteria	Marginal Value	Ideal Value
API Error Rate (steady-state)	% of failed HTTP requests at 75 RPS	< 5%	< 1%
API Error Rate (payroll spike)	% of failed HTTP requests at 1,000 VUs	< 10%	< 5%
Tenant Provisioning Latency (p95)	ms per provisioning request at 50 VUs	< 10,000 ms	< 5,000 ms
Cluster Autoscaler Response	Node count increases under 5,000 VU load	Pass	Pass

4 DESIGN DESCRIPTION

4.1 Overview

The Hummingbird system is deployed on AWS in the ap-southeast-1 (Singapore) region. All resources reside within a single VPC. External traffic enters through the Gateway node, which uses NodePort to direct traffic to the ingress controller, Traefik. Traefik routes the user to the correct tenant namespace based on the HTTP Host header. All application data is stored in dedicated PostgreSQL databases on in-cluster nodes that are unreachable from the public internet.

The design description is organized into three main sections: Feature Design, Cluster Design, and Application Design. Feature Design (Section 4.2) describes the three primary feature workflows: the normal user traffic flow, the auto-scaling behavior, and the CI/CD pipeline. Cluster Design (Section 4.3) covers the cluster design in detail, including the general architecture and database isolation strategy (4.3.1), the Kubernetes namespace logical design (4.3.2), the physical node configuration and Terraform provisioning (4.3.3), the database design and schema (4.3.4), the AWS networking and Security Group policy (4.3.5), and the monitoring and observability stack (4.3.6). The Application Design is also described in Section 4.4.

4.2 Feature Design

4.2.1 Normal User Traffic Flow

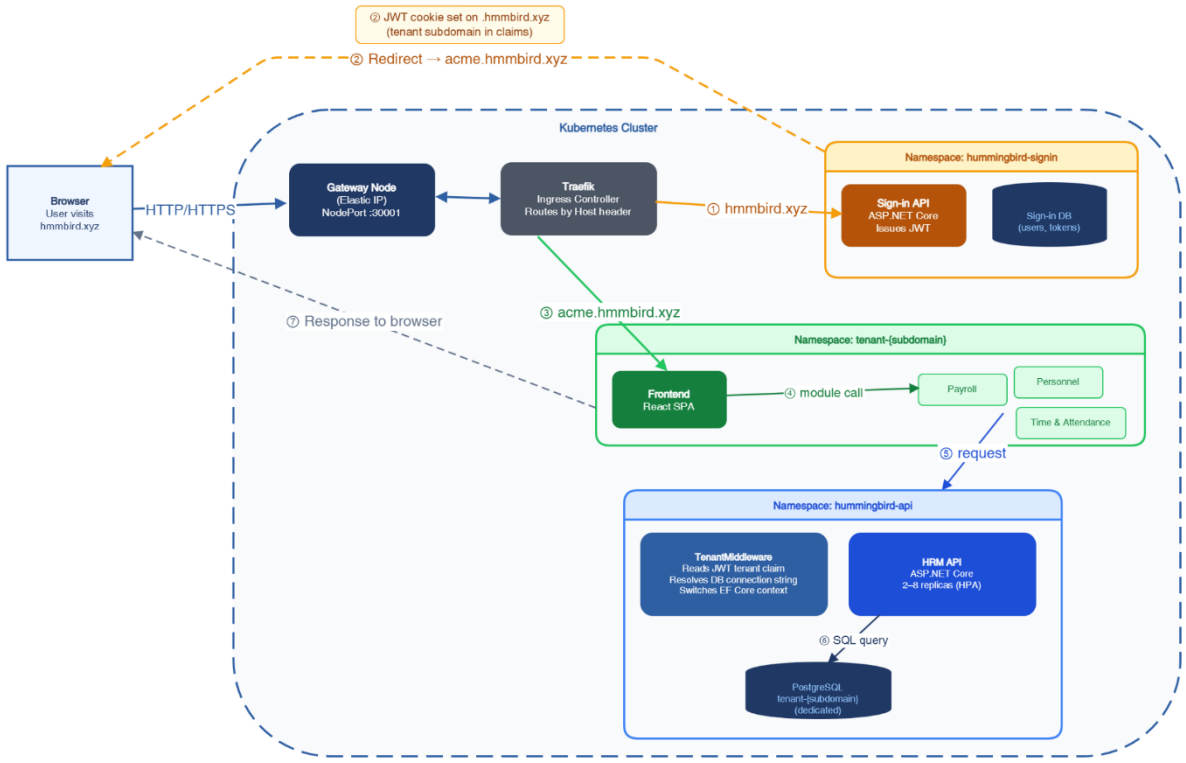


Figure 4.1: Normal User Traffic Flow

A normal user follows a specific flow from the browser through the sign-in service to the tenant-specific frontend and then to the shared HRM API. The user visits hmmbird.xyz and authenticates via the centralized sign-in service. The API issues a JSON Web Token (JWT) with the tenant subdomain encoded in the token and sets it as a cookie on .hmmbird.xyz for cross-domain sharing. The browser is then redirected to the correct tenant subdomain (e.g. acme.hmmbird.xyz) where the cookie is used to skip a second login. From the tenant frontend, the user interacts with one of the application modules. Each module communicates with the shared HRM API. The TenantMiddleware in the API finds the correct PostgreSQL connection string from the JWT and switches the EF Core database context on every request, ensuring full data isolation at the application layer.

4.2.2 Automatic Scaling

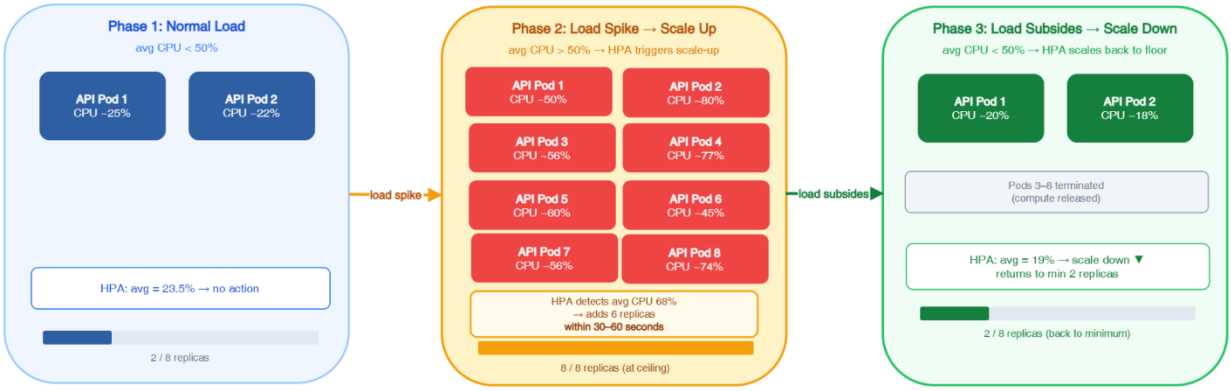


Figure 4.2: Auto Scaling — Scaling Up and Scaling Down

The Horizontal Pod Autoscaler (HPA) monitors CPU and memory utilization of the HRM API deployment and triggers scaling events when the average CPU or memory usage across all API pods exceeds 50% and 80% respectively. The HPA is configured with a minimum of 8 replicas and a maximum of 50 replicas. When a load spike is detected, for example during a month-end payroll run, the HPA creates additional API pods within approximately 30–60 seconds. As load subsides after the peak window, the HPA scales the deployment back down to the minimum replica count, releasing compute capacity. This elasticity prevents over-provisioning during normal hours while guaranteeing headroom for demand spikes without manual intervention.

In the event that pod-level scaling alone does not satisfy the demand including when newly scheduled pods are stuck in a Pending state, node-level scaling is handled by the Kubernetes Cluster Autoscaler, deployed in the kube-system namespace. The Cluster Autoscaler operates in AWS auto-discovery mode, monitoring the cluster for unschedulable pods and responding by increasing the desired capacity of a dedicated AWS Auto Scaling Group (ASG). The ASG is configured with a t3.medium launch template that includes a kubeadm join script in its user data, so newly provisioned EC2 instances automatically install the Kubernetes toolchain and join the cluster without manual intervention. The ASG starts at a baseline of zero additional nodes and can scale up to five, meaning the total Worker Node pool can grow from the four static baseline nodes up to eight nodes under peak load. Node-level scaling operates on a slower timescale than pod-level scaling — typically several minutes for a new EC2 instance to boot, execute the kubeadm join sequence, and reach a schedulable Ready state. The two mechanisms therefore form a two-

tier elasticity model: the HPA responds immediately to CPU and memory pressure by expanding pod replica counts within available node capacity, while the Cluster Autoscaler expands the node pool itself when that capacity ceiling is reached.

4.2.3 Automated Tenant Provisioning

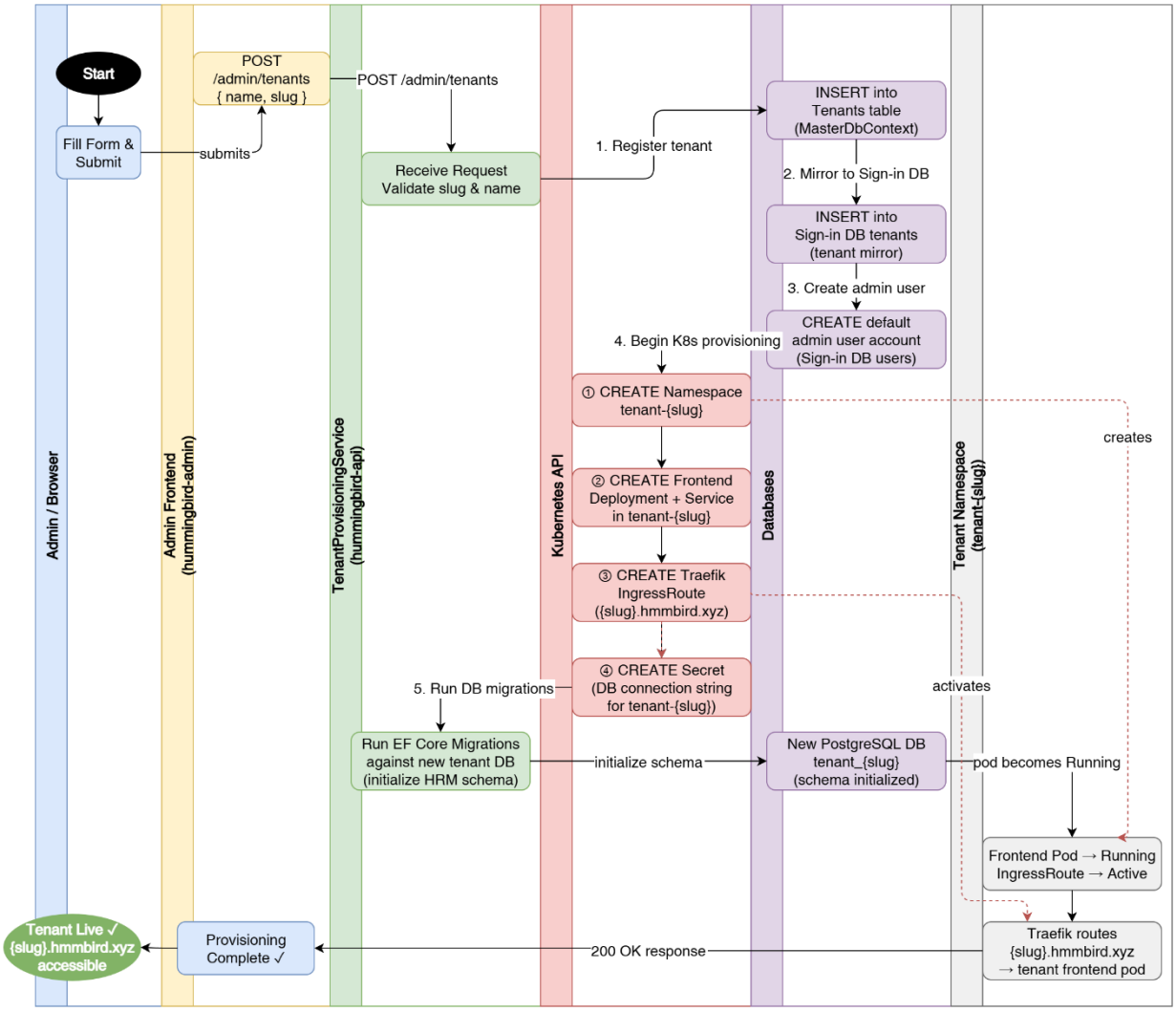


Figure 4.3: Automated Tenant Provisioning Flow

The automated tenant provisioning workflow allows a SaaS administrator to bring a new hotel tenant from zero to a fully operational, publicly accessible environment through a single action in the Admin Frontend, without any manual infrastructure intervention.

The workflow proceeds in four stages as shown in the figure above:

Stage 1: Admin Creates Tenant. The administrator navigates to the Admin Frontend and submits a new tenant creation form, providing the hotel name and the desired subdomain slug (for

example, "grand" for grand.hmmbird.xyz). The Admin Frontend sends a POST request to the /admin/tenants endpoint of the HRM API.

Stage 2: Tenant Records Are Created from Template. The TenantProvisioningService receives the request and performs two registration operations in sequence. First, it creates a new entry in the master Tenants table (MasterDbContext), recording the tenant name, subdomain, and the connection string for the new per-tenant PostgreSQL database. Second, it registers a corresponding tenant record in the Sign-in database so that the new tenant's users can authenticate through the centralized sign-in service. A default administrator user account is also created at this stage. Both records are derived from a standard template, ensuring that every onboarded tenant starts from a consistent, known configuration.

Stage 3: Kubernetes Resources Are Provisioned. Using the .NET Kubernetes client library, the TenantProvisioningService executes four sequential Kubernetes API calls against the cluster: (1) a new namespace named tenant-{slug} is created to provide logical isolation for all of the tenant's workloads; (2) a frontend Deployment and its associated ClusterIP Service are created within that namespace, deploying a pre-built tenant frontend container configured with the tenant's subdomain; (3) a Traefik IngressRoute resource is created, routing all HTTP/HTTPS traffic arriving on the tenant subdomain to the frontend Service; and (4) a Kubernetes Secret containing the per-tenant PostgreSQL connection string is created in the namespace, making it available to the HRM API's TenantMiddleware at runtime. After the Kubernetes resources are created, EF Core migrations are executed against the newly provisioned PostgreSQL database, initializing the full HRM schema.

Stage 4: Tenant Environment Is Live and Accessible. Once the frontend pod reaches the Running state and the Traefik IngressRoute is active, the tenant's subdomain (e.g., grand.hmmbird.xyz) resolves to the Gateway node's Elastic IP and Traefik begins routing requests to the new tenant frontend. The administrator can verify liveness by receiving a successful HTTP response from the IngressRoute probe. From this point, the hotel's HR administrators can log in through hmmbird.xyz, be redirected to their tenant subdomain, and begin configuring the system.

4.2.4 Automatic Deployment and Update (CI/CD)

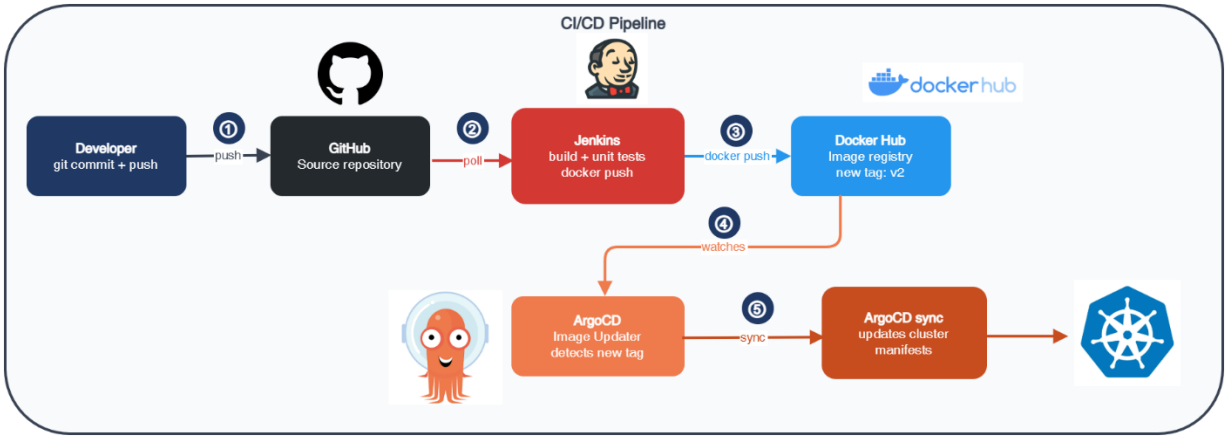


Figure 4.4: CI/CD Pipeline — From Developer to Cluster

Jenkins is an open-source automation server primarily used for Continuous Integration and Continuous Delivery (CI/CD). It helps developers automate building, testing, and deployment of code whenever changes are made. ArgoCD is an open-source Continuous Delivery (CD) tool designed for Kubernetes deployment. It automates application deployment to Kubernetes using Git as the source of truth, also known as GitOps.

With the Hummingbird CI/CD system, the deployment flow proceeds in five stages: (1) developer commits code to the GitHub repository; (2) Jenkins detects the commit via polling and triggers the build pipeline; (3) the pipeline builds and pushes a new Docker image to Docker Hub; (4) ArgoCD Image Updater detects the new image tag and updates the Kubernetes manifests; (5) ArgoCD synchronizes the cluster state to the new manifests, deploying the updated image to the target namespace.

4.3 Cluster Design

This section describes the Hummingbird system Kubernetes design in detail including the general design, logical design, physical design, and networking.

4.3.1 General Cluster Design

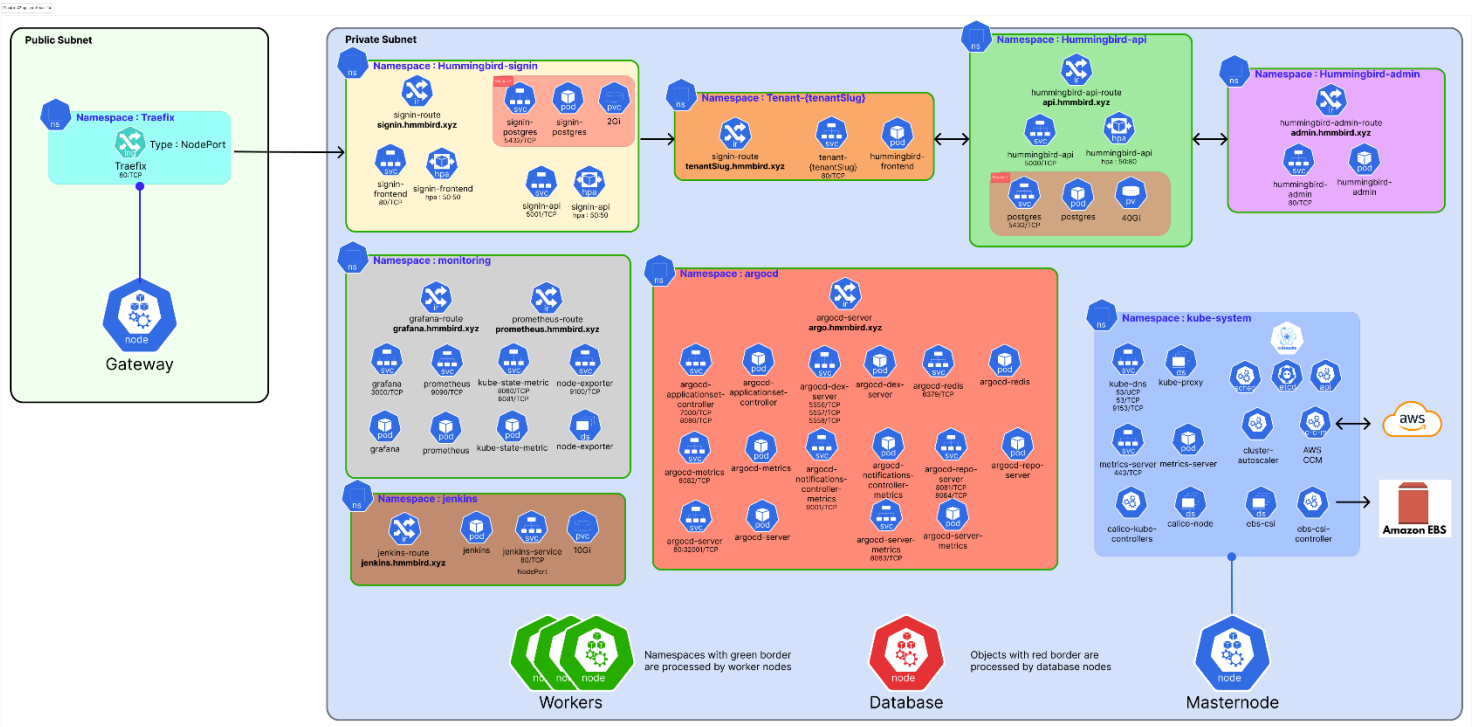


Figure 4.5: Master Kubernetes Cluster Design

The Hummingbird cluster is a self-managed Kubernetes cluster provisioned on AWS EC2 using kubeadm. The cluster consists of 7 namespaces (excluding kube-system, defaults, and tenants) for different functions of the system such as ingress, signin, API, and CI/CD tools. The cluster follows a database-per-tenant isolation model: each provisioned hotel tenant receives a dedicated PostgreSQL database, ensuring that no SQL queries from one tenant can ever reach

another tenant's data. A separate PostgreSQL instance serves as the master database for the sign-in service. The cluster currently serves five tenants, each with 3,000 employees seeded for load testing purposes (15,000 total).

4.3.2 Logical Design

The cluster is organized into namespaces for each of its functions. Fixed namespaces that are always present in the cluster include the following:

- traefik: Traefik ingress controller, receiving NodePort traffic from the Gateway node
- hummingbird-signin: centralized authentication service and its external database proxy
- hummingbird-api: shared HRM API, PostgreSQL master StatefulSet, RBAC resources
- hummingbird-admin: admin portal frontend
- devops-tools: Jenkins Deployment with PersistentVolume
- argocd: ArgoCD server and application controller
- monitoring: Prometheus and Grafana for monitoring the status of the cluster

When a tenant is onboarded, a new namespace (tenant-{slug}) is created automatically by the admin API for that tenant's customizable frontend configuration. One namespace is provisioned for each active tenant.

4.3.3 Physical Design

The cluster consists of four distinct node roles. Each has specific placement, computing, and networking requirements. The following table summarizes the specifications.

Table 4.1: EC2 Node Specifications and Roles

Node Role	Instance	vCPU	RAM	Storage	Subnet	Cost / Month
Control Plane (x1)	t3.medium	2	4 GB	16 GiB	Private	$36.21
Worker Node (x4)	t3.medium	2	4 GB	16 GiB	Private	$36.21 each
Database Node (x1)	t3.xlarge	4	16 GB	128 GiB	Private	$60.74
Gateway Node (x1)	t3.small	2	2 GB	8 GiB	Public + EIP	$18.10
NAT Gateway (x1)	AWS managed	—	—	—	Public	~$32.00

4.3.3.1 Control Plane Node

The Control Plane runs the primary components of the Kubernetes cluster consisting of Kubernetes API server, etcd, kube-scheduler, and the AWS Cloud Controller Manager. A static private IP is assigned so that all the Worker Nodes are able to maintain kubeconfig connection even when the instance restarts and AWS might otherwise reassign a dynamic address. An IAM role is assigned to the node, permitting it to call AWS APIs on behalf of the whole cluster. The node has no public IP and is only reachable from within the VPC.

4.3.3.2 Worker Nodes

The Worker Nodes run the application pods: tenant frontend deployments, shared API, CI/CD, and admin portal. There are three steps in the initialization join process. First, the node boots up and executes setup.sh script via the user_data attribute in the Terraform aws_instance resource; the script disables swap, loads the overlay and br_netfilter kernel modules, installs containerd, and installs kubeadm, kubelet, and kubectl. Second, a bootstrap token is generated on the Control Plane with 'kubeadm token create --print-join-command'. Lastly, the worker node executes the join command and registers with the Control Plane; after which the node is labelled and possibly tainted to restrict namespace pod assignment. Worker Nodes also have no public IP addresses.

When the cluster is under high load and pod scaling is insufficient, the Cluster Autoscaler automatically spins up new Worker Nodes to handle the additional traffic using the AWS Auto Scaling Group (ASG) with the Worker Node template configured. This allows the cluster to handle more traffic automatically without the need for administrative intervention.

4.3.3.3 Gateway Node

The Gateway Node is the only node with direct connection to the internet and is assigned a public IP address using AWS Elastic IP. This node has functions. Administrators SSH into the Gateway and use it as a jump host to reach private-subnet nodes, restricted by Security Group to the var.admin_cidrs list of trusted IP addresses. As a NAT host, the Gateway's public IP is the address seen by external services when the other private nodes require outbound connections. Gateway Node is labelled and tainted, with dedicated=traefik:NoSchedule, to allow only pods in Traefik namespace. This ensures that tenant workloads cannot degrade the latency of ingress traffic flowing through the cluster.

4.3.3.4 Database Node

The Database Node is provisioned as a t3.xlarge in the private subnet with a 30 GiB EBS volume to host all PostgreSQL StatefulSets. The node is tainted with workload=postgres:NoSchedule, ensuring that only the PostgreSQL StatefulSet pods, which explicitly tolerate this taint, can be scheduled there. This dedicated placement isolates database I/O and memory pressure from the application workload running on the t3.medium Worker Nodes, eliminating the resource contention that caused DB deadlocks under concurrent write load when databases and application pods shared the same nodes.

4.3.3.5 Terraform

Terraform is an infrastructure-as-code tool that allows the entire cluster infrastructure to be declared in version-controlled configuration files and provisioned reproducibly with a single terraform apply command. All EC2 nodes (the Control Plane, Worker Nodes, Database Node, and Gateway nodes) are created via aws_instance resources in main.tf, with each node type receiving a distinct configuration for instance type, subnet placement, Security Group assignment, and startup script via the user_data attribute.

The Control Plane node is provisioned as a t3.medium in the private subnet with a static private IP address and an IAM role attached. Its user_data script runs the kubeadm init sequence, configuring the pod network CIDR and generating the cluster's bootstrap token. Worker Nodes are provisioned as t3.medium instances in the private subnet; their user_data executes setup.sh, which installs the container runtime and the Kubernetes toolchain (kubeadm, kubelet, kubectl), but does not perform the cluster join automatically. The join step is manual because the bootstrap token must first be created on the Control Plane. The Gateway Node is provisioned as a t3.small in the public subnet and is assigned an Elastic IP (aws_eip) to give it a stable public address. The Database Node is provisioned as a t3.xlarge in the private subnet with a 128 GiB EBS volume; it is tainted workload=postgres:NoSchedule so that only the PostgreSQL StatefulSet pods schedule onto it. A NAT Gateway is provisioned in the public subnet with an associated Elastic IP to allow private-subnet nodes to initiate outbound connections for container image pulls and OS patches.

The Terraform deliverables consist of two files: main.tf and variables.tf. variables.tf declares all parameterizable values that vary between deployments, keeping sensitive or environment-specific data out of the shared main.tf. Key variables include admin_cidrs (trusted administrator IP CIDRs for SSH and API server access), cluster_node_cidr (private subnet CIDR of all cluster nodes), vpc_cidr (the full VPC CIDR block), and ami_id, instance_type, key_name (EC2 configuration values that may differ per region).

main.tf declares all infrastructure resources in the recommended order of dependency, covering: VPC and subnets (aws_vpc, aws_subnet, aws_internet_gateway, aws_eip + aws_nat_gateway, aws_route_table); Security Groups (one resource per node role: k8s-control-plane-sg, k8s-worker-sg, jenkins-sg, gateway-sg); and Compute instances for each node role with their respective user_data scripts. The outputs block exports the node private IPs for use in cluster bootstrap scripts.

4.3.4 Database Design

The system utilizes two primary database environments for the Sign-in API and the HRM API. The Sign-in database is a separate database that stores users, tokens, and login audit records for the Sign-in API, while the HRM database consists of a PostgreSQL cluster containing a master database, managed by MasterDbContext, which stores the tenant registry and is accessed by the HRM API's provisioning service. Additionally, within this cluster, each provisioned tenant has a dedicated per-tenant database managed by AppDbContext to store specific tenant data. Database access is restricted to application pods within the cluster. The EF Core TenantMiddleware resolves the correct per-tenant connection string from the JWT claim on every request, and no database port is exposed outside the cluster boundary. Network Policies enforce that only pods in the

hummingbird-api namespace may initiate connections to the database StatefulSets, preventing any cross-tenant database access at the network layer regardless of application-level configuration.

4.3.4.1 HRM API Database Schema (AppDbContext, per-tenant)

Each tenant database is provisioned by running EF Core migrations against a new PostgreSQL database created during tenant onboarding. The schema consists of five tables across two concerns: HR data (Positions, Employees, TimeAttendanceRecords, PayrollRecords) and per-tenant configuration (AppConfigs).

Table 4.2: HRM API Database Schema — Positions Table

Column	Type	Notes
Id	integer	PK, identity
Name	text	Job title
DefaultSalary	numeric(18,2)	Base salary for this role
ShiftType	text	Enum: Morning | Afternoon | Evening
ClockInTime	interval	Expected shift start time
ClockOutTime	interval	Expected shift end time
TotalHours	numeric(5,2)	Expected daily hours
ServiceChargePercentage	numeric(5,2)	SC allocation share for Version A payroll

Table 4.3: HRM API Database Schema — Employees Table

Column	Type	Notes
Id	integer	PK, identity
EmployeeCode	text	Unique identifier; used to match CSV import records
Name	text	First name
Surname	text	Last name
PositionId	integer	FK → Positions.Id (ON DELETE RESTRICT)
Salary	numeric(18,2)	Actual salary; may override position default
DateJoined	timestamptz	Employment start date (UTC)
Status	text	Enum: Active | Terminated; default Active

The TimeAttendanceRecords table is denormalized and linked to the Employees table via EmployeeCode string rather than a foreign key to enable CSV imports to be stored independently of the Employees table.

Table 4.4: HRM API Database Schema — TimeAttendanceRecords Table

Column	Type	Notes
Id	integer	PK, identity
EmployeeCode	text	Loose coupling to Employees.EmployeeCode
EmployeeName	text	Denormalized name from CSV
Department	text	Denormalized department from CSV
Date	timestamptz	Attendance date (UTC)
ShiftType	text	Shift classification
ClockIn	interval	Actual clock-in time (nullable)
ClockOut	interval	Actual clock-out time (nullable)
TotalHours	numeric(5,2)	Calculated hours worked
IsMissing	boolean	true = no-show; default false
IsPenalty	boolean	true = late arrival >30 min; default false
Month	integer	Calendar month (1–12)
Year	integer	Calendar year

Table 4.5: HRM API Database Schema — PayrollRecords Table

Column	Type	Notes
Id	integer	PK, identity
EmployeeId	integer	FK → Employees.Id (ON DELETE CASCADE)
Month	integer	1–12
Year	integer	Calendar year
BaseSalary	numeric(18,2)	Gross base salary
Deductions	numeric(18,2)	Late arrival + no-show penalties
ServiceChargeBonus	numeric(18,2)	Service charge distribution amount

Column	Type	Notes
NetSalary	numeric(18,2)	BaseSalary − Deductions + ServiceChargeBonus
TotalScheduledDays	integer	Expected working days in the month
WorkDays	integer	Actual days attended
PenaltyDays	integer	Days with late arrival penalty
MissingDays	integer	No-show days
ServiceChargeInput	decimal	Total SC revenue input for the month
ServiceChargeVersion	text	'A' = fixed rate per position; 'B' = workday-weighted
CalculatedAt	timestamptz	Timestamp when record was computed (UTC)

The AppConfigs table is a key-value configuration store that holds per-tenant runtime settings, seeded on provisioning with three default entries: ServiceChargeVersion ('A'), CompanyName ('My Hotel'), and Onboarded ('false').

Table 4.6: HRM API Database Schema — AppConfigs Table

Column	Type	Notes
Id	integer	PK, identity
Key	text	Config key (unique index)
Value	text	Config value
Description	text	Human-readable description
UpdatedAt	timestamptz	Last update timestamp (UTC)

4.3.4.2 HRM API Master Database Schema (MasterDbContext, shared)

The master database is shared across all tenants and holds the tenant registry used by the TenantMiddleware to resolve each incoming request to the correct per-tenant database connection string. It is accessed only by the HRM API's provisioning and routing logic and is never exposed to tenant-level application code.

Table 4.7: HRM API Master Database Schema — Tenants Table

Column	Type	Notes
Id	integer	PK, identity
Name	text	Display name of the hotel tenant
Subdomain	text	URL subdomain (unique); used for Traefik IngressRoute routing
DatabaseName	text	PostgreSQL database name (unique)
ConnectionString	text	Full database connection string for this tenant
IsActive	boolean	Whether the tenant is currently active; default true
CreatedAt	timestamptz	Provisioning timestamp (UTC)

4.3.4.3 Sign-in API Database Schema

The Sign-in API uses a separate PostgreSQL database, independent of the EF Core contexts used by the HRM API. It manages user authentication state, token lifecycle, and login audit trails across all tenants. The schema consists of four tables: tenants, users, tokens, and login_audit.

The tenants table is a mirror of the tenant registry within the sign-in database and is used to scope user accounts to their correct tenant and to provide the frontend_url redirect target after successful authentication.

Table 4.8: Sign-in API Database Schema — tenants Table

Column	Type	Notes
id	serial	PK
slug	varchar(50)	URL-safe tenant identifier (unique); e.g. 'hotel-grand'
name	varchar(150)	Display name; e.g. 'Hotel Grand Bangkok'
frontend_url	varchar(500)	Redirect target after login; e.g. 'https://grand.hmmbird.xyz'
is_active	boolean	Whether the tenant is active; default true
created_at	timestamptz	Creation timestamp; default NOW()
updated_at	timestamptz	Last update timestamp; auto-updated by trigger

The user table stores all user accounts for all tenants. Email and username uniqueness is scoped per tenant, and the same email address may exist in different tenant accounts. Accounts support lockout via failed_login_count and locked_until, reset automatically on successful login by a database trigger.

Table 4.9: Sign-in API Database Schema — users Table

Column	Type	Notes
id	serial	PK
tenant_id	int	FK → tenants.id (ON DELETE RESTRICT)
email	varchar(150)	Unique per tenant (composite unique constraint)
username	varchar(50)	Unique per tenant (composite unique constraint)
password_hash	text	Bcrypt hash via pgcrypto
role	enum	Enum: superadmin | hr_manager | hr_staff | manager | employee
first_name / last_name	varchar(100)	Optional display name fields
is_active	boolean	Account enabled flag; default true
is_email_verified	boolean	Email verification flag; default false
failed_login_count	smallint	Incremented on failed attempts; reset on success
locked_until	timestamptz	Account lockout expiry (nullable)
last_login_at	timestamptz	Timestamp of most recent successful login
created_at / updated_at	timestamptz	Auto-managed timestamps

The tokens table manages the lifecycle of all non-password tokens issued to users: JWT refresh tokens, password reset tokens, and email verification tokens. Each token is stored as a hash to prevent exposure in the event of a database compromise.

Table 4.10: Sign-in API Database Schema — tokens Table

Column	Type	Notes
id	serial	PK
user_id	int	FK → users.id (ON DELETE CASCADE)
token_type	enum	Enum: refresh | password_reset | email_verify
token_hash	text	Hash of the issued token (unique)
expires_at	timestamptz	Token expiry time

Column	Type	Notes
used_at	timestamptz	When the token was consumed (nullable)
revoked_at	timestamptz	When the token was explicitly revoked (nullable)
ip_address	inet	Client IP at time of issue
user_agent	text	Client user-agent string
created_at	timestamptz	Issue timestamp

The login_audit table is an append-only audit log of all login attempts, both successful and failed. Tenant and user foreign keys are nullable (SET NULL on delete) to preserve the audit trail even if a tenant or user is removed.

Table 4.11: Sign-in API Database Schema — login_audit Table

Column	Type	Notes
id	bigserial	PK (large integer for high-volume append)
tenant_id	int	FK → tenants.id (ON DELETE SET NULL; nullable)
user_id	int	FK → users.id (ON DELETE SET NULL; nullable)
email_attempt	varchar(150)	Email submitted in the login attempt
result	enum	Enum: success | bad_password | not_found | inactive | locked
ip_address	inet	Client IP address
user_agent	text	Client user-agent string
created_at	timestamptz	Event timestamp

4.3.5 Network Infrastructure Design

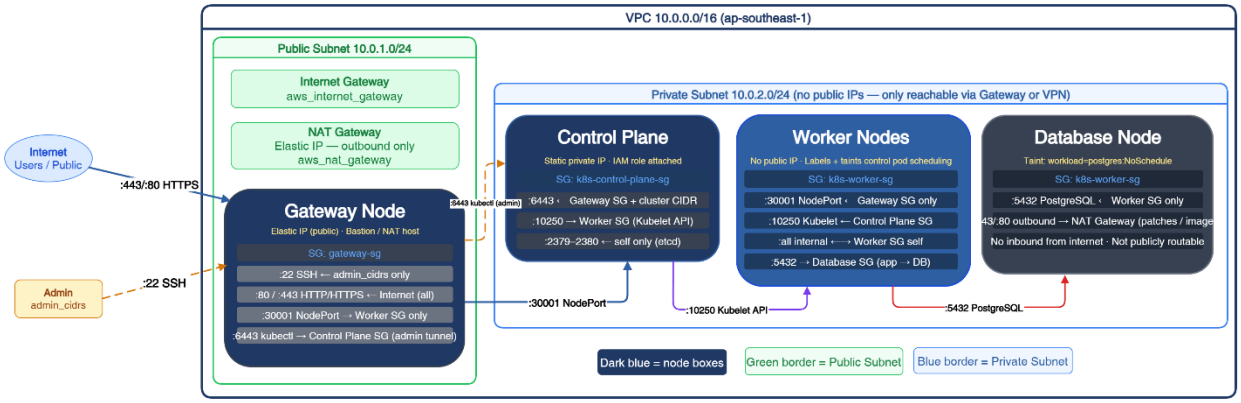


Figure 4.6: Network Infrastructure — Subnet, Security Group, and Port Layout

The VPC uses a /16 CIDR block (10.0.0.0/16), providing 65,534 usable addresses including headroom for future node groups and availability zone expansion without renumbering. DNS Hostnames and DNS Support are explicitly enabled on the VPC resource, as Kubernetes relies on internal DNS for all service discovery between pods.

This project intentionally uses a single Availability Zone to keep networking straightforward and to maintain a focused scope, as multi-AZ high availability falls outside the goals of this work. The subnet strategy follows a strict public/private split: public subnets are reserved for the Gateway node and the Internet Gateway only; no application workload or database is ever placed in a public subnet. Private subnets host the Control Plane and all Worker Nodes; these instances are not assigned public IP addresses and are only reachable through the Gateway node or a VPN. A NAT Gateway is assigned an Elastic IP so that private-subnet nodes can initiate outbound connections for pulling container images or applying OS security patches, without being reachable from the internet.

Role-specific Security Groups scope traffic to Control Plane, Workers, PostgreSQL, and Gateway Node using narrow CIDRs instead of open access. The following tables show the full policies for each Security Group in the cluster: gateway-sg, k8s-control-plane-sg, and k8s-worker-sg.

Table 4.12: Security Group Rules — gateway-sg

Port	Protocol	Source	Reason
22	TCP	var.ssh_cidrs	Allow only admin to access
10250	TCP	var.admin_cidrs	Kubelet API
4789	UDP	10.0.2.0/24	Calico VXLAN overlay from worker subnet

Port	Protocol	Source	Reason
80	TCP	0.0.0.0/0	HTTP traffic into Traefik / public services
443	TCP	0.0.0.0/0	HTTPS traffic into Traefik / public services
egress	all	0.0.0.0/0	Standard outbound

Table 4.13: Security Group Rules — *k8s-control-plane-sg*

Port	Protocol	Source	Reason
all	all	k8s-worker-sg	Worker node interaction
6443	TCP	var.admin_cidrs	kube-apiserver from admin IPs (kubectl from outside)
2379–2380	TCP	self	etcd client + peer
4789	UDP+TCP	self	Calico VXLAN overlay
179	TCP	self	Calico BGP peering
0	IPIP	self	Calico IPIP overlay
5473	TCP	self	Calico Typha
9091	TCP	self	Calico Felix metrics
9093	TCP	self	Calico Typha metrics
9094	TCP	self	Calico Typha health
8080	TCP	self	Calico Felix health
9099	TCP	self	Calico Felix readiness/liveness
egress	all	0.0.0.0/0	Standard outbound

Table 4.14: Security Group Rules — k8s-worker-sg

Port	Protocol	Source	Reason
all	all	k8s-control-plane-sg	Control plane interaction
all	all	self	Inter-worker pod traffic
4789	UDP+TCP	self	Calico VXLAN overlay

Port	Protocol	Source	Reason
179	TCP	self	Calico BGP peering
0	IPIP	self	Calico IPIP overlay
5473	TCP	self	Calico Typha
9091	TCP	self	Calico Felix metrics
9093	TCP	self	Calico Typha metrics
9094	TCP	self	Calico Typha health
8080	TCP	self	Calico Felix health
9099	TCP	self	Calico Felix readiness/liveness
egress	all	0.0.0.0/0	Standard outbound

The Gateway node is the single public entry point for all user traffic into the cluster. DNS A records for hmmbird.xyz and *.hmmbird.xyz resolve to the Gateway node's Elastic IP, so all tenant subdomains resolve to the same host. The client follows four steps: (1) the browser resolves the tenant subdomain and sends an HTTP/HTTPS request to the Gateway node; (2) the Gateway forwards the packet via NodePort 30001 to the Traefik service on the Worker Nodes; (3) Traefik inspects the HTTP Host header, matches it against its IngressRoute rules, and forwards the request to the ClusterIP service of the corresponding tenant frontend pod; (4) the pod processes the request and the response travels back through Traefik, the Gateway, and the internet to the client. Worker Nodes never receive traffic directly from outside the cluster as the Worker Security Group permits NodePort traffic only from the Gateway Security Group.

4.3.6 Monitoring and Observability

The observability stack is deployed in a dedicated monitoring namespace consisting of four components: Prometheus for metrics collection, Grafana for visualization, Node Exporter for host-level metrics, and kube-state-metrics for Kubernetes object state metrics.

Prometheus runs as a single-replica Deployment on port 9090. Grafana runs as a single-replica Deployment on port 3000 and is exposed externally via a Traefik IngressRoute at grafana.hmmbird.xyz. Node Exporter runs as a DaemonSet across all six nodes, exposing host-level CPU, memory, disk, and network metrics on port 9100. kube-state-metrics exposes Kubernetes object state metrics including Deployment replica counts, HPA scaling events, StatefulSet status, PVC binding state, and pod lifecycle events.

Table 4.15: Prometheus Scrape Jobs

Job	Method	Notes
prometheus	Static — localhost:9090	Self-scrape
node-exporter	Static — node-exporter:9100	All 6 nodes via DaemonSet
kube-state-metrics	Static — kube-state-metrics:8080	Dedicated job; annotation scrape disabled
kubernetes-pods	Kubernetes SD (role: pod)	Opt-in via prometheus.io/scrape: "true" annotation
kubernetes-services	Kubernetes SD (role: service)	Opt-in via prometheus.io/scrape: "true" annotation
kubernetes-nodes	Kubernetes SD (role: node)	Direct kubelet scrape; requires nodes/metrics subresource RBAC
kubernetes-cadvisor	Kubernetes SD (role: node)	Proxied via API server
kubernetes-apiservers	Kubernetes SD (role: endpoints)	kube-apiserver metrics

4.3.7 Costs

For a single hotel managing approximately 3,000 employees, the capital expenditure for an on-premises payroll system typically includes dedicated server hardware, high-availability storage, and licensing for both the operating system and the database management system. On average, the initial investment for such a setup can range from $15,000 to $25,000 per hotel. When expanded to a 5-hotel chain, these costs multiply, requiring a total upfront investment of $75,000 to $125,000 for the infrastructure alone.

The Hummingbird architecture operates on a cost-effective, cloud-native model utilizing AWS ap-southeast-1 (Singapore) regional pricing for t3 instances. The infrastructure is configured to balance high availability with economic efficiency, the total monthly cost for the initial infrastructure configuration is summarized in the following table.

Table 4.16: Cluster Node Cost Breakdown — Initial State

Node	Instance Type	Quantity	Monthly Cost
Control Plane	t3.medium	1	$36.21
Worker Node	t3.medium	4	$144.84 ($36.21 each)
Database Node	t3.xlarge	1	$60.74

Node	Instance Type	Quantity	Monthly Cost
Gateway Node	t3.small	1	$18.10
NAT Gateway	AWS Managed	1	~$32.00
		Total Cluster Cost	$291.89

This total monthly expenditure of $291.89 successfully satisfies the ideal infrastructure cost target of less than $300 defined in the project requirements.

While the initial state maintains a steady footprint, the architecture is designed to scale horizontally. In a fully scaled state, the worker node pool expands to handle the increased computational demand. The following table details the costs associated with the cluster operating at its maximum defined capacity.

Table 4.17: Cluster Node Cost Breakdown — Full Scale

Node	Instance Type	Quantity	Monthly Cost
Control Plane	t3.medium	1	$36.21
Worker Node	t3.medium	21	$760.41 ($36.21 each)
Database Node	t3.xlarge	1	$60.74
Gateway Node	t3.small	1	$18.10
NAT Gateway	AWS Managed	1	~$32.00
		Total Cluster Cost	$907.46

4.4 Application Design

4.4.1 Sign-in Service

The sign-in module is a React frontend with an ASP.NET Core Web API backend deployed in the hummingbird-signin namespace. Users are able to visit hmmbird.xyz and authenticate. The browser redirects the user to the correct tenant subdomain (e.g. acme.hmmbird.xyz) where a token is used to avoid a second login.

4.4.2 Hummingbird HRM API

The Human Resource Management (HRM) API is an ASP.NET Core modular monolith with three modules and an admin endpoint. A TenantMiddleware component extracts the tenant ID from the JWT, resolves the PostgreSQL connection string for that tenant, and switches the EF Core database context on every request. The Payroll Module supports two versions: Version A

(fixed service charge rate per position) and Version B (workday-weighted distribution). The Personnel Module manages the full employee lifecycle. The Time Attendance Module parses CSV exports from physical attendance machines and applies penalty rules for late arrivals and absences.

4.4.3 Hummingbird Admin

The administrator module allows the SaaS provider to manage the creation of new tenants, edit existing tenants, and change the Payroll Module mode between Version A and Version B as a demonstration on the per-tenant customizability of the frontend and business logic.

4.4.4 Automated Tenant Provisioning

The TenantProvisioningService uses the .NET Kubernetes client library to execute four API calls when a new tenant is created: create namespace, create frontend Deployment and Service, create Traefik IngressRoute for the tenant subdomain, and create a Kubernetes Secret with the database connection string. EF Core migrations are then run against the new tenant database.

5 EXPERIMENTAL SETUP AND METHODS

5.1 Experimental Setup

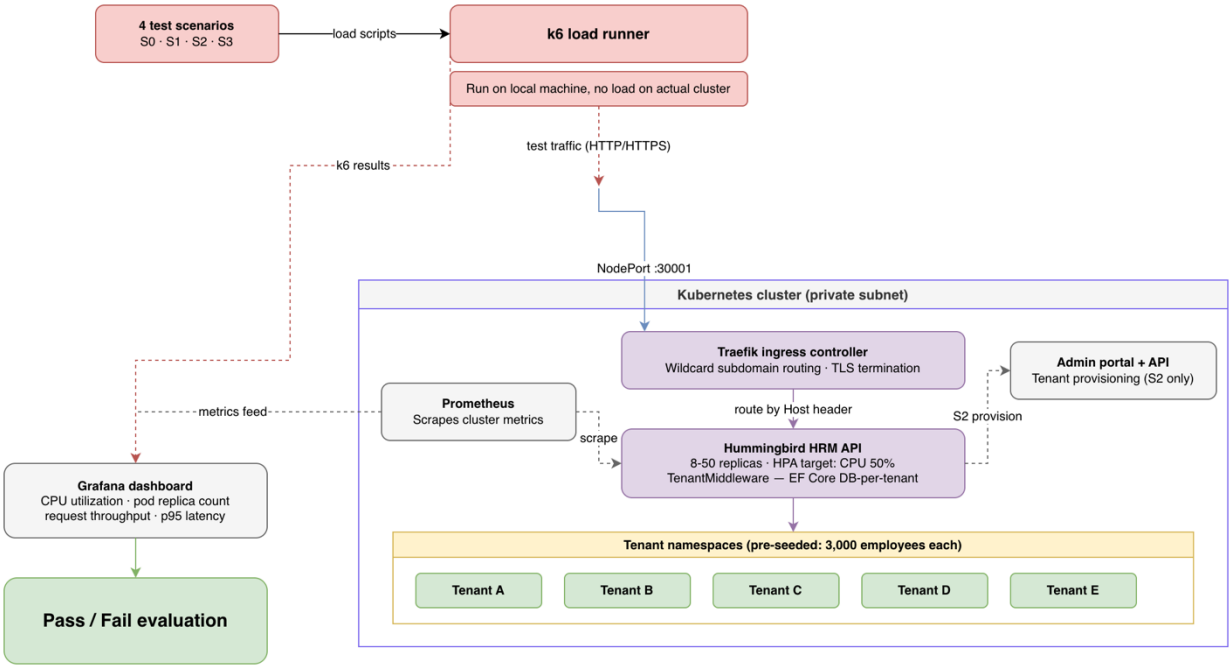


Figure 5.1: Experimental Testing Setup and Scenarios

5.1.1 Load Test Runner

Load testing is performed using Grafana k6, which is an open-source load testing tool that executes JavaScript test scripts that generate configurable HTTP traffic. k6 is run on a local machine outside of the cluster to ensure that the test execution computation does not interfere with the system's performance and mimic real traffic as much as possible. Test traffic is sent over the internet to the Gateway Node's public Elastic IP address, which enters the cluster just like real user traffic: Gateway → NodePort 30001 → Traefik → tenant namespace → HRM API → PostgreSQL. Running k6 externally means the load test generation consumes no cluster resources, so CPU and memory measurements on the Worker Nodes reflect only the application workload under test.

Each test script is structured around k6's options object, which defines the virtual user (VU) count and ramp-up/ramp-down stages via the stages array. All scenarios use k6's thresholds block to specify pass/fail Service Level Objectives (SLOs) for error rate. The shared authentication library (k6/lib/auth.js) submits credentials using the emailOrUsername field against the sign-in API. Application routes follow the unversioned API contract: /api/employees, /api/payroll/calculate, and /api/attendance. The employee creation payload uses the schema-aligned fields name, surname, employeeCode, positionId, salary, dateJoined, and status. All scenarios are executed via a run.sh wrapper script which configures Prometheus remote write output by setting the Server URL, Trend Stats, and Stale Markers environment variables, enabling

k6 throughput and latency metrics to be pushed into the same Prometheus time-series database as the cluster metrics and viewed in Grafana alongside node CPU and HPA events.

5.1.2 Tenant and Employee Test Data Seeding

Before any test scenario is executed, five tenant namespaces (tenant-a through tenant-e) are provisioned via the admin API. Each tenant's PostgreSQL database is then seeded with 3,000 employee records, totaling to 15,000 employees across all tenants. The seed data covers all four Positions defined in the schema (Food and Beverage, Cleaner, Receptionist, Manager), with employees distributed proportionally across positions to produce representative payroll calculation workloads. Time attendance records are also pre-populated for the current and previous months so that the payroll module has complete data to process during load scenarios.

5.1.3 Cluster Monitoring

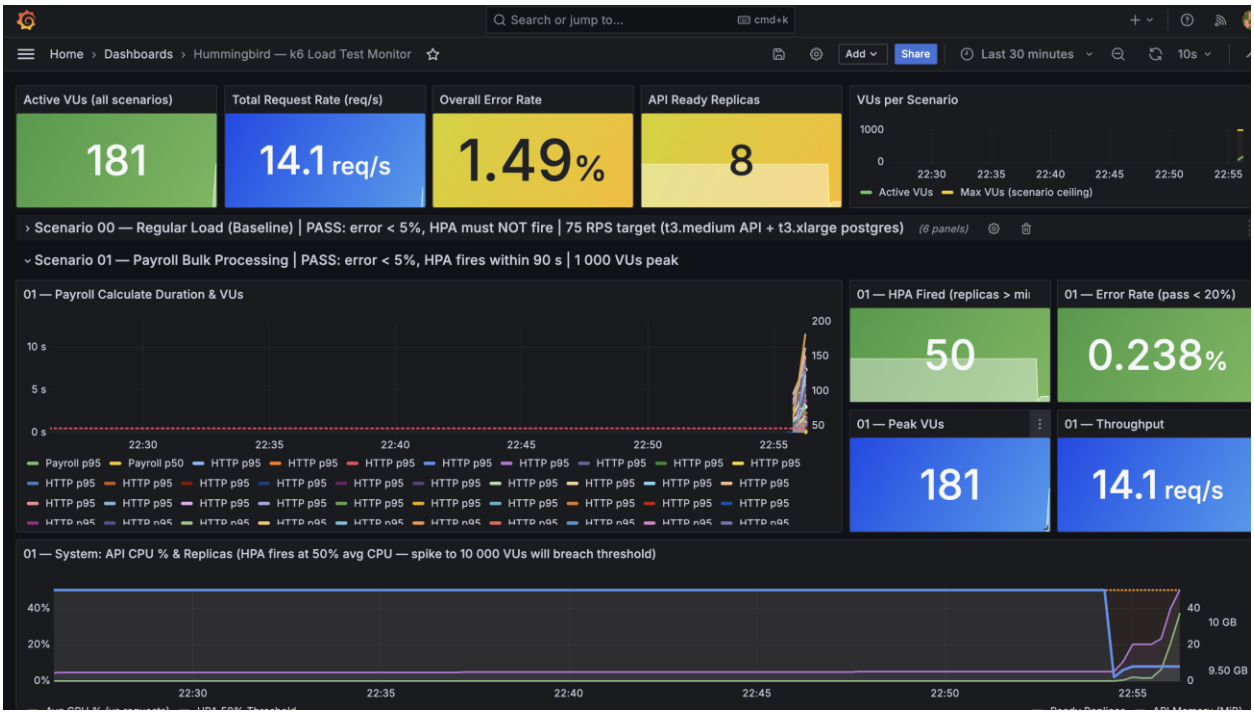


Figure 5.2: Kubernetes Scaling Test Grafana Dashboard

All test runs are monitored in real time using Prometheus and Grafana dashboard stack described in Section 4.3.6. Prometheus scrapes metrics from all six nodes and from the k6 process itself via the k6 Prometheus remote write output, which pushes throughput, p95 latency, error rate, and VU count metrics into the same time series database as the cluster metrics. The Grafana dashboard displays these side-by-side: node CPU and memory utilization per node, HPA replica count over time (to observe scale-up and scale-down events), database connection pool usage per tenant, and k6 request throughput and p95 latency. Each scenario is tagged so that its metrics window can be isolated in Grafana by filtering on the scenario label, allowing direct comparison against the pass criteria SLOs defined in Section 5.2.2.

5.2 Experimental Methods

k6 allows for flexible configuration of test scenarios through the options object, making it easy to define ramp-up and ramp-down stages to mimic user behavior. All four scenarios use the stages configuration in k6's options to define load profiles, with thresholds specifying the pass/fail criteria as Service Level Objectives (SLOs) that the test must satisfy. The testing philosophy for all scenarios is infrastructure verification on the current test cluster hardware rather than application-level performance benchmarking. The Worker Nodes are t3.medium burstable instances with a 20% baseline CPU allocation, and the payroll API responses are approximately 450 KB in size. Pass criteria therefore focus on platform availability (error rate) and scaling behavior (HPA and Cluster Autoscaler events) rather than absolute latency, which is deferred to production-class hardware.

The Grafana dashboard provides the primary operational view during all test scenarios, covering node CPU and memory breakdown, pod restart counts, HPA replica scaling events, and k6 throughput and p95 latency metrics in real time.

5.2.1 Feature Testing

Feature testing validates the core operational capabilities of the infrastructure beyond just load handling. Three feature areas are verified:

- Auto-scaling response time: measured from the moment CPU utilization exceeds 50% or memory utilization exceeds 80% HPA threshold to the point at which the first new pod reaches the Running state. The expected response is within 60 seconds under normal cluster conditions.

- Tenant onboarding time: measured from the admin API call that initiates provisioning to the moment the new tenant's frontend pod is reachable via its Traefik IngressRoute. The target is under 5 minutes for the ideal case and under 10 minutes for the marginal case.

- CI/CD pipeline end-to-end time: measured from a code commit being pushed to GitHub to the updated image being live in the Kubernetes cluster via ArgoCD synchronization. This validates the canary deployment path as well as the full Jenkins build pipeline.

5.2.2 Load Scenarios

The following load scenarios simulate real-world situations that the system would experience derived from the hotel HR usual operations calendar.

5.2.2.1 Scenario S0 — Steady-State Normal Usage

Every other scenario validates the system's behavior under exceptional conditions. This test addresses the question that precedes all other test scenarios: does the system perform acceptably during an ordinary working day? A system that passes peak-load tests but exhibits unacceptable latency under moderate routine traffic would be unsuitable for production deployment regardless of its stress-test credentials.

This scenario establishes that the minimum replica configuration is correctly sized for normal daily HR traffic. It is the precondition all other scenarios depend on: if the system cannot sustain routine traffic without triggering HPA, the spike scenarios cannot be interpreted meaningfully. No HPA scale-up event should occur during this run — the API replica count must remain at 8 throughout.

The traffic mix is GET-only, weighted randomly per iteration across four endpoints: 40% list all employees (GET /api/personnel/employees), 25% view single employee (GET /api/personnel/employees/{id}), 20% fetch payroll for a month (GET /api/payroll/{year}/{month}), and 15% attendance summary (GET /api/timeattendance/{year}/{month}/summary). The executor is ramping-arrival-rate: 1 minute ramp to 75 RPS, 3 minutes steady state, 1 minute ramp down. 120 VUs are pre-allocated with a ceiling of 250.

Pass Criteria

- baseline_errors (5xx + network failures): < 5%
- http_req_failed: < 5%
- API replica count: 8 throughout (HPA must not fire)

5.2.2.2 Scenario S1 — New Tenant Onboarding

This scenario tests a capability that is qualitatively different from the others: it exercises the infrastructure provisioning path rather than the application request-serving path. When a SaaS provider acquires new hotel clients, which may happen in clusters when a hotel chain signs up multiple properties simultaneously, the automated provisioning service must create Kubernetes namespaces, deploy frontend applications, configure Traefik routing, and initialize PostgreSQL databases in parallel for multiple new tenants. A provisioning failure or excessive delay in this path directly impairs the commercial onboarding experience: a new hotel client who cannot access their tenant environment within the expected window will lose confidence in the platform before their first employee record has been created.

This scenario tests the admin API's ability to provision new hotel tenants sequentially. It uses a single VU executing 5 iterations, one per hotel, for a total runtime of approximately 5–8 minutes. Each iteration onboards one of the five test hotel datasets in three sequential steps. In Step 1 (Provision), the VU calls POST /admin/tenants to create the tenant, registers the tenant in the sign-in database, registers an admin user, and probes the Traefik IngressRoute until it returns a live response, confirming the tenant environment is reachable. In Step 2 (Append Employees), the VU uploads a multipart CSV from demo-data/dataset_1 via the time attendance import endpoint, seeding 3,000 employees per hotel. In Step 3 (Append Attendance), the VU uploads a second multipart CSV containing approximately 13,700–14,000 attendance rows per hotel.

All requests are tagged with tenant={slug} to enable per-tenant breakdown in Grafana. Five custom duration metrics are recorded: tenant_provision_duration (Step 1), tenant_employees_duration (Step 2), tenant_attendance_duration (Step 3), tenant_append_duration (Steps 2 + 3 combined), and tenant_completion_duration (full onboarding end-to-end.

Pass Criteria

- Error rate: < 5%
- All 5 tenants complete provisioning and both CSV data appends without failure

5.2.2.3 Scenario S2 — Bulk Month-End Payroll Processing

Month-end payroll processing represents the single most computationally intensive recurring event in a hotel HR management system's operational calendar. Unlike the distributed, low-intensity traffic of normal usage, this event is coordinated and time-constrained: hotel HR administrators across all tenants initiate bulk payroll calculations during the same narrow window at the end of each calendar month.

This scenario tests the HPA's ability to detect CPU pressure from sustained payroll traffic and scale the API deployment within the 60-second stabilization window. Payroll calculation is the most CPU-intensive operation as it reads all employee records, calculates net salary, and bulk-inserts results. A sustained spike of concurrent payroll requests should push average pod CPU above the 50% HPA threshold and trigger a scale-up event that brings replica count above 8.

The traffic mix is 80% GET /api/payroll/{year}/{month} (drives CPU load through large 450 KB responses) and 20% POST /api/payroll/calculate (write-heavy DB operation, spread across 24 historical periods from 2023–2024 to limit lock contention). The executor is ramping-vus: 2 minutes ramp to 500 VUs, 30-second spike to 600 VUs, 5-minute hold, 2-minute ramp down.

Pass Criteria

- payroll_errors: < 5%
- http_req_failed: < 5%
- HPA replica count: increases above 8 within 90 seconds of the hold phase start

5.2.2.4 Scenario S3 — Seasonal Staff Onboarding and Offboarding

Thailand's tourism economy is strongly seasonal, with international arrivals concentrated in the November–April peak season and domestic tourism spiking around Thai New Year and school holiday periods. Hospitality HR software must handle the complex logistics of seasonal demand, including the need to adapt new staff roster at short notice, planning for a variety of roles, and managing a high turnover of staff. For the system, peak season onset and offset are represented as sudden, high-volume create and delete operations on the employee personnel module as hotel operators hire large cohorts of seasonal staff at the start of the season and remove them when it ends.

This scenario simulates the full lifecycle of seasonal hotel staff across a peak tourism window: rapid bulk hiring, sustained operational access during the season, and systematic offboarding at the end. It is structured into three explicit phases with different executors and load profiles, reflecting the qualitatively different nature of each activity.

- Phase 1 — Load (0–1 min): A constant-arrival-rate executor fires at 50 requests per second to POST /api/employees, creating temporary employees with SEAS-* employee codes. This phase simulates a hotel chain rapidly hiring seasonal staff at the start of peak season.
- Phase 2 — Access (1–4 min): A constant-arrival-rate executor fires at 200 requests per second across a read-heavy GET mix: employee directory, individual employee profile, payroll records, attendance summary, and position listings. This phase simulates the sustained operational load during the season itself — HR staff and employees accessing the system daily across all five tenants.
- Phase 3 — Offload (4 min+): A per-vu-iterations executor runs 5 iterations × 1 VU, sequentially listing all SEAS-* employees per tenant and deleting each one. This phase simulates the end-of-season offboarding process, cleaning up the seasonal workforce in a controlled, sequential manner rather than a concurrent burst.

Four custom metrics are recorded: staff_load_duration (Phase 1 POST response times), staff_access_duration (Phase 2 GET response times), staff_offload_duration (Phase 3 DELETE response times), and staff_errors (failures across all phases).

Pass Criteria

- staff_errors: < 10%
- http_req_failed: < 10%
- All three phases complete without exceeding the error threshold

5.2.3 Cross-Scenario Infrastructure Metrics (Grafana)

All scenarios emit the following system-level metrics to the Grafana dashboard via Prometheus, alongside the scenario-specific k6 custom metrics. These panels are monitored during every run as the primary indicators of infrastructure behavior.

Table 5.1: Grafana Infrastructure Monitoring Panels

Job	Method	Notes
API CPU % vs requests	container_cpu_usage / kube_pod_resource_requests × 100	Shows when the HPA trigger is breached
API ready replicas	kube_deployment_status_replicas_ready	Confirms HPA fired and pods are Ready
HPA desired vs current	kube_horizontalpodautoscaler_status_*_replicas	Scale-up timeline
Node count	count(kube_node_info)	Cluster Autoscaler activity
Pod restarts	increase(kube_pod_container_status_restarts_total[5m])	Stability signal under load

6 RESULTS AND DISCUSSION

6.1 Results

This section presents the outcomes and test results of all tests executed against the deployed Hummingbird system cluster. This section includes the three feature test results and the four load test scenarios. Each section contains a pass/fail summary table of the measured value against the pass criteria defined in Section 5.2, followed by the full k6 metric output for reference.

6.1.1 Feature Tests

Feature tests were done to verify the three important operational capabilities of the infrastructure: auto-scaling response time, tenant onboarding time, and CI/CD pipeline end-to-end time. Each test was measured under conditions as described in Section 5.2.1.

Table 6.1: Feature Test Results

Metric	Target	Measured Results	Pass/Fail
HPA scale-up response time	< 60 s from CPU > 50% / memory > 80% to new pod Running	15 seconds	Pass
Tenant onboarding time	< 10 min (marginal) / < 5 min (ideal) from API call to live tenant	11.47 seconds From Load Scenario S1	Pass
CI/CD pipeline end-to-end time	Commit to live image in cluster via ArgoCD	4:16 minutes on average	Pass

For the CI/CD pipeline timing test, the time began at first push to GitHub and ended when ArgoCD successfully synced the new image to the cluster.

Table 6.2: CI/CD Pipeline End-to-End Full Runs

Run No.	App Component	Jenkins Build Complete	ArgoCD Sync Complete	Image Deploy Complete	Total Time
1	signin-frontend	4:56 m	28 s	23 s	5:47 m
2	admin	1:11 m	46 s	16 s	2:13 m
3	api	2:27 m	1:41 m	40 s	4:48 m
	Average	2:51 m	58 s	26 s	4:16 m

6.1.2 Load Scenario Tests

6.1.2.1 Scenario S0 — Steady-State Normal Usage

Constant-arrival-rate test targeting 75 requests per second for 5 minutes across all five tenants. Exercises login, attendance summary, payroll fetch, list employees, and employee detail endpoints.

Table 6.3: Scenario S0 — Steady-State Normal Usage Test Results

Metric	Target	Measured Results	Pass/Fail
baseline_errors	< 5%	0%	Pass
http_req_failed	< 5%	0%	Pass
API replica count	8	8	Pass

Table 6.4: Scenario S0 — Steady-State Normal Usage Full k6 Metrics

Metric	Average	p90	p95
baseline_attendance_duration	1.34 s	2.73 s	3.57 s
baseline_detail_duration	233.29 ms	488.73 ms	699.54 ms
baseline_error_breakdown	2297 (7.645038/s)		
baseline_errors	0% (0 out of 14856)		
baseline_list_duration	4.02 s	7.51 s	9.03 s
baseline_payroll_duration	7.91 s	13.57 s	16.26 s
checks (login)	100% (14861 correct)		
http_req_duration	3.43 s	8.94 s	11.49 s
http_req_failed	0% (0 out of 14861)		
http_req_receiving	3.2 s	8.64 s	11.2 s
http_req_sending	37.05 µs	41 µs	53 µs
http_reqs	14861 (49.461434 req/s)		

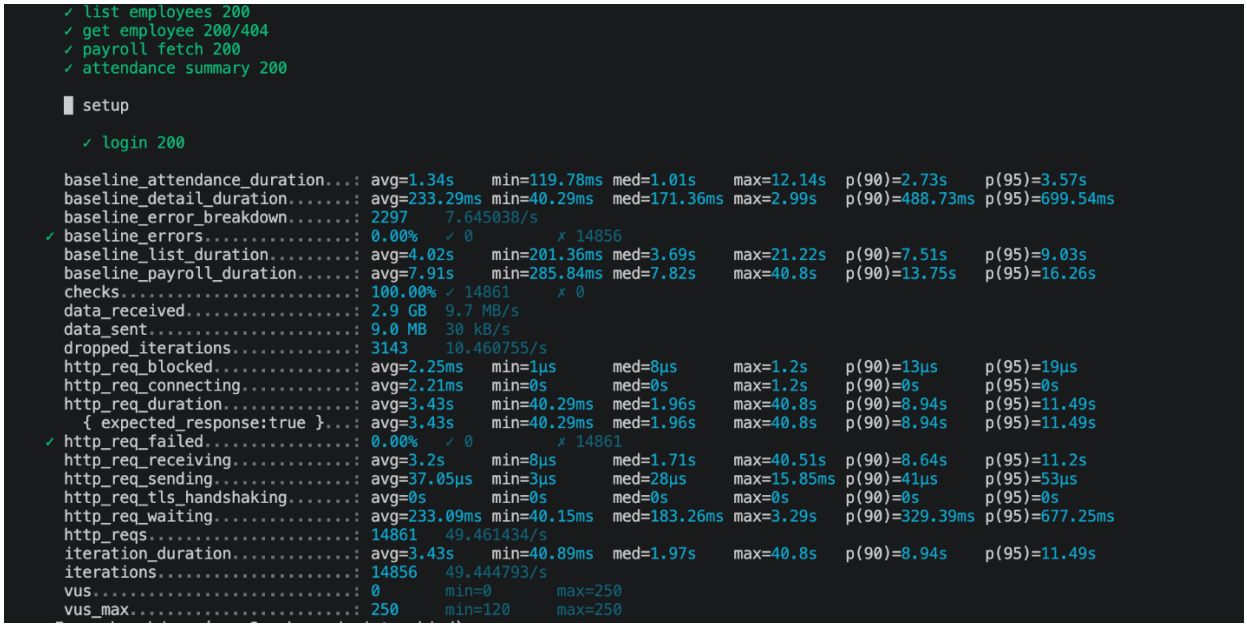


Figure 6.1: Scenario S0 — Steady-State Normal Usage Full k6 Metrics

6.1.2.2 Scenario S1 — New Tenant Onboarding

Sequential 1 VU × 5 iterations, one per hotel. Each iteration: provision via POST /admin/tenants + sign-in registration + ingress probe, then CSV employee upload (3,000 records), then CSV attendance upload (~13,700–14,000 rows). Total runtime ~5–8 minutes.

Table 6.5: Scenario S1 — Onboarding Test Results

Metric	Target	Measured Results	Pass/Fail
Error rate	< 5%	0%	Pass
All 5 tenants: provisioning + employee CSV + attendance CSV complete	Pass	All onboarding steps successfully run	Pass

Table 6.6: Scenario S1 — Onboarding Test Full k6 Metrics

Metric	Average	p90	p95
tenant_provision_duration	5.7 s	6.44 s	6.7 s
tenant_employees_duration	1.41 s	2.05 s	2.28 s
tenant_attendance_duration	4.36 s	5.25 s	5.25 s
tenant_append_duration (employees + attendance)	5.77 s	6.47 s	6.53 s
tenant_completion_duration (full onboarding)	11.47 s	11.83 s	11.89 s
Error rate (http_req_failed)	0%		
Total runtime	57.5 s		

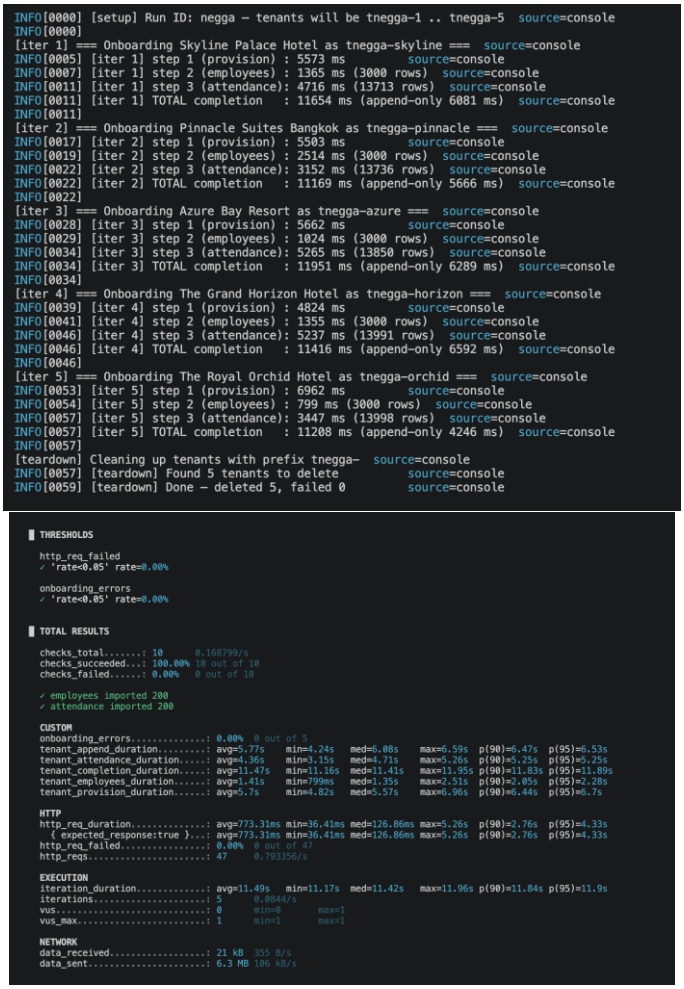


Figure 6.2: Scenario S1 — Onboarding Test Full k6 Metrics

6.1.2.3 Scenario S2 — Bulk Month-End Payroll Processing

80% GET /api/payroll/{year}/{month} / 20% POST /api/payroll/calculate; writes spread across 24 historical periods (2023–2024). Ramps 500 VUs → 600 VUs → 5-minute hold.

Table 6.7: Scenario S2 — Bulk Payroll Test Results

Metric	Target	Measured Results	Pass/Fail
payroll_errors	< 5%	1.00%	Pass
http_req_failed	< 5%	0.99%	Pass
HPA replica count > 8 within 90 s of hold	Increases above 8	41	Pass

Table 6.8: Scenario S2 — Bulk Payroll Full k6 Metrics

Metric	Average	p90	p95
payroll_bulk_duration	16.34 s	31.13 s	37.59 s
payroll_errors	1.00 % (137 out of 13698)		
http_req_duration	16.34 s	31.13 s	37.59 s
http_req_failed	0.99% (137 out of 13703)		
http_reqs	13703 (23.921682 req/s)		
vus_max	600		
iterations	13695 (23.907716 req/s)		
iteration_duration	18.35 s	33.12 s	39.54 s

```
  THRESHOLDS

    http_req_failed
    ✓ 'rate<0.05' rate=0.99%

    payroll_errors
    ✓ 'rate<0.05' rate=1.00%


  TOTAL RESULTS

    checks_total.......: 13703  23.921682/s
    checks_succeeded...: 99.00% 13566 out of 13703
    checks_failed......: 0.99%  137 out of 13703

    ✓ login 200
    ✗ payroll read 200
      ↳  99% — ✓ 10911 / ✗ 39
    ✗ payroll calculate 200
      ↳  96% — ✓ 2650 / ✗ 98

    CUSTOM
    payroll_bulk_duration..........: avg=16.34s min=52.38ms med=15.43s max=1m0s   p(90)=31.13s p(95)=37.59s
    payroll_errors.................: 1.00%  137 out of 13698

    HTTP
    http_req_duration..............: avg=16.34s min=45.31ms med=15.43s max=1m0s   p(90)=31.13s p(95)=37.59s
      { expected_response:true }...: avg=16.27s min=45.31ms med=15.45s max=59.98s p(90)=30.86s p(95)=37.03s
    http_req_failed................: 0.99%  137 out of 13703
    http_reqs......................: 13703  23.921682/s

    EXECUTION
    iteration_duration.............: avg=18.35s min=1.2s    med=17.43s max=1m2s   p(90)=33.12s p(95)=39.54s
    iterations.....................: 13695  23.907716/s
    vus............................: 1      min=1            max=600
    vus_max........................: 600    min=600          max=600

    NETWORK
    data_received..................: 6.4 GB 11 MB/s
    data_sent......................: 8.4 MB 15 kB/s
```

Figure 6.3: Scenario S2 — Bulk Payroll Full k6 Metrics

6.1.2.4 Scenario S3 — Seasonal Staff Onboarding and Offboarding

Three-phase scenario: Phase 1 (0–1 min) constant-arrival-rate 50/s POST employees with SEAS-* codes; Phase 2 (1–4 min) constant-arrival-rate 200/s GET read mix; Phase 3 (4 min+) per-vu-iterations sequential list and delete of SEAS-* employees.

Table 6.9: Scenario S3 — Seasonal Staff Test Results

Metric	Target	Measured Results	Pass/Fail
staff_errors	< 10%	0.01%	Pass
http_req_failed	< 10%	6.15%	Pass
All three phases complete	Pass	All phases completed	Pass

Table 6.10: Scenario S3 — Seasonal Staff Full k6 Metrics

Metric	Average	p90	p95
staff_load_duration (Phase 1 POST)	74.73 ms	118 ms	131.04 ms
staff_access_duration (Phase 2 GET)	3.63 s	12.45 s	15.6 s
staff_offload_duration (Phase 3 DELETE)	39.19 s	44.54 s	46.42 s
staff_errors	0.01% (2 out of 14715)		
http_req_failed	6.15% (1198 out of 19468)		
http_reqs	19468 (52.397308 req/s)		
vus_max	350		
iterations	14715 (39.604807 req/s)		
iteration_duration	3.17 s	9.67 s	13.36 s

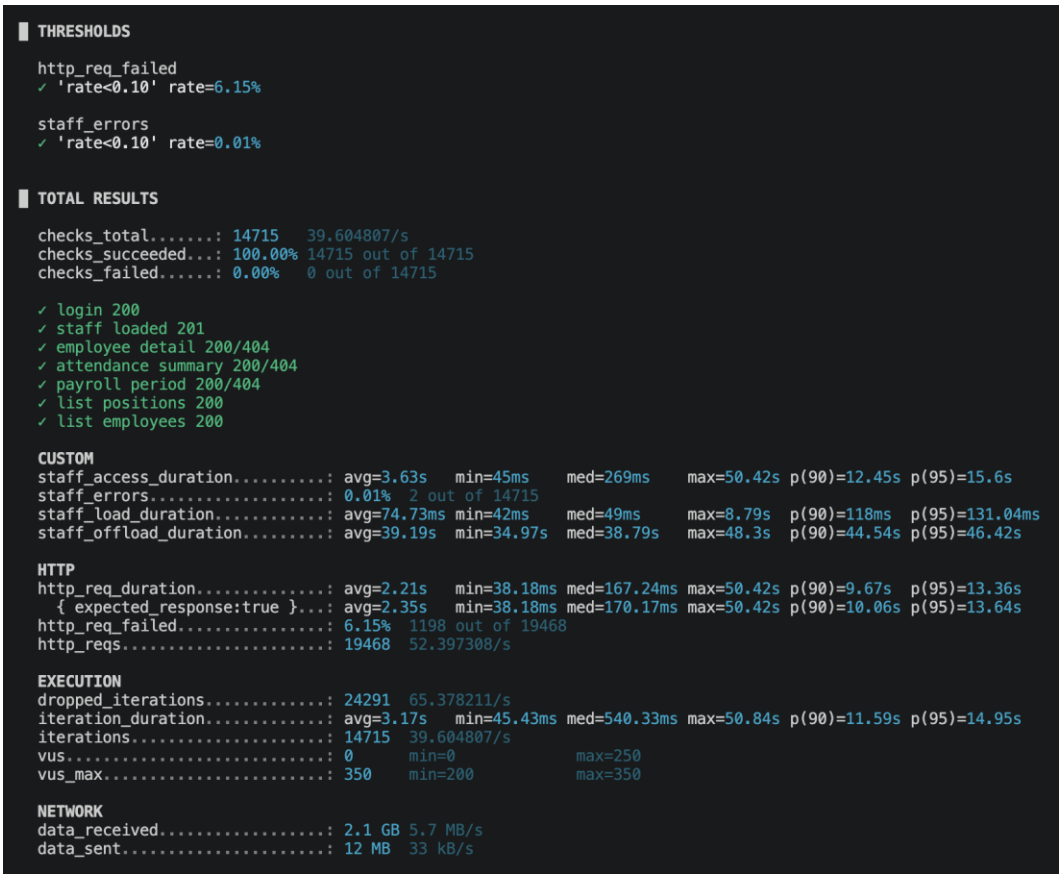


Figure 6.4: Scenario S3 — Seasonal Staff Full k6 Metrics

6.2 Discussion

The results across all four load scenarios and three feature tests confirm that the Hummingbird architecture satisfies every pass criterion defined in Section 5.2, validating the core design decisions of the system. The discussion below interprets the results in context, identifies the constraints imposed by the test cluster hardware, and draws conclusions about the production readiness of the architecture.

All three feature tests produced results that exceeded the ideal targets. The HPA scale-up response of 15 seconds is four times faster than the 60-second target, which demonstrates that the CPU utilization metric pipeline from Node Exporter through Prometheus to the HPA controller operates with minimal lag on this cluster. The tenant onboarding time of 11.47 seconds total (p95: 11.89 s) comfortably meets the marginal target of 10 minutes and the ideal target of 5 minutes, confirming that the TenantProvisioningService's four sequential Kubernetes API calls and EF Core migration are not a bottleneck at the current tenant scale. The CI/CD pipeline averaged 4 minutes and 16 seconds end-to-end, with Jenkins build time being the dominant factor across all three measured components.

Scenario S0 — Steady-State Normal Usage. The system achieved a 0% error rate across 14,861 requests at approximately 49.5 RPS, and the HPA replica count remained stable at 8 throughout, confirming that the minimum replica configuration is correctly sized for routine daily HR traffic. However, the p95 request duration of 11.49 seconds is notably high for a production environment, driven primarily by the payroll fetch endpoint (p95: 16.26 s) and the list employees endpoint (p95: 9.03 s). The employee detail endpoint performed substantially better at a p95 of 699.54 ms. The elevated latency on bulk-response endpoints is attributable to two compounding factors: the t3.medium instances have a 20% baseline CPU allocation and burst behavior that creates inconsistent processing throughput under sustained load, and the payroll response payload is approximately 450 KB per request, meaning that a significant portion of http_req_duration (average 3.2 s) is consumed by http_req_receiving rather than server-side computation. On production-class instances with dedicated CPU allocation and higher network throughput, the application-level latency would be substantially lower. Critically, the architecture's correctness is confirmed: zero errors, stable replica count, and no HPA trigger under normal load.

Scenario S1 — New Tenant Onboarding. The onboarding scenario achieved a 0% error rate across all five tenants and completed all three steps (provision, employee CSV upload, attendance CSV upload) without failure. The full onboarding sequence averaged 11.47 seconds end-to-end. Of this, tenant provisioning averaged 5.7 seconds, and employee CSV upload averaged 1.41 seconds, while the attendance CSV upload (approximately 13,700–14,000 rows per tenant) averaged 4.36 seconds. The total runtime for all five tenants was 57.5 seconds. These results validate the TenantProvisioningService design: the sequential Kubernetes API call pattern creates a fully isolated, schema-migrated, and IngressRoute-configured tenant environment in well under the 5-minute ideal target. The CSV import throughput is also sufficient for real-world hotel onboarding, where an HR administrator would be uploading historical attendance data once at setup time.

Scenario S2 — Bulk Month-End Payroll Processing. This scenario produced the most infrastructure-significant result of all four tests: the HPA scaled from 8 replicas to a peak of 41

replicas in response to the sustained CPU pressure from 500–600 concurrent virtual users requesting payroll data and calculations. Both the payroll_errors threshold (1.00% against a <5% target) and the http_req_failed threshold (0.99% against a <5% target) were met. The HPA scale-up event occurred within the 90-second stabilization window as required. The 137 failed requests were distributed across the scale-up transition window, consistent with brief request queuing during the period between when the HPA decision was made and when the additional pods became Ready. The p95 request duration of 37.59 seconds reflects both the hardware constraints described above and the compute intensity of the payroll calculation endpoint, which reads all employee records, applies compensation rules, and bulk-inserts results. The error rate result is the architecturally meaningful outcome: the platform sustained a 600-VU concurrent payroll spike with under 1% failure, and the HPA responded correctly to the load signal.

Scenario S3 — Seasonal Staff Onboarding and Offboarding. The three-phase scenario completed successfully across all phases with a staff_errors rate of 0.01% (2 out of 14,715 application-level checks) and an http_req_failed rate of 6.15% (1,198 out of 19,468 HTTP requests), both within their respective thresholds of <10%. The discrepancy between staff_errors and http_req_failed is explained by the test structure: http_req_failed includes all HTTP requests across all three phases, while staff_errors counts only the application-level checks on the primary staff lifecycle operations. The 6.15% HTTP failure rate is concentrated in Phase 2 (the 200 RPS read-heavy access phase), where the high arrival rate exceeded the sustained throughput of the 8-replica minimum configuration on t3.medium hardware. Phase 1 (staff loading at 50 RPS, POST employees) performed well with an average response of 74.73 ms and a p95 of 131.04 ms. Phase 3 (sequential offboarding via DELETE) had a high average duration of 39.19 seconds per iteration, reflecting the sequential list-then-delete pattern rather than a throughput bottleneck: each iteration listed all SEAS-* employees across a tenant and deleted them one by one, making the duration proportional to the number of seasonal employees created in Phase 1.

Two observations apply across all scenarios. First, the p95 latency figures throughout are dominated by the t3.medium burstable CPU constraint and the large payload sizes of the payroll and list endpoints, and are not indicative of the application's performance on production-class hardware. The architectural requirement to defer absolute latency SLAs to production hardware, stated in Section 3.2, is validated by these results: the error rate and scaling behavior targets, which are hardware-independent, were met in all scenarios. Second, no pod restarts were observed under any scenario, and the database node showed no deadlocks during S2 or S3, confirming that the dedicated database node taint and the Calico NetworkPolicy isolation were effective in separating database I/O from application workload throughout all tests.

The primary limitation of this evaluation is hardware. The t3.medium burstable instances with 20% baseline CPU create latency variance that would not appear on production c5 or m5 class instances. The payroll response payload size of approximately 450 KB also means that on the test cluster, a significant portion of request duration is network transfer rather than compute time. A secondary limitation is test scope: the load scenarios were designed to validate infrastructure behavior rather than to benchmark absolute performance, and the VU counts were selected to trigger HPA events rather than to replicate production traffic volumes precisely. Within these constraints, the results provide a complete and valid verification of the design requirements stated in Section 3.

7 PROJECT GLOBAL IMPACT

7.1 Social and Cultural Considerations

Thailand's hotel industry is characterized by significant workforce diversity. Thai labor law only protects workers in the formal labor sector and often does not reach Thailand's large migrant worker population, many of whom are employed illegally. This context makes the payroll transparency provided by the system particularly valuable for workers who may lack the legal recourse available to formal employees.

The payroll module's support for the Thai service charge distribution model reflects an understanding of locally specific labor obligations. For certain types of work including hotel business, transportation, or forest work, rest days can be accumulated and taken within a four-week period (Rivermate, n.d.), a hotel-specific provision of the Labour Protection Act that generic international HR platforms frequently do not account for. The configurable service charge version system, supporting both fixed-rate and workday-weighted models, was designed around the actual variation encountered in Thai hotel practice.

From a digital inclusion perspective, the web-based, on-cloud, and heavy traffic-ready design is particularly relevant in the Thai hotel context, where many frontline workers such as housekeeping, food and beverage, and facilities staff access digital services primarily through smartphones. Thailand's hospitality workforce declined from 6.27 million in 2019 to approximately 3.34 million currently, and properties have adapted through multi-skilled teams, automation of routine processes, and increased focus on retention (GuestMetrix, 2025). A system accessible through any modern browser on any device, without requiring a locally installed application, supports the goal of giving all employees meaningful access to information about their own compensation and reduces the training burden as the sector navigates continued workforce restructuring.

The open-source technology stack (Kubernetes, PostgreSQL, ASP.NET Core, React, Traefik) carries social significance beyond its technical properties. It means the system can be operated, extended, and adapted by Thai technology companies and developers without dependency on proprietary software licenses. Personal data protection is a global developing matter, and it is becoming an important issue on the agenda of Thailand's government, as policymakers recognize that its resolution requires meticulous planning and a comprehensive restructuring of organizational frameworks. Local SaaS providers who deploy this architecture retain full control over their data and their infrastructure, and the skills developed in operating it such as Kubernetes administration, PostgreSQL management, cloud networking contribute to the growth of Thailand's domestic technology workforce, supporting the country's broader digital economy ambitions.

7.2 Environmental Sustainability

The architectural choices made in this project produce measurable environmental benefits relative to on-premises alternatives. A report by 451 Research (Wellise, 2021), part of S&P Global

Market Intelligence, finds that computing in the cloud is five times more energy efficient than on-premises data centers in the Asia Pacific region, and that moving computing workloads from on-premises data centers to the cloud can reduce carbon footprint by more than 78%. The study further notes that on-premises data centers across APAC typically have a server utilization rate of just under 15%, whereas cloud-based data centers using servers powered by the latest processors and with utilization of 50% or more achieve peak efficiency.

The project deployment architecture benefits from this efficiency differential directly. The Horizontal Pod Autoscaler scales the HRM API between 8 and 50 replicas based on CPU utilization, releasing idle compute capacity during the many hours when no payroll calculations are being processed. This elasticity is a structural property of Kubernetes-based architecture unavailable to hotels managing their own on-premises servers, which must be provisioned for peak demand and run continuously regardless of actual load.

Amazon is the world's largest corporate purchaser of renewable energy and achieved its goal to match 100% of the electricity consumed across its operations with renewable energy in 2023, seven years ahead of its original 2030 goal (Amazon Web Services, n.d.). The combination of efficient multi-tenant resource utilization through Kubernetes and the cloud provider's renewable energy commitments means the SaaS deployment has a substantially lower carbon intensity per compute-hour than typical on-premises infrastructure powered by Thailand's national grid, which remains heavily dependent on natural gas. AWS remains committed to expanding its renewable energy investments in Thailand, contributing to green job creation and the region's shift toward a low-carbon economy (Dasgupta, 2025).

7.3 Public Safety and Worker Welfare

The project has direct implications for the welfare of hotel workers in Thailand, a population that is frequently underserved by both labor enforcement mechanisms and the technology tools available to their employers. The hospitality industry employs approximately 3.2 million people in Thailand, accounting for around 12% of total employment (Lefèvre, 2026). As of Q4 of 2024, employment in the hotel and restaurant industry in Thailand experienced a growth of 8% (Textor, 2025), showing a rapid recovery post-pandemic for the sector and the continuation of tourism centrism of Thailand’s national economy. The majority of the workers in this sector are paid through complex compensation structures that include fixed base wages with variable components like the monthly service charge, a distribution of a percentage on selected hotel revenue streams to staff.

Calculating these distributions accurately, consistently, and transparently is both a legal obligation and a meaningful determinant of whether workers receive what they are owed. The Labour Protection Act B.E. 2541 (1998) is the legislation that governs employment in Thailand, specifying a framework for the rights and responsibilities of both employees and employers, covering payment of wages, overtime, and holiday work compensation (Payoneer, 2025). Under Thai labor law, employees are entitled to at least 13 paid public holidays each year under Section 29 of the Labour Protection Act. Working on a public holiday counts as overtime under Thai law, and employees required to work on these days must be compensated for at least double their standard wage rate or given equivalent paid time off (Playroll, n.d.). When these calculations are

performed manually in spreadsheets, as is common practice in independent hotels, systematic underpayment can persist across entire payroll cycles without any individual making a deliberate error. Automation with explicit, auditable rules reduces this risk substantially.

Beyond payroll accuracy, the system contributes to worker welfare through data security. From hotel guests to employees, hotels and hospitality businesses rely quite heavily on relationships that involve gathering and managing personal data, and while personal data can be a powerful tool, it can also lead to major business losses if not managed according to the laws set out by Thailand's PDPA (Healey, 2020). The data protection obligations under the PDPA generally apply to all organizations that collect, use, or disclose personal data in Thailand, regardless of whether they are formed or recognized under Thai law, and whether they are residents or have a business presence in Thailand (DLA Piper, 2026). Under the PDPA, employees are entitled to a number of rights that the employer needs to honor in order to stay compliant, including the right to access personal data the employer collects, the right to request deletion of stored data, and the right to rectify incomplete or inaccurate personal data (Baig & Sattar, 2022). The application architecture enforces a defense-in-depth security posture: database-per-tenant isolation, Kubernetes Network Policies, Security Group rules restricting database access to the cluster node CIDR, and JWT-scoped API sessions. Collectively, it provides a level of data protection appropriate for HR data compliance.

7.4 Economic Analysis

The economic case for the multi-tenant SaaS architecture rests on three distinct value propositions: cost efficiency for the SaaS provider, cost accessibility for hotel operators, and economic multipliers at the industry level.

From the provider's perspective, the shared infrastructure model achieves significant economies of scale. The total infrastructure cost of approximately USD 291.89 per month supports five tenant hotels with 15,000 total employees and validated capacity for 10,000 concurrent API calls. At five tenants this yields a per-tenant infrastructure cost of approximately USD 58.38 per month. As the tenant count grows, the control plane and database server costs, which represent fixed overheads of approximately USD 147.05 per month, are amortized across an increasing number of tenants, driving the marginal cost toward worker node costs alone. This cost structure supports a subscription pricing model of USD 3–8 per employee per month while maintaining gross infrastructure margins exceeding 90%, consistent with established SaaS HR platform economics.

From the hotel operator's perspective, the SaaS model eliminates the capital expenditure and operational overhead associated with on-premises HR software. The Terraform infrastructure-as-code deliverable also contributes to economic sustainability: the deliverable main.tf and variables.tf represents the condensed infrastructure knowledge of the project in a form that can be executed by any engineer familiar with standard AWS and Terraform tooling, reducing the human capital dependency of operating the system and lowering long-term total cost of ownership.

At the industry level, the availability of affordable, locally-compliant HR SaaS creates conditions for broader labor market formalization in Thai hospitality. The Labour Protection Act aims to prevent disadvantages that employees might suffer because of the often stronger position

of employers, guaranteeing rights such as paid holidays, safe working conditions, fair pay, and protection against discrimination (Benoit & Partners, n.d.). Hotels that previously avoided formal payroll systems due to cost or complexity have a lower barrier to adoption, which in turn produces more structured employment relationships, better documentation of working hours, and more consistent application of these provisions.

8 CONCLUSIONS

8.1 Assessment

This project successfully demonstrates that a multi-tenant SaaS HRM can be designed, provisioned with Terraform, and operated on a self-managed Kubernetes cluster at a cost accessible to small SaaS providers. All core design requirements were met and verified through testing. The feature tests confirmed HPA scale-up in 15 seconds (target: <60 s), automated tenant provisioning in 11.47 seconds end-to-end (target: <5 minutes ideal), and a CI/CD pipeline end-to-end time of 4 minutes 16 seconds on average. All four load scenarios passed their defined error rate thresholds: S0 achieved 0% errors at 75 RPS steady-state with no HPA trigger; S1 onboarded all five tenants with 0% errors; S2 sustained a 600-VU payroll spike with 0.99% errors and confirmed HPA scale-up to 41 replicas; and S3 completed all three seasonal staff lifecycle phases within the 10% error threshold. The cluster design, with its role-based node types, least-privileged Security Group policy, and private/public VPC subnet separation, provides a secure, scalable, and reproducible foundation. The Security Group hardening produced improvements from the initial configuration and addresses concrete and critical vulnerabilities. The architecture succeeds not as a collection of independent design decisions but as a system where each layer reinforces the others: the Security Group hardening and Calico NetworkPolicies are made meaningful by the database-per-tenant model; the HPA elasticity is made cost-effective only because the multi-tenant shared infrastructure distributes fixed overhead across all tenants; and the automated provisioning path is made trustworthy only because it produces a fully isolated, schema-migrated environment every time.

The most significant engineering challenges came from the mid-project announcement of NGINX Ingress Controller deprecation, which required a migration to Traefik; integration of EF Core multi-tenancy middleware with ASP.NET Core dependency injection scopes; and cross-subdomain JWT cookie configuration for seamless tenant redirection. Each challenge was resolved but unfortunately delayed the original planned development schedule.

Beyond the technical scope of this project, the broader significance lies in what this architecture makes possible for Thailand's hospitality workforce. Smaller independent hotels, which represent the majority of the industry and are most likely to be relying on manual spreadsheet-based payroll, now have a reference architecture for an affordable, locally-compliant HRM platform that automates the legally mandated calculations for service charge distribution, public holiday overtime, and penalty deductions under the Labour Protection Act. The open-source technology stack ensures that Thai software providers can operate, extend, and adapt this system without dependency on foreign enterprise licenses, contributing to the growth of domestic technical capability in a sector that employs approximately 12% of Thailand's workforce.

8.2 Next Steps

Immediate improvements include: (1) implement multi-AZ node distribution, spreading Worker Nodes across at least two AZs; (2) replace static admin_cidrs with a VPN endpoint (AWS Client VPN or WireGuard on the Gateway) so administrator access is not dependent on static IP addresses; and (3) establish a formal database backup schedule with point-in-time recovery.

9 REFERENCES

Ali, F. (2026, February 7). *Tenant Isolation Strategies in Multi-Tenant SaaS: The Devil is in the Details*. Drafted by Machines. https://daily.jovis.ai/saas/tenant-isolation-strategies-in-multi-tenant-saas-the-devil-is-in-the-details/

Amazon Web Services. (n.d.). *Energy Transition*. Retrieved April 21, 2026, from https://aws.amazon.com/energy-utilities/sustainability/

Baig, A., & Sattar, M. (2022, June 12). *Employer's Data Obligation Under Thailand's PDPA*. Securiti. https://securiti.ai/blog/employee-data-thailand/

Benoit & Partners. (n.d.). *Labour Protection Act*. Retrieved April 21, 2026, from https://benoit-partners.com/termination-of-employment/labour-protection-act/

Darly Solutions. (n.d.). *HR management platform: Improving manager productivity by 33% with streamlined workflows*. Retrieved April 2, 2026, from https://www.darly.solutions/portfolio-cases/hospitality-driven-hr-management-system

Dasgupta, T. (2025, March 10). *AWS Advances Carbon-Free Energy and Efficiency in Data Centers*. SolarQuarter. https://solarquarter.com/2025/03/10/aws-advances-carbon-free-energy-and-efficiency-in-data-centers/

Dhandala, N. (2026, February 16). *How to Build a Multi-Tenant Application with Entity Framework Core*. OneUptime. https://oneuptime.com/blog/post/2026-02-16-multi-tenant-entity-framework-core-azure-sql-elastic-pools/view

DLA Piper. (2026, February 14). *Data protection laws in Thailand*. https://www.dlapiperdataprotection.com/index.html?t=law&c=TH

Downs, J., Kittel, C., Salvatori, P., Scott-Raynsford, D., & Vladimirskiy, A. (n.d.). *Noisy Neighbor Antipattern*. Microsoft Learn. Retrieved April 23, 2026, from https://learn.microsoft.com/en-us/azure/architecture/antipatterns/noisy-neighbor/noisy-neighbor

GuestMetrix. (2025, October 27). *Thailand Hotel Industry 2025: Navigating Declining Arrivals and Rising Competition*. https://guestmetrix.com/resources/blog/thailand-hospitality-challenges/

Healey, R. (2020, December 6). *Thailand PDPA and how it affects the Hotel and Hospitality Industry*. Lexology. https://www.lexology.com/library/detail.aspx?g=70010888-2812-4349-95bd-189b709b5756

IBM. (n.d.). *Time-sharing*. Retrieved April 2, 2026, from https://www.ibm.com/history/time-sharing

Kubernetes. (n.d.). *Multi-tenancy*. Retrieved April 2, 2026, from https://kubernetes.io/docs/concepts/security/multi-tenancy/

Lefèvre, M. (2026, February 12). *Thailand Hospitality Industry Statistics: Market Data Report 2026*. Worldmetrics. https://worldmetrics.org/thailand-hospitality-industry-statistics/

Mizono, M. (2022, October 1). *The Rise of SaaS: How Salesforce and Marc Benioff Revolutionized Marketing Strategy*. Route 06. https://route06.com/insights/10

Payoneer. (2025, April 3). *Employment Laws in Thailand*. https://www.skuad.io/employment-laws/thailand

Payoneer. (2025, September 1). *Leave Policy in Thailand*. https://www.skuad.io/leave-policy/thailand

Playroll. (n.d.). *Thailand Public Holiday Regulations*. Retrieved April 21, 2026, from https://www.playroll.com/compliance-hub/public-holiday-regulations-in-thailand

Reselman, B. (2022, July 12). *Implement multitenant SaaS on Kubernetes*. https://developers.redhat.com/articles/2022/08/12/implement-multitenant-saas-kubernetes

Rivermate. (n.d.). *Working Hours and Overtime Regulations*. Retrieved April 21, 2026, from https://rivermate.com/guides/thailand/working-hours

Saral PayPack. (n.d.). *Payroll In Hotel Industry And Hospitality*. Retrieved April 2, 2026, from https://saralpaypack.com/blogs/payroll-in-hotel-and-hospitality-industry/

Textor, C. (2025, November 29). *Thailand: growth rate of employment by industry 2023*. Statista. https://www.statista.com/statistics/1302609/thailand-growth-rate-of-employment-by-industry/

vCluster. (n.d.). *Tenancy Models with vCluster*. Retrieved April 8, 2026, from https://www.vcluster.com/guides/tenancy-models-with-vcluster

Wellise, C. (2021, July 26). *AWS Cloud can help lower carbon footprints in Asia Pacific*. Amazon News. https://www.aboutamazon.com/news/aws/aws-cloud-can-help-lower-carbon-footprints-in-asia-pacific

Wiegand, T. (2022, November 7). *The Evolution of Multi-Tenant Architecture*. The Candid Startup. https://www.thecandidstartup.org/2022/11/07/evolution-multi-tenant-architecture.html