Technical Design Document: Autonomous Ecommerce Provisioning Platform for Urumi AI

Executive Summary and Architectural Vision

The objective of this technical design document is to define the architectural blueprint, implementation strategy, and operational rigor required to construct a production-grade "Store Provisioning Platform" for the Urumi AI SDE Internship. The proposed system serves as a foundational infrastructure layer capable of orchestrating complex, stateful ecommerce workloads—specifically WooCommerce and MedusaJS—on Kubernetes.[1] This platform is not merely a deployment script; it is designed as a scalable, multi-tenant control plane that bridges the gap between traditional DevOps automation and the emerging paradigm of AI-driven infrastructure management required for Round 2 of the assessment.[1]

To maximize the probability of success in both rounds, the architecture prioritizes distinct separation of concerns, utilizing a three-tier model: a React-based frontend for user interaction, a Node.js API for state management and validation, and a high-performance Go-based orchestration controller that interfaces directly with the Kubernetes API via the Helm SDK.[2] This decoupled approach ensures that the system is resilient, idempotent, and capable of handling the asynchronous nature of cloud resource provisioning. Furthermore, the design explicitly addresses the "local-to-production" requirement by enforcing strict configuration management through Helm values, ensuring that the exact same artifacts deployed to a local Kind/k3d cluster can be promoted to a production Azure VPS running k3s with zero code changes.[4]

The architecture anticipates the requirements of Round 2—Gen AI Orchestration—by exposing a deterministic, well-documented API surface that Large Language Models (LLMs) can reason about and invoke reliably.[6] By treating infrastructure-as-code (IaC) as a functional toolset accessible to AI agents, the platform lays the groundwork for natural language-driven store creation, modification, and management.

1. System Architecture and Component Design

The platform architecture is composed of four primary domains: the User Interface Domain, the Control Plane Domain, the Orchestration Domain, and the Data Plane Domain. Each domain interacts through strict interfaces, ensuring that internal implementation details (such as the specific ingress controller or storage provider) are abstracted away from the consuming services.

1.1 The User Interface Domain (Node Dashboard)

The requirement mandates a Node/React dashboard.[1] This component serves as the visual control center for the platform. It is responsible for:

- **State Visualization:** Polling the backend to display real-time status updates (Provisioning, Ready, Failed).[1]
- **User Intent Capture:** Collecting configuration parameters (store name, type, subdomain) and submitting them to the API.
- **Access Management:** Providing the interface for store deletion and resource teardown.

To support the "Stand Out" requirements, the dashboard should also visualize advanced metrics such as provisioning duration and resource utilization, which are fetched from the underlying Kubernetes metrics server or a Prometheus instance.[1]

1.2 The Control Plane Domain (Backend API)

The Backend API, implemented in Node.js, acts as the system's brain. It maintains the "Desired State" of the platform. Unlike a simple proxy, this component persists store metadata (ID, owner, creation time, type, configuration) in a relational database (PostgreSQL). This persistence layer is crucial for:

- **Idempotency:** Preventing duplicate provisioning requests for the same store ID.
- **Quota Management:** Enforcing limits on the number of stores per user or IP address to prevent abuse.[1]
- **Audit Logging:** Recording who created what and when, satisfying security requirements.[1]

The API exposes endpoints (e.g., POST /api/stores, DELETE /api/stores/:id) that do not directly touch Kubernetes. Instead, they publish events or enqueue jobs that the Orchestration Domain consumes. This asynchronous pattern prevents long-running Kubernetes operations from blocking the user interface.

1.3 The Orchestration Domain (Go/Helm Controller)

This is the most critical component for the System Design assessment. While the prompt allows for a Node.js backend to handle provisioning, a Principal Architect's approach favors a dedicated controller written in Go. Go is the native language of Kubernetes and Helm, providing first-class support for their SDKs.[2]

Using the Helm Go SDK offers significant advantages over wrapping the Helm CLI in shell commands:

- **Type Safety:** Compilation checks prevent syntax errors in command arguments.
- **Granular Error Handling:** The SDK returns structured errors rather than text blobs from stderr, allowing for sophisticated retry logic and status reporting.[2]

- **Performance:** Eliminates the overhead of spawning new OS processes for every Helm command.
- **Context Management:** Allows for graceful cancellation of provisioning jobs using Go contexts.[10]

The Orchestrator watches the Backend API (or a shared queue) for provisioning tasks. Upon receiving a task, it executes the following workflow:

1. **Namespace Creation:** Creates a dedicated Kubernetes namespace for the store (e.g., store-xyz-123).[12]
2. **Secret Injection:** Generates random credentials for the database and application admins and injects them as Kubernetes Secrets.[1]
3. **Helm Install:** Uses action.NewInstall to deploy the unified ecommerce-store chart into the namespace.[3]
4. **Post-Provisioning Automation:** Triggers necessary jobs (like WP-CLI setup) to reach the "Ready" state.[17]

1.4 The Data Plane Domain (Workloads)

This domain consists of the actual ecommerce applications running in the cluster.

- **WooCommerce:** A Pod running Apache/PHP with WordPress, connected to a MySQL StatefulSet.[19]
- **MedusaJS:** A Pod running the Node.js backend, connected to a PostgreSQL StatefulSet and a Redis Deployment.[20]

These workloads are isolated by namespace and protected by NetworkPolicies, ensuring that a compromise in one store does not affect others.[22]

1.5 Architecture Comparison Table

Feature	Standard Implementation (Junior/Mid)	Principal Architect Implementation (Target)
Language	Node.js wrapping exec('helm install')	Go Service using helm.sh/helm/v3/pkg/action SDK
Isolation	Shared namespace with	Dedicated Namespace per

	unique labels	Tenant + RBAC + Quotas
State	In-memory or reliance on K8s tags	Persistent Relational DB (Postgres) + K8s State Reconciliation
Ingress	Single Ingress resource editing	One Ingress per Store (fan-out) via Ingress Controller
Security	Default settings	NetworkPolicies (Deny-All default), Non-root containers
Automation	Manual setup steps in UI	Fully automated WP-CLI / Medusa migrations via Jobs

2. Kubernetes Native Provisioning & Isolation Strategy

The "Definition of Done" requires a system that supports multi-tenant isolation and robust resource management.[1] This section details the Kubernetes primitives utilized to achieve this.

2.1 Namespace-as-a-Boundary

The "Namespace-per-Store" pattern is selected as the primary isolation mechanism.[1] This offers several operational benefits:

- **Resource Quotas:** We can apply a ResourceQuota object to the namespace, rigidly defining the maximum CPU, RAM, and Storage the tenant can consume.[23]
- **Access Control:** RBAC roles can be scoped to the namespace, allowing the platform to grant permissions for specific store operations without exposing the cluster scope.
- **Clean Teardown:** Deleting the namespace automatically garbage collects all resources (PVCs, Secrets, Services, Deployments) contained within it, simplifying the deletion logic.[13]

2.2 Resource Constraints and Quotas

To prevent "noisy neighbor" issues where one high-traffic store starves others, strict ResourceQuota and LimitRange configurations are applied during namespace creation.[1]

Example ResourceQuota Strategy:

- **Requests:** Guaranteed resources required to start the store (e.g., 0.5 vCPU, 512Mi RAM).
- **Limits:** Maximum burstable resources (e.g., 1.0 vCPU, 1Gi RAM).
- **Storage:** Limit on total PVC capacity (e.g., 5Gi) to prevent disk exhaustion.

Example LimitRange Strategy:

- Sets default request/limit values for any container created without explicit specifications, ensuring no pod runs unbounded.[26]

2.3 Persistent Storage Strategy

Persistence is mandatory for the database layers (MySQL/PostgreSQL).[1] The architecture utilizes Kubernetes PersistentVolumeClaims (PVCs) which abstract the underlying storage technology.

- **Local Development (Kind/k3d):** Uses the local-path storage class, which maps PVCs to directories on the host machine.[27]
- **Production (Azure):** Uses the managed-csi (Azure Disk) storage class. This provides durability, snapshots, and can be detached/reattached to different nodes if a node fails.[29]

The Helm chart is parameterized to accept global.storageClass as a value, allowing the Orchestrator to inject the correct class based on the environment (values-local.yaml vs values-prod.yaml).[14]

2.4 Networking and Ingress

The platform must expose each store on a stable URL.[1]

- **Ingress Controller:** We utilize NGINX Ingress Controller due to its widespread adoption, granular annotation support (e.g., rate limiting, rewriting), and stability.[5] While K3s includes Traefik by default, switching to NGINX demonstrates "Stand Out" capability in managing cluster components.[4]
- **DNS Strategy:**
  - **Local:** A wildcard domain *.local.urumi.ai (or similar) pointing to 127.0.0.1.
  - **Production:** A wildcard DNS record *.urumi.app pointing to the Azure VM's public IP.
- **Traffic Routing:** The Orchestrator creates an Ingress resource for each store with host: store-id.urumi.app. The Ingress Controller reads the Host header of incoming HTTP requests and routes traffic to the correct Service within the correct Namespace.[35]

2.5 Network Security Policies (Hardening)

To satisfy the security requirements [1], the architecture implements **NetworkPolicies** using a CNI plugin like Calico or Cilium.[24] The default posture is "Default Deny," meaning no traffic is allowed unless explicitly whitelisted.

Policy Rules:

1. **Deny All Ingress/Egress:** Applied to the store namespace by default.[24]
2. **Allow DNS:** Egress allowed to kube-system/coredns on UDP/53.[22]
3. **Allow Ingress from Controller:** Ingress on port 80/443 allowed only from the ingress-nginx namespace.[38]
4. **Allow Database Access:** The WordPress pod is allowed to talk to the MySQL pod on port 3306.[38]

This micro-segmentation ensures that if a WordPress container is compromised via a plugin vulnerability, the attacker cannot scan the internal network or access other stores' databases.[23]

3. Helm Strategy: The Universal Deployment Artifact

Helm is mandatory for this assessment.[1] The strategy involves creating a single, highly parameterized "Universal Store Chart" that can deploy either WooCommerce or MedusaJS based on the values provided. This avoids maintaining two completely separate charts and promotes code reuse.

3.1 Chart Structure

The chart follows a library pattern structure [42]:

```
charts/universal-store/
├── Chart.yaml          # Metadata
├── values.yaml         # Default values (Production-ready defaults)
├── values-local.yaml   # Overrides for local dev (NodePort, no TLS)
├── values-prod.yaml    # Overrides for Prod (LoadBalancer, Let's Encrypt)
├── templates/
│   ├── _helpers.tpl    # Template logic for names and labels
│   ├── common/
│   │   ├── namespace.yaml # (Optional, if not handled by Orchestrator)
│   │   ├── networkpolicy.yaml
│   │   ├── resourcequota.yaml
│   │   └── ingress.yaml
```

```
│   ├── woocommerce/
│   │   ├── deployment.yaml
│   │   ├── service.yaml
│   │   ├── pvc.yaml
│   │   ├── secret.yaml
│   │   └── job-setup.yaml # WP-CLI Automation
│   └── medusa/
│       ├── deployment.yaml
│       ├── service.yaml
│       ├── pvc.yaml
│       ├── secret.yaml
│       └── job-migration.yaml
└── charts/             # Dependencies (optional)
```

3.2 Conditional Logic

The templates/ directory utilizes Go templating conditionals to control what gets deployed.[30]

- {{- if eq.Values.engine "woocommerce" }} surrounds the WordPress and MySQL resources.
- {{- if eq.Values.engine "medusa" }} surrounds the Medusa backend, Postgres, and Redis resources.

This allows a single helm install command to provision widely different stacks based purely on the engine value.

3.3 The "Local-to-Prod" Value Strategy

The difference between environments is strictly managed via values files, fulfilling the requirement for portability.[1]

Configuration Key	values-local.yaml	values-prod.yaml
ingress.className	traefik (if default k3d) or nginx	nginx
ingress.tls.enabled	false	true

ingress.annotations	(Empty)	cert-manager.io/cluster-issuer: letsencrypt-prod
storage.className	local-path	default (Azure Disk)
resources.requests	Low (e.g., 100m CPU)	Moderate (e.g., 500m CPU)
replicaCount	1	2 (for high availability)

3.4 Automated Provisioning via Helm Go SDK

The Orchestrator utilizes the Helm Go SDK to perform installs programmatically. This is superior to os.Exec wrappers because it allows direct manipulation of the release object and better status monitoring.

Key SDK Implementation Details:

- **Configuration:** action.NewInstall(actionConfig) initializes the installer.[3]
- **Namespace Management:** client.CreateNamespace = true and client.Namespace = "store-id" ensures the namespace is created as part of the release.[44]
- **Value Injection:** User options (e.g., Store Name) are passed as a map[string]interface{} to the Run() method, effectively doing a --set operation dynamically.[3]
- **Wait Strategy:** client.Wait = true forces the controller to wait until all Pods are Ready before returning a success status to the UI.[11]

4. Workload Specifics: Achieving the "Definition of Done"

The assessment requires that the provisioned store is *actually working*, meaning a user can check out. This requires automation beyond simple container startup.

4.1 Automating WooCommerce (The Hard Part)

WordPress/WooCommerce is notoriously difficult to automate because it relies on an interactive "Setup Wizard" upon first login.[47] A standard docker run wordpress is insufficient because the database tables for WooCommerce are not initialized, and the store settings

(currency, payments) are unset.

Solution: The wp-cli Automation Job To pass the assessment, the Helm chart must include a Kubernetes Job (hooked post-install) that runs wp-cli commands against the WordPress container to configure it without human intervention.[17]

The Automation Script Sequence:

1. **Wait for DB:** The script loops until MySQL is ready.
2. **Core Install:** wp core install --url=... --title=... --admin_user=... sets up the WP tables.[49]
3. **Plugin Install:** wp plugin install woocommerce --activate downloads and enables the plugin.[52]
4. **Bypass Wizard:** This is crucial. We must update database options to fool WooCommerce into thinking the wizard is complete.
   - wp option set woocommerce_onboarding_profile '{"skipped": true}' --format=json.[54]
   - wp option set woocommerce_task_list_hidden yes.[47]
5. **Enable Payments:** wp option update woocommerce_cod_settings '{"enabled":"yes"}' enables Cash on Delivery, allowing test checkouts without a credit card.[55]
6. **Seed Products:** wp wc product create --name="Demo Product" --regular_price="10"... creates a test product so the user can immediately "Add to Cart".[51]

This script is mounted as a ConfigMap and executed by a Job sharing the same PVC/Network as the WordPress pod.[48]

4.2 Automating MedusaJS

Medusa is more modern and API-first, simplifying automation. The Helm chart includes a post-install Job that runs medusa migrations run and medusa seed to populate the Postgres database.[20] The storefront (Next.js) is deployed as a separate service pointing to the backend API URL.

5. Security Strategy and Multi-Tenancy

Security cannot be an afterthought; it must be intrinsic to the provisioning process.

5.1 RBAC and Least Privilege

The Orchestrator runs with a Kubernetes ServiceAccount. Instead of giving it cluster-admin, we define a ClusterRole that permits creating Namespaces and a limited set of resources *only*.[12]

- **Can:** Create/Delete Namespaces, Deployments, Services, Ingress, Secrets, PVCs.
- **Cannot:** Modify Nodes, view Secrets in kube-system, alter ClusterRoles.

5.2 Secret Management

Requirement: No hardcoded secrets.[1] **Implementation:**

1. The Node.js Backend generates a random 32-character string for the database password.
2. It passes this string to the Orchestrator.
3. The Orchestrator creates a Kubernetes Secret object db-creds in the target namespace.
4. The Helm chart references this Secret via envFrom or valueFrom: secretKeyRef in the Deployment spec.[14] This ensures passwords exist only in the volatile memory of the Orchestrator during creation and securely in etcd (encrypted at rest on Azure) thereafter.

5.3 Abuse Prevention

To satisfy the "Abuse Prevention" requirement [1]:

- **Rate Limiting:** The Node.js API implements express-rate-limit to block IP addresses requesting more than 5 stores in 1 minute.
- **Quotas:** The database tracks count(stores) where user_id = X. If count >= 3, the request is rejected before ever reaching Kubernetes.
- **Timeouts:** The Helm SDK Run() context is set to timeout after 5 minutes. If the store isn't ready, the Orchestrator triggers a Uninstall() rollback to clean up resources.[11]

6. The "Local-to-VPS" Production Execution Blueprint

This section provides the step-by-step execution plan for deploying to an Azure VPS, satisfying the "Ways to Stand Out" requirement.[1]

6.1 Azure Infrastructure Provisioning

We do not use AKS (Managed Kubernetes) for this specific requirement to demonstrate "VPS/k3s" mastery, although the architecture supports AKS.

1. **VM Creation:** Create an Ubuntu 22.04 VM (Standard_B2s or D2s) using Azure CLI or Portal.
2. **Networking:** Configure the Azure Network Security Group (NSG) to allow Inbound TCP on ports 22 (SSH), 80 (HTTP), 443 (HTTPS), and 6443 (K8s API).[59]
3. **Cloud-Init:** Use a cloud-init script to bootstrap K3s automatically on VM launch.[61]

K3s Installation Command:

Bash

```
curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="--tls-san <VM_PUBLIC_IP> --disable traefik" sh -
```

- --tls-san: Adds the Azure Public IP to the API server certificate so we can control it remotely.[4]
- --disable traefik: We disable the default Traefik to install NGINX Ingress Controller via Helm, demonstrating granular control.[4]

6.2 Remote Cluster Access

1. SSH into the Azure VM and copy /etc/rancher/k3s/k3s.yaml.
2. Edit the file locally, replacing 127.0.0.1 with the Azure Public IP.
3. Set export KUBECONFIG=./azure-k3s.yaml. Now, local tools (Helm, Kubectl, the Orchestrator) talk to Azure.

6.3 Ingress and TLS (Cert-Manager)

Production requires real HTTPS.

1. **Install Cert-Manager:** Use the official Helm chart to install cert-manager into the cluster.[66]
2. **Configure ClusterIssuer:** Create a ClusterIssuer resource configured for Let's Encrypt Production.[69]
3. **DNS:** Create a wildcard A record (*.demo.urumi.app) in your DNS provider (e.g., Namecheap/Cloudflare) pointing to the Azure VM IP.[34]
4. **Ingress Class:** Install ingress-nginx via Helm. It will bind to the host ports 80/443 of the Azure VM (since K3s uses host networking for LoadBalancers by default).[31]

When a store is provisioned with values-prod.yaml, the Ingress resource will have the annotation cert-manager.io/cluster-issuer: letsencrypt-prod. Cert-manager will see this, solve the HTTP-01 challenge automatically, and create a TLS Secret.[34]

7. Round 2 Strategy: Gen AI Orchestration

The infrastructure designed above is explicitly "AI-Ready." Round 2 asks to create stores via Gen AI. This architecture simplifies that task immensely.

7.1 The AI-API Interface

LLMs (like GPT-4) interact with systems via "Function Calling" or "Tool Use" definitions.[6] Because our Backend API is structured and strictly typed, we can auto-generate an OpenAPI (Swagger) specification.[7]

Example Function Definition for the AI:

JSON

```
{
  "name": "provision_store",
  "description": "Deploys a new ecommerce store infrastructure.",
  "parameters": {
    "type": "object",
    "properties": {
      "name": {"type": "string"},
      "engine": {"type": "string", "enum": ["woocommerce", "medusa"]},
      "region": {"type": "string", "enum": ["us-east", "eu-west"]}
    },
    "required": ["name", "engine"]
  }
}
```

7.2 The ReAct Loop

The AI agent operates in a loop:

1. **User Prompt:** "I need a shoe store."
2. **Reasoning:** "I should provision a store. The user didn't specify an engine, so I'll ask or default to Medusa."
3. **Action:** AI calls provision_store(name="shoe-store", engine="medusa").
4. **Observation:** The API returns {"status": "provisioning", "id": "store-xyz"}.
5. **Response:** "I've started building your shoe store. It will be ready at store-xyz.urumi.app shortly."

7.3 Infrastructure-as-Code Generation

For advanced customization (e.g., "Make it a high-performance store"), the AI can be prompted to generate the *values.yaml* itself.[74] The Orchestrator exposes an endpoint that accepts a raw JSON/YAML configuration, validating it against a schema before passing it to Helm. This allows the AI to tune memory limits, replica counts, or plugin lists dynamically based on the user's natural language request.[76]

8. Scalability & Operational Trade-offs

8.1 Horizontal Scaling

- **Dashboard/API:** Stateless Node.js apps scale easily with replicas: 3 and a standard K8s

Service.[78]

- **Orchestrator:** Can be scaled, but requires leader election (e.g., using a text record in Kubernetes) to ensure two controllers don't try to provision the same store ID simultaneously.
- **Workloads:** WooCommerce scales via HPA, but requires ReadWriteMany storage for wp-content if running multiple replicas (e.g., using Azure Files or NFS).[14] The database scales vertically or via Read Replicas (complex for an internship, single StatefulSet is the correct tradeoff).[19]

8.2 Trade-offs

- **Helm vs. Operators:** Writing a full Kubernetes Operator (CRDs + Controller) using Kubebuilder is the "pure" K8s way but is complex and time-consuming.[79] The chosen approach (API + Helm SDK) acts as a "lite" operator. It achieves the same goal (automation) with significantly less boilerplate, fitting the 6-month internship constraint.
- **K3s vs. AKS:** K3s on a VPS is cheaper and sufficient for this test but requires manual management of etcd and upgrades.[4] AKS is production-standard but overkill/expensive for a test task. The architecture supports both, minimizing lock-in.

9. Conclusion

This design document presents a comprehensive, executable blueprint for the Urumi AI assessment. By leveraging the power of **Helm SDK** for programmatic control, **WP-CLI** for application-layer automation, and **Kubernetes Namespaces** for strict isolation, the platform meets every functional and non-functional requirement. The rigorous attention to **Azure integration** and **security hardening** directly addresses the "Ways to Stand Out" criteria. Furthermore, the API-first design ensures a seamless transition to Round 2, enabling Gen AI agents to drive the infrastructure as effectively as a human engineer. This architecture is not just a test submission; it is a scalable, modern platform engineering foundation.

Detailed Implementation Guide

1. Local Environment Setup (Kind/K3d)

To ensure the "local-to-prod" story holds up, we start with a local cluster that mimics the production network topology.

Command:

Bash

```
k3d cluster create urumi-local \
  --api-port 6443 \
  --port "80:80@loadbalancer" \
  --port "443:443@loadbalancer" \
  --k3s-arg "--disable=traefik@server:0"
```

Note: Disabling Traefik allows us to install Nginx Ingress manually, matching the production setup.[83]

Helm Setup:

Initialize the repo structure.

Bash

```
helm create charts/ecommerce-store
rm -rf charts/ecommerce-store/templates/* # Clean slate
```

2. The Orchestrator (Go Code Blueprint)

The heart of the system. Below is the logic flow for the ProvisionStore function using the Helm SDK.

2.1 Initialization

The orchestrator initializes the Helm action configuration using the cluster's REST client. This allows it to talk to K8s without shelling out to kubectl.[2]

Go

```
// Blueprint for initializing Helm SDK
settings := cli.New()
actionConfig := new(action.Configuration)
// Initialize with the namespace logic
```

```
if err := actionConfig.Init(settings.RESTClientGetter(), settings.Namespace(),
os.Getenv("HELM_DRIVER"), log.Printf); err!= nil {
    // Handle error
}
```

2.2 Installing the Chart

The install action is configured with the store-specific parameters.[3]

Go

```
client := action.NewInstall(actionConfig)
client.Namespace = "store-" + storeID
client.CreateNamespace = true // Critical for isolation [45]
client.ReleaseName = "store-" + storeID
client.Wait = true // Wait for Pods to be Ready [11]
client.Timeout = 5 * time.Minute

// Load the chart
chartObj, _ := loader.Load("/path/to/charts/ecommerce-store")

// Merge values (User input + Environment defaults)
vals := map[string]interface{}{
    "engine": "woocommerce",
    "storeName": "My Demo Store",
    "global": map[string]interface{}{
        "storageClass": "local-path", // "default" in Prod
    },
}

// Execute
release, err := client.Run(chartObj, vals)
```

3. Automation Scripts (The "Secret Sauce")

To pass the "Definition of Done," the store must be checkout-ready.

3.1 WooCommerce job-setup.yaml

This Job runs after the main deployment is ready. It mounts a ConfigMap containing the setup

script.

YAML

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: wp-setup
  annotations:
    "helm.sh/hook": post-install
spec:
  template:
    spec:
      containers:
      - name: wp-cli
        image: wordpress:cli
        command: ["/bin/bash", "/scripts/setup.sh"]
        volumeMounts:
        - name: web-root
          mountPath: /var/www/html
        - name: scripts
          mountPath: /scripts
```

3.2 The setup.sh Script

This script orchestrates the internal WordPress state.[49]

Bash

```bash
#!/bin/bash
# Wait for DB connection...
wp core install --url="http://${STORE_URL}" --title="${STORE_NAME}"
--admin_user="${ADMIN_USER}" --admin_password="${ADMIN_PASS}"
--admin_email="${ADMIN_EMAIL}"

# Install WooCommerce
wp plugin install woocommerce --activate
```

```
# SKIP THE WIZARD
wp option set woocommerce_task_list_hidden yes
wp option set woocommerce_onboarding_profile '{"skipped": true}' --format=json

# ENABLE COD [55]
wp option update woocommerce_cheque_settings '{"enabled":"yes", "title":"Cash on Delivery"}'

# CREATE DUMMY PRODUCT
wp wc product create --name="Test Sneaker" --type=simple --regular_price=50 --user=1
```

4. Security & Isolation Configurations

4.1 NetworkPolicy (Deny Default)

This YAML is injected into every store namespace.[22]

YAML

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

4.2 Allow DNS & Ingress

We must explicitly allow the store to talk to DNS and receive traffic from the Ingress Controller.[38]

YAML

```
apiVersion: networking.k8s.io/v1
```

```
kind: NetworkPolicy
metadata:
  name: allow-essential
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
  egress:
  - to:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: kube-system
    ports:
    - protocol: UDP
      port: 53
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: ingress-nginx
```

5. Monitoring & Observability

To satisfy the "Abuse Prevention" and "Observability" stand-out criteria, we implement a metrics middleware in the Node.js API.[1]

5.1 Prometheus Metrics

The Backend API exposes /metrics for Prometheus scraping.

- urumi_stores_total{status="ready"}: Gauge.
- urumi_provisioning_duration_seconds: Histogram.

5.2 Dashboard Integration

The React dashboard queries these metrics (or the backend state) to show:

1. **"System Health":** Green/Red indicator based on k3s API availability.
2. **"Provisioning Logs":** A streaming log window that tails the wp-setup job logs using the Kubernetes Websocket API, giving users real-time feedback (e.g., "Installing Database...", "Activating Plugins...").[1]

This document serves as the comprehensive roadmap for executing the internship

assessment with an architectural rigor that exceeds expectations.

Works cited

1. Urumi SDE Internship - Round 1.pdf
2. Introduction - Helm, accessed February 6, 2026, https://helm.sh/docs/sdk/gosdk/
3. Samples on kubernetes helm golang client - Stack Overflow, accessed February 6, 2026, https://stackoverflow.com/questions/45692719/samples-on-kubernetes-helm-golang-client
4. How to Set Up K3s Production Cluster - OneUptime, accessed February 6, 2026, https://oneuptime.com/blog/post/2026-01-26-k3s-production-cluster/view
5. Installing the Nginx Ingress Controller on K3S | by Alesson Viana - Medium, accessed February 6, 2026, https://medium.com/@alesson.viana/installing-the-nginx-ingress-controller-on-k3s-df2c68cae3c8
6. Designing APIs for LLM Apps: Build Scalable and AI-Ready Interfaces - Gravitee, accessed February 6, 2026, https://www.gravitee.io/blog/designing-apis-for-llm-apps
7. Automate AI Workflows with OpenAPI to Build LLM-Ready APIs - Gravitee, accessed February 6, 2026, https://www.gravitee.io/blog/ai-workflows-with-openapi-and-llm-apis
8. Raise the Level of Abstraction using Kubernetes Operators by Tom De Wolf - YouTube, accessed February 6, 2026, https://www.youtube.com/watch?v=XDRAukTJMWo
9. Should I wrap CLI or call the API ? : r/golang - Reddit, accessed February 6, 2026, https://www.reddit.com/r/golang/comments/1gtheiw/should_i_wrap_cli_or_call_the_api/
10. action package - helm.sh/helm/v3/pkg/action - Go Packages, accessed February 6, 2026, https://pkg.go.dev/helm.sh/helm/v3/pkg/action
11. Examples - Helm, accessed February 6, 2026, https://helm.sh/docs/sdk/examples/
12. Setup a Kubernetes Namespace and Roles Using Helm - Stack Overflow, accessed February 6, 2026, https://stackoverflow.com/questions/66286181/setup-a-kubernetes-namespace-and-roles-using-helm
13. Create Namespace Using Helm Templates | Baeldung on Ops, accessed February 6, 2026, https://www.baeldung.com/ops/helm-templates-create-namespace
14. Example: Deploying WordPress and MySQL with Persistent Volumes - Kubernetes, accessed February 6, 2026, https://kubernetes.io/docs/tutorials/stateful-application/mysql-wordpress-persistent-volume/
15. Deploy helm charts with Go lang, accessed February 6, 2026, https://medium.com/@varunrathod0045/deploy-helm-charts-with-go-lang-fa3b967af124
16. Using Helm SDK: Programmatically Managing Helm Deployments | by Sonali

Mishra, accessed February 6, 2026, https://medium.com/@sonali.mishra6998/using-helm-sdk-programmatically-managing-helm-deployments-8deeed2efefa
17. Deploying a kubernetes job via helm - Stack Overflow, accessed February 6, 2026, https://stackoverflow.com/questions/55458237/deploying-a-kubernetes-job-via-helm
18. Chart Hooks - Helm, accessed February 6, 2026, https://helm.sh/docs/topics/charts_hooks
19. How to Deploy WordPress on Kubernetes with Nginx and MySQL - DevOpsCube, accessed February 6, 2026, https://devopscube.com/deploy-wordpress-on-kubernetes/
20. 1.2.1. Install Medusa with Docker, accessed February 6, 2026, https://docs.medusajs.com/learn/installation/docker
21. Medusajs 2.0 + Storefront - Deploy - Railway, accessed February 6, 2026, https://railway.com/deploy/gkU-27
22. The Recommended default deny policy should allow DNS over TCP · Issue #8224 · projectcalico/calico - GitHub, accessed February 6, 2026, https://github.com/projectcalico/calico/issues/8224
23. Building a Secure and Scalable Multi-Tenancy Model on GKE | LiveRamp, accessed February 6, 2026, https://liveramp.com/blog/building-a-secure-and-scalable-multi-tenancy-model-on-gke
24. Enable a default deny policy for Kubernetes pods - Calico Documentation, accessed February 6, 2026, https://docs.tigera.io/calico/latest/network-policy/get-started/kubernetes-default-deny
25. Simplify Amazon EKS multi-tenant application deployment by using Flux, accessed February 6, 2026, https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/simplify-amazon-eks-multi-tenant-application-deployment-by-using-flux.html
26. What are your best practices deploying helm charts? : r/kubernetes - Reddit, accessed February 6, 2026, https://www.reddit.com/r/kubernetes/comments/1jn7lwo/what_are_your_best_practices_deploying_helm_charts/
27. Volumes and Storage - K3s - Lightweight Kubernetes, accessed February 6, 2026, https://docs.k3s.io/add-ons/storage
28. Bare Metal Storage choices for a homelab - kubernetes - Reddit, accessed February 6, 2026, https://www.reddit.com/r/kubernetes/comments/18njhpk/bare_metal_storage_choices_for_a_homelab/
29. Concepts - Storage in Azure Kubernetes Services (AKS) - Microsoft Learn, accessed February 6, 2026, https://learn.microsoft.com/en-us/azure/aks/concepts-storage
30. Built-in Objects - Helm, accessed February 6, 2026,

https://helm.sh/docs/chart_template_guide/builtin_objects
31. Securing NGINX-ingress - cert-manager Documentation, accessed February 6, 2026, https://cert-manager.io/docs/tutorials/acme/nginx-ingress/
32. Using Helm with Kubernetes: A Guide to Helm Charts and Their Implementation, accessed February 6, 2026, https://dev.to/alexmercedcoder/using-helm-with-kubernetes-a-guide-to-helm-charts-and-their-implementation-8dg
33. Traefik vs. NGINX: Comparison and Practical Guide - Cast AI, accessed February 6, 2026, https://cast.ai/blog/traefik-vs-nginx/
34. How to Enable Let's Encrypt TLS in K3s with Traefik - DEV Community, accessed February 6, 2026, https://dev.to/giveitatry/how-to-enable-lets-encrypt-tls-in-k3s-with-traefik-282n
35. Kubernetes Ingress Controllers Explained: NGINX vs Traefik vs HAProxy (2025 Edition), accessed February 6, 2026, https://medium.com/@canaldoagdias/kubernetes-ingress-controllers-explained-nginx-vs-traefik-vs-haproxy-2025-edition-6e288e3f7d1a
36. K3S: understanding the network model and applying TLS certificates - Server Fault, accessed February 6, 2026, https://serverfault.com/questions/1156016/k3s-understanding-the-network-model-and-applying-tls-certificates
37. Calico vs. Cilium: Which Kubernetes CNI Is Right for You? - Zesty, accessed February 6, 2026, https://zesty.co/finops-glossary/calico-vs-cilium-in-kubernetes-networking/
38. Kubernetes - Network isolation with NetworkPolicy - Qovery, accessed February 6, 2026, https://www.qovery.com/blog/basic-network-isolation-in-kubernetes
39. Use service rules in policy - Calico Documentation, accessed February 6, 2026, https://docs.tigera.io/calico/latest/network-policy/policy-rules/service-policy
40. Add Network Policy for DNS - Kyverno, accessed February 6, 2026, https://kyverno.io/policies/best-practices/add-networkpolicy-dns/add-networkpolicy-dns/
41. Namespace isolation and access controls - Calico Documentation, accessed February 6, 2026, https://docs.tigera.io/calico-cloud/tutorials/enterprise-security/namespace-isolation
42. Charts - Helm, accessed February 6, 2026, https://helm.sh/docs/topics/charts/
43. Building a Helm Chart for Multi-App Deployments | by Young Gyu Kim | Dec, 2025 | Medium, accessed February 6, 2026, https://medium.com/@nsalexamy/building-a-helm-chart-for-multi-app-deployments-ffbd3593be50
44. helm install, accessed February 6, 2026, https://helm.sh/docs/helm/helm_install/
45. How to create a namespace if it doesn't exists from HELM templates? - Stack Overflow, accessed February 6, 2026, https://stackoverflow.com/questions/51783651/how-to-create-a-namespace-if-it-doesnt-exists-from-helm-templates
46. How to update Helm Chart values via go programmatically · Issue #11209 -

GitHub, accessed February 6, 2026, https://github.com/helm/helm/issues/11209
47. WP-CLI and WooCommerce setup | WordPress.org, accessed February 6, 2026, https://wordpress.org/support/topic/wp-cli-and-woocommerce-setup/
48. Using Wordpress CLI image on Kubernetes - Stack Overflow, accessed February 6, 2026, https://stackoverflow.com/questions/48623764/using-wordpress-cli-image-on-kubernetes
49. Using wp-cli with Docker. Earlier this week, I posted a tutorial... | by A. Tate Barber | Medium, accessed February 6, 2026, https://medium.com/@tatemz/using-wp-cli-with-docker-21b0ab9fab79
50. A practical guide to using WP-Cli to install and manage WordPress - YouTube, accessed February 6, 2026, https://www.youtube.com/watch?v=JJbL-wOBisM
51. Using WP-CLI to Speed Up Your WooCommerce Development - Robot Ninja, accessed February 6, 2026, https://robotninja.com/blog/wp-cli-woocommerce-development/index.html
52. The Complete Guide to Installing WP-CLI - Delicious Brains, accessed February 6, 2026, https://deliciousbrains.com/complete-guide-to-installing-wp-cli/
53. How to Manage WordPress and WooCommerce with WP-CLI - Servebolt, accessed February 6, 2026, https://servebolt.com/help/technical-resources/manage-wordpress-and-woocommerce-via-ssh-with-wp-cli/
54. Disable WooCommerce Setup Wizard - Random Adult, accessed February 6, 2026, https://randomadult.com/disable-woocommerce-setup-wizard/
55. Cash on Delivery Documentation - WooCommerce, accessed February 6, 2026, https://woocommerce.com/document/cash-on-delivery/
56. How to Add Cash on Delivery Payment Method in WooCommerce for Free? - YouTube, accessed February 6, 2026, https://www.youtube.com/watch?v=AAbb3Nms5_k
57. How to Use WooCommerce CLI, accessed February 6, 2026, https://developer.woocommerce.com/docs/wc-cli/using-wc-cli/
58. 1.2. Install Medusa, accessed February 6, 2026, https://docs.medusajs.com/learn/installation
59. Connection timeout port 80 on new Azure VM with NSG rules configured - Stack Overflow, accessed February 6, 2026, https://stackoverflow.com/questions/33133448/connection-timeout-port-80-on-new-azure-vm-with-nsg-rules-configured
60. Install on Virtual Machine (K3s) - Atlan Documentation, accessed February 6, 2026, https://docs.atlan.com/secure-agent/how-tos/k3s/install-secure-agent-on-virtual-machine-k3s
61. Prepare your Kubernetes cluster - Azure IoT Operations - Microsoft Learn, accessed February 6, 2026, https://learn.microsoft.com/en-us/azure/iot-operations/deploy-iot-ops/howto-prepare-cluster
62. cloud-init support for virtual machines in Azure - Microsoft Learn, accessed

February 6, 2026, https://learn.microsoft.com/en-us/azure/virtual-machines/linux/using-cloud-init
63. Tutorial - How to use cloud-init to customize a Linux virtual machine in Azure on first boot, accessed February 6, 2026, https://learn.microsoft.com/en-us/azure/virtual-machines/linux/tutorial-automate-vm-deployment
64. How to access k3s cluster created in an azure vm with public and private ip? - Server Fault, accessed February 6, 2026, https://serverfault.com/questions/1142236/how-to-access-k3s-cluster-created-in-an-azure-vm-with-public-and-private-ip
65. Quickstart for Calico on K3s, accessed February 6, 2026, https://docs.tigera.io/calico/latest/getting-started/kubernetes/k3s/quickstart
66. Deploy cert-manager on Azure Kubernetes Service (AKS) and use Let's Encrypt to sign a certificate for an HTTPS website, accessed February 6, 2026, https://cert-manager.io/docs/tutorials/getting-started-aks-letsencrypt/
67. Manage SSL certificates and ingress for services in k3s kubernetes cluster using cert-manager, letsencrypt and traefik | Hashbang, accessed February 6, 2026, https://hashbang.nl/blog/manage-ssl-certificates-and-ingress-for-services-in-k3s-kubernetes-cluster-using-cert-manager-letsencrypt-and-traefik
68. Installing Cert-Manager and NGINX Ingress with Let's Encrypt on Kubernetes, accessed February 6, 2026, https://hbayraktar.medium.com/installing-cert-manager-and-nginx-ingress-with-lets-encrypt-on-kubernetes-fe0dff4b1924
69. HTTPS with Cert-Manager and Letsencrypt - K3S Rocks, accessed February 6, 2026, https://k3s.rocks/https-cert-manager-letsencrypt/
70. Using Nginx Ingress Controller and Cert-Manager for HTTPS with Let's Encrypt, accessed February 6, 2026, https://dev.to/hkhelil/using-nginx-ingress-controller-and-cert-manager-for-https-with-lets-encrypt-2flh
71. Deploy a WordPress Application on a Kubernetes Cluster using Ingress, Cert-Manager, and Helm. | by Ezekiel Umesi | Medium, accessed February 6, 2026, https://medium.com/@ezekiel.umesi/deploy-a-wordpress-application-on-a-kubernetes-cluster-using-ingress-cert-manager-and-helm-1f3b34356197
72. Making REST APIs Agent-Ready: From OpenAPI to Model Context Protocol Servers for Tool-Augmented LLMs - arXiv, accessed February 6, 2026, https://arxiv.org/html/2507.16044v1
73. mgorav/APIAide: LLM REST APIs Orchestration - GitHub, accessed February 6, 2026, https://github.com/mgorav/APIAide
74. Analyzing and Mitigating (with LLMs) the Security Misconfigurations of Helm Charts from Artifact Hub - arXiv, accessed February 6, 2026, https://arxiv.org/html/2403.09537v1
75. An Engineer's Exploration of LLMs for Infrastructure Provisioning - StackGen, accessed February 6, 2026, https://stackgen.com/blog/engineers-exploration-of-llms-for-infrastructure-provi

sioning
76. Helm Chart Usage Explained | Simplyblock, accessed February 6, 2026, https://www.simplyblock.io/glossary/what-is-a-helm-chart/
77. Helm Charts 101: Templating Your AI Infrastructure Like a Pro | by Pranav Prakash I GenAI I AI/ML I DevOps I | Medium, accessed February 6, 2026, https://medium.com/@pranavprakash4777/helm-charts-101-templating-your-ai-infrastructure-like-a-pro-32a04de34cb8
78. Top 46 Kubernetes Interview Questions and Answers in 2026 - DataCamp, accessed February 6, 2026, https://www.datacamp.com/blog/kubernetes-interview-questions
79. CNCF Operator White Paper, accessed February 6, 2026, https://tag-app-delivery.cncf.io/whitepapers/operator/
80. What is a Kubernetes operator? - Red Hat, accessed February 6, 2026, https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-operator
81. What Are Kubernetes Operators, and Do You Still Need Them in 2025? - Syntasso, accessed February 6, 2026, https://www.syntasso.io/post/what-are-kubernetes-operators-and-do-you-still-need-them-in-2025
82. k3s in a vm, not in aks. How the app gateway will be linked to it and use lets encrypt certificate there | by intruder | Medium, accessed February 6, 2026, https://medium.com/@intruder2021/k3s-in-a-vm-not-in-aks-2348aa9febdf
83. Setup Traefik on Kubernetes, accessed February 6, 2026, https://doc.traefik.io/traefik/setup/kubernetes/
84. Kubernetes DNS and Network Policies: Secure Service Communication in Your Cluster, accessed February 6, 2026, https://jay75chauhan.medium.com/kubernetes-dns-and-network-policies-secure-service-communication-in-your-cluster-1ee2e378f7a6