ChatGPT

Store Provisioning Platform Architecture

The proposed platform has a **web UI (Next.js)** for users to request new stores, a **FastAPI backend** that orchestrates provisioning, and a **Kubernetes cluster** where each WooCommerce store is deployed. Each store runs in its *own namespace* with its own WordPress+WooCommerce deployment and a dedicated MySQL database (see diagram below). The FastAPI service uses the Kubernetes API (via Helm) to create namespaces and resources. An ingress controller (Traefik, Nginx, etc.) routes public domains (e.g. `store1.example.com`) to the correct namespace's WordPress Service, with TLS certificates managed by **cert-manager**. PersistentVolumes back the MySQL data and WordPress files; secrets store DB and admin credentials. The system enforces *idempotent* provisioning: if a store already exists, re-running "create" is a no-op or update. Kubernetes' self-healing ensures pods restart on failure, and metrics/auto-scaling handle load (e.g. scale out Traefik if needed) [1] [2].

Resource	Purpose/Use
Namespace	Per-store isolation. All resources for one store live in its namespace (tenant-per-namespace) [3].
Deployment/StatefulSet	Run WordPress+PHP (Deployment) and MySQL (Deployment or StatefulSet). Each has one replica (or HA DB if needed).
Service (ClusterIP)	Exposes WordPress to ingress; exposes MySQL only internally (WordPress connects to `mysql-<release>` Service).
Ingress	Routes external HTTP(S) to WordPress Services, based on hostnames/subpaths. Uses TLS (with `cert-manager` issuing Let's Encrypt certificates for each store's domain [4]).
PVC (PersistentVolumeClaim)	Back MySQL data (`/var/lib/mysql`) and WordPress uploads (`/var/www/html`). For example, one sample deployed a 20Gi volume for MySQL and 30Gi for WordPress [5] [6].
Secret	Stores sensitive values (DB root/user passwords, WordPress salts, admin creds). Injected as env vars into containers [7] [8].
ConfigMap	(Optional) Static config data (e.g. `wp-config.php` snippets, plugin settings).
Job/CronJob	For one-time or scheduled tasks (e.g. initialize WP plugins via WP-CLI, run DB backups, cleanup expired orders).
ResourceQuota	Limits total CPU/memory/storage in the namespace so a single store can't hog cluster resources. Kubernetes docs recommend quotas per namespace to prevent "noisy neighbor" issues [9].

Resource	Purpose/Use
LimitRange	Sets default and max CPU/memory per pod/container in each namespace, enforcing that ResourceQuota can be applied.
RBAC (Role, RoleBinding)	The orchestrator runs with a service account granted rights to create namespaces, deployments, etc., but limited so no user can escape their namespace. Follow least-privilege RBAC (only allow the FastAPI service account to modify tenant namespaces) [10].
NetworkPolicy	By default, block cross-namespace pod communication and allow only within each namespace (plus DNS). In a multi-tenant cluster, it's recommended to start with a "default deny" policy and only allow intra-namespace traffic [11].

[5] [6] *Figure: Example PersistentVolume definitions for MySQL (20Gi) and WordPress (30Gi) from a Kubernetes deployment guide* [5] [6].

WooCommerce on Kubernetes

Each store is a **WordPress + WooCommerce** site. WooCommerce runs as a WordPress plugin and stores its data in the WordPress database (MySQL). It fully integrates with the WordPress REST API, so products, orders, customers, etc. can be accessed/managed via HTTP JSON endpoints [12]. In practice, the platform installs WordPress (using a container image like `wordpress:php-fpm`), then activates WooCommerce. WordPress connects to the MySQL service in the same namespace via environment variables (`WORDPRESS_DB_HOST`, etc.), which are populated from Kubernetes Secrets [7]. The MySQL root/password, WP salt keys, and database name/user are generated per-store and kept in a Secret (never hardcoded).

The WordPress pod should run as a non-root user (e.g. `www-data`), and its container can mount two volumes: one for the WordPress code/uploads (`/var/www/html`) and one for connecting to MySQL. In the sample above, `/var/www/html` is on a 30Gi PVC [6] so that plugin uploads (images, etc.) persist. MySQL's `/var/lib/mysql` is on a 20Gi PVC [5]. Both use ReadWriteOnce access modes. This ensures data is durable.

Readiness and liveness probes ensure traffic only goes to a healthy container. For WordPress, a simple HTTP GET to "/index.php" or "/wp-login.php" often suffices. For example, one case showed configuring:

```
readinessProbe:
  httpGet: { path: "/index.php", port: 80 }
  initialDelaySeconds: 15
  failureThreshold: 5
livenessProbe:
  httpGet: { path: "/index.php", port: 80 }
```

```
  initialDelaySeconds: 10
  failureThreshold: 5
```

This waits for WordPress to be fully up before marking the pod ready [13]. If probes fail repeatedly, K8s restarts the pod. (Care must be taken: WordPress's canonical redirects may cause probes to get 301; one can disable that or probe a static file.)

After deploy, WordPress must be configured (set admin user, etc.). This can be automated with a Kubernetes Job or a Helm post-install hook using `wp-cli`. For example, the backend could run a Job to create the WooCommerce admin account and initialize settings. The admin credentials (user, pass) are then stored in a Secret and returned to the user via the FastAPI API.

Ingress rules expose each store. In production, we recommend each store have its own domain or subdomain (e.g. `store1.example.com`). Using a real public DNS name allows **cert-manager** to automatically provision Let's Encrypt TLS certificates for HTTPS. For example, an Ingress manifest might look like the tutorial's example [4]:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: wordpress
  namespace: store1
  annotations:
    kubernetes.io/ingress.class: nginx
    cert-manager.io/cluster-issuer: letsencrypt-production
spec:
  rules:
  - host: store1.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: wordpress
            port: { number: 80 }
  tls:
  - hosts: [store1.example.com]
    secretName: store1-tls
```

This uses a ClusterIssuer (e.g. DNS or HTTP challenge) to keep `store1.example.com` certified [4]. The tutorial notes: "We strongly recommend a real DNS name... this will let you use Let's Encrypt" [4]. The same cert-manager setup can issue wildcards or individual certs as needed.

Helm Chart and Tenant Templates

We recommend creating a **Helm chart** for a store deployment, with templated parameters so one chart can instantiate any number of stores. A tenant chart might include templates like:

```
tenant-chart/
├── Chart.yaml
├── values.yaml
└── templates/
    ├── namespace.yaml
    ├── resourcequota.yaml
    ├── limitrange.yaml
    ├── networkpolicy.yaml
    ├── rbac.yaml
    ├── serviceaccount.yaml
    ├── secrets.yaml
    ├── pvc.yaml
    ├── mysql-deployment.yaml
    ├── wordpress-deployment.yaml
    └── ingress.yaml
```

This is inspired by multi-tenant Helm patterns [3]. The `namespace.yaml` template creates the namespace (e.g. "store-$CUSTOMER"). The `resourcequota.yaml` and `limitrange.yaml` set quotas on CPU/memory/pods etc. (values injected from `values.yaml`). `networkpolicy.yaml` can define a default-deny plus allow rules within the namespace. `rbac.yaml` and `serviceaccount.yaml` can create a namespace-scoped role if needed (e.g. allow read-only roles). `secrets.yaml` generates the DB credentials (unless supplied in values). The MySQL and WordPress deployments use these secrets for env vars. Finally, `ingress.yaml` uses the store's domain from values.

By using Helm, provisioning is **idempotent**: running `helm upgrade --install store1 ...` will either create or update the store without duplicating resources. The FastAPI backend can call Helm via subprocess or a Kubernetes operator to manage releases. This allows versioned upgrades or rollbacks (e.g. `helm rollback store1` if a plugin update breaks the site).

Kubernetes Resources and Best Practices

Each store's namespace will consume the Kubernetes resources listed above. We enforce **ResourceQuotas** per namespace so no store exceeds its share. The official docs note: "By mapping tenants to namespaces, cluster admins can use quotas to ensure a tenant cannot monopolize a cluster's resources" [14]. For example, one might set a quota of `requests.cpu=1`, `requests.memory=2Gi`, `pods=10`, `persistentvolumeclaims=2`, etc. In conjunction, **LimitRanges** ensure each pod/container sets reasonable CPU/memory limits, or they cannot be created at all. Always set quotas and limits for production namespaces [10].

Network policies isolate tenants: by default we should deny all cross-namespace traffic and only allow intra-namespace (and DNS). Kubernetes docs recommend a default-deny policy, then add a rule to allow pods to talk to the DNS server, and finally open intra-namespace communications [11]. For example, a simple NetworkPolicy in each store-namespace could allow only traffic to ports 80/443 on the namespace's labels. This prevents one store's pod from accidentally (or maliciously) talking to another's.

We also practice **least-privilege RBAC**: only the FastAPI orchestrator's ServiceAccount needs rights to create/modify namespaces and objects. Normal users never get raw kubeconfig access. Within each namespace, RBAC can lock down who can list pods or services. For audit trails, Kubernetes' audit logging can record every API action (who created which namespace, etc.) [15]. Additionally, the application should log all provisioning actions (in a database or log store) with the user's identity and timestamp.

Observability (Logging & Metrics)

Multi-tenant observability is crucial. We propose aggregating logs and metrics by namespace (store). For **metrics**, run a Prometheus server that scrapes relevant endpoints and Kubernetes. Use relabeling so that each metric has a `namespace` label [16]. For example, Prometheus `scrape_config` might take the pod's namespace meta-label and map it to `namespace`. This ensures all metrics (store creation time, number of orders placed, PHP errors, etc.) can be queried per-store.

For **logs**, deploy a logging agent (e.g. Fluent Bit) that adds Kubernetes metadata to each log line. Fluent Bit's Kubernetes filter can attach `{namespace, pod_name, container_name}` labels to logs [17]. These logs (WordPress errors, DB logs, provisioning logs) can be sent to a central store (e.g. Loki, Elasticsearch). In Grafana or Kibana, queries can then be filtered like `namespace="store1"` so each store's team only sees their own logs [17]. As one engineer notes, "Each layer respects namespace boundaries: every log line is enriched with namespace, and dashboards/alerts are scoped by namespace" [18] [17]. We should also instrument the FastAPI backend with Prometheus (e.g. client library) to emit metrics like provisioning latency or error counts, again labeled by user or namespace.

FastAPI Backend API

The FastAPI backend exposes a RESTful API for managing stores. Key endpoints include:

- `POST /stores`: **Create** a new store. The request includes user ID, store name (slug), optional configuration (theme, plugins). The backend checks the user's quota, then generates a unique namespace and Helm release name. It writes a record to its own database (e.g. Postgres) for tracking, then asynchronously invokes Helm to install the chart. The response returns a provisional store ID and status ("provisioning"). If the store already exists (same name), the call is idempotent (returns the existing store info).
- `GET /stores`: **List** all stores for the authenticated user, with statuses (Provisioning, Ready, Error).
- `GET /stores/{id}`: **Get details** (domain, status, creation time, last error).
- `DELETE /stores/{id}`: **Delete** a store. This triggers `helm delete` and deletes the namespace. The action is logged and the record removed.
- `GET /stores/{id}/health`: **Health check** (optionally calls into K8s to verify pods are Running and probes are passing).

All endpoints require authentication (e.g. JWT or API token) and enforce per-user isolation (a user can only see or delete their own stores). FastAPI's dependency injection can handle auth and database sessions. Long-running ops (like `helm install`) can be done in a background thread or queued worker; the client can poll the store's status or a WebSocket endpoint for real-time updates.

API design follows standard REST conventions (use 201 Created on success, 202 Accepted for async start, 4xx for client errors, 5xx for cluster errors). FastAPI/Pydantic models define the request/response schemas. Using HTTPS and proper CORS is assumed.

Next.js Frontend

The React/Next.js frontend provides a **dashboard** for users. After login, a user sees a list of their stores and statuses (Pending, Ready, Error). They can fill out a "Create Store" form (choosing name, domain suffix, etc.). Upon submitting, the UI calls `POST /stores` and shows the new entry as "Provisioning...". The frontend can use **WebSockets** or **Server-Sent Events** to get live updates from the backend (FastAPI supports WebSockets). For example, the backend might send a message when the Helm install succeeds or fails, and the UI updates the status badge. If WebSockets are too complex, the UI can poll `GET /stores/{id}` every few seconds until status changes.

Once a store is Ready, the UI shows its URL (e.g. `https://store1.example.com`) and possibly WP admin credentials (or triggers an email with login info). The dashboard may also integrate basic logs or metrics: e.g. a panel showing "Recent Logins" or "5 recent orders" by calling the WooCommerce REST API directly (using the store's API keys). Charts can be done via libraries (Chart.js, etc.) to show real-time order count, sales, or health metrics (PHP-FPM usage, DB connections) scraped from Prometheus.

Overall, the UI is a single-page app with a table of stores and a form/modal for creation. It uses Next.js's SSR or API routes as needed for SEO or auth. UI components should handle errors (e.g. show if store creation fails, or if the Helm release times out).

Security Considerations

- **Containers as non-root**: Use containers that run as a non-root user. In Kubernetes specs, set `securityContext.runAsNonRoot: true` and a non-zero `runAsUser`. (Bitnami's images and official WordPress support non-root modes.) This limits container privileges even if compromised.
- **Pod Security**: Enforce Pod Security Admission (or policies) at the cluster level for "restricted" mode (no privileged, no hostNetwork, readOnlyRootFilesystem, drop capabilities). This ensures tenants' pods are sandboxed.
- **Network Policies**: As noted, implement default-deny so pods can't reach outside their namespace (blocks things like SSH or unauthorized DB access). Only allow the Ingress controller to reach web pods on port 80/443.
- **Ingress Security**: Use HTTPS everywhere. Apply HTTP security headers via the ingress (e.g. HSTS). Cert-manager automates certificate renewals, so no expired certs.
- **FastAPI Security**: Validate all input with Pydantic. Use HTTPS/TLS for API endpoints. Implement rate-limiting (e.g. FastAPI middleware or ingress annotations) to prevent abuse (e.g. throttle requests by IP or user). Given our architecture, rate-limiting is recommended at the ingress level (Nginx Ingress

Controller supports annotations like `limit-rpm` for requests per minute) or via a gateway/CDN in front of the FastAPI service.
- **Least Privilege**: FastAPI's Kubernetes client should use a K8s ServiceAccount with only the needed permissions (create/delete in each store namespace). Do not give cluster-admin to arbitrary components. Use RoleBindings in each namespace if users ever need direct K8s access (unlikely here).
- **Secrets Management**: Store all passwords/keys in Kubernetes Secrets (which are base64 but can also be encrypted at rest with tools like `kubernetes.io/encryption`). Do *not* hardcode any sensitive data in Helm charts or image; read them from `values.yaml` or generate. For higher security, tools like Sealed Secrets or external secrets managers (HashiCorp Vault, AWS Secrets Manager) can be integrated, but the basic use of K8s Secrets (and limiting who can read them via RBAC) is required.

Scalability and Concurrency

The orchestrator (FastAPI) is stateless and can scale horizontally behind a service load balancer (e.g. Kubernetes Service). It can simply run multiple replicas; they coordinate via the backing database or a locking mechanism. For Helm operations, however, caution is needed: running two Helm installs in parallel on the same cluster can conflict. One approach is to queue provisioning requests and process them sequentially (or only allow one install per namespace name). Alternatively, each store uses a distinct namespace, so the main conflict is cluster-level resources (not many aside from ingress/controller).

If provisioning is slow, FastAPI should not block the HTTP request: instead, start a background task or Kubernetes Job. For example, on `POST /stores`, FastAPI enqueues a Celery task that runs `helm install`, and immediately returns "202 Accepted". The UI then polls or uses events to follow progress. If the Helm install hangs or fails, the orchestrator catches that and marks the store's status as Error (with logs).

Kubernetes itself can scale the WordPress and MySQL pods (HorizontalPodAutoscaler) if needed under load. For the cluster, if using k3s on a VPS, one can add more worker nodes or control-plane nodes for HA. Traefik (k3s's default ingress) can be scaled by adding replicas. The underlying VPS could use a cloud disk or attached storage class for PVCs.

Because multiple users may provision concurrently, implement **per-user quotas**. For example, in the FastAPI user database, track how many stores each user has, and limit it (e.g. max 5 stores). If they hit the limit, `POST /stores` returns 403. This is separate from the Kubernetes ResourceQuota. Also consider throttling: e.g. allow only 1 store creation per minute per user to avoid sudden spikes.

Production Considerations

For production on k3s (or any cluster):

- **Ingress on VPS**: If using k3s's bundled Traefik, configure a static public IP or DNS to point to the VPS. Alternatively, use a bare-metal Ingress with MetalLB or NodePort+DNS.
- **DNS and cert-manager**: Use either a wildcard DNS (e.g. `*.example.com` A-record to cluster IP) or dynamically create A/CNAME records for each store subdomain. Tools like ExternalDNS can

automate DNS entry creation using your DNS provider API. Cert-manager can use either HTTP-01 (if Ingress is reachable on port 80) or DNS-01 (for wildcard certs) to obtain certs for each store domain [4]. For many stores, a wildcard cert (via DNS challenge) might simplify things, or use Let's Encrypt's SAN limits carefully.

- **Rollbacks/Upgrades**: Because each store is a Helm release, upgrades (e.g. to a new WordPress image version or plugin version) can be done with `helm upgrade`. If something goes wrong, `helm rollback <revision>` can revert. The FastAPI may expose an "Update Store" action or automatically patch Helm chart values (e.g. PHP version). Ensure Helm history is kept (secrets + configmaps) for each release.
- **Backups**: (Not explicitly asked but worth mentioning) Back up MySQL volumes regularly (e.g. using Velero or CronJobs to dump to object storage) so stores aren't lost on VPS failure. WordPress content (media) is on PVC too, so back that up or use ReadWriteMany NFS for easier snapshot.
- **Monitoring**: Besides Prometheus, watch cluster health: ensure etcd is healthy (if HA), k3s services are running, etc. The LiteSpeed tutorial notes that Kubernetes will auto-restart failed pods [2], which helps reliability.
- **Performance**: Cache static assets (images, JS) either on a CDN or via the ingress. Consider adding a caching layer (Redis or LiteSpeed Cache plugin) inside WordPress if stores get heavy traffic. The base design is scale-out: you can add more CPU/memory to nodes or add nodes.

References

This design follows established Kubernetes best practices for multi-tenancy and WordPress deployment. For example, multi-tenant Helm patterns use **one namespace per tenant** with quotas, network policies, and namespace-level RBAC [3] [10]. A Kubernetes tutorial for WordPress/WooCommerce illustrates a similar setup: it uses Secrets for DB credentials, PersistentVolumes for MySQL and WordPress storage, and an Ingress + cert-manager for external access [5] [4]. Applying those principles ensures each store is isolated, secure, and manageable.

Sources: Kubernetes docs and community guides on multi-tenant Helm charts [3] [10]; WooCommerce REST API documentation [12]; a Kubernetes WordPress tutorial [5] [6]; a WooCommerce+K8s blog [1] [4]; and multi-tenant observability patterns [18] [17].

---

[1] [2] [4] [5] [6] [7] [8] WordPress, WooCommerce, and Kubernetes ⋆ LiteSpeed Blog
https://blog.litespeedtech.com/2022/09/26/wordpress-woocommerce-and-kubernetes-with-litespeed-ingress-controller/

[3] [10] Multi-Tenancy Patterns with Helm
https://oneuptime.com/blog/post/2026-01-17-helm-kubernetes-multi-tenancy-patterns/view

[9] [11] [14] Multi-tenancy | Kubernetes
https://kubernetes.io/docs/concepts/security/multi-tenancy/

[12] WooCommerce REST API Documentation - WooCommerce
https://woocommerce.com/document/woocommerce-rest-api/

[13] wordpress - Kubernetes HTTP liveness probe fails with "connection refused" even though URL works without it - Stack Overflow
https://stackoverflow.com/questions/59280829/kubernetes-http-liveness-probe-fails-with-connection-refused-even-though-url-w

15 Auditing | Kubernetes
https://kubernetes.io/docs/tasks/debug/debug-cluster/audit/

16 17 18 Metrics Without Noise: How I Architected Multi-Tenant Observability in Kubernetes | by Gokul Srinivas | Medium
https://medium.com/@gokulsrinivas.b/metrics-without-noise-how-i-architected-multi-tenant-observability-in-kubernetes-0a7a7cb53576