ADR-011: Kubernetes Secrets for Secret Management

Submitters

- Luke Doyle (D00255656)
- Hafsa Moin (D00256764)

Change Log

- approved 2026-02-14

Referenced Use Case(s)

- UVote Secret Management Requirements - Secure injection of database credentials, JWT signing keys, and encryption keys across six microservices in the UVote Kubernetes cluster.

Context

The UVote platform requires a mechanism for managing sensitive configuration data across six microservices running in a Kubernetes cluster. Database credentials, JWT signing keys, API tokens, and encryption keys must be injected into pods at runtime without being hardcoded in source code, Docker images, or Kubernetes manifests committed to version control.

During early prototyping, secrets were embedded directly in Python source files and Docker environment variables defined in `docker-compose.yml`. This approach was identified as a critical vulnerability: credentials were visible in plaintext in Git history, Docker image layers, and container inspection output. The team evaluated four approaches to replace this, ranging from simple environment variable injection to dedicated secret management platforms.

The project runs on a local Kind cluster with an ephemeral lifecycle: the cluster is destroyed and recreated regularly during development via `deploy_platform.py`. There is no cloud provider KMS or managed secret store available, and the cluster is not exposed to external networks.

Proposed Design

Services and modules impacted:

All six microservices consume secrets as environment variables via `secretKeyRef` references in their Deployment manifests. The `deploy_platform.py` script creates all Secret objects during cluster bootstrapping, before deploying application pods. Application code reads secrets from environment variables via `os.environ` with no Kubernetes-specific code required.

Secret creation (`deploy_platform.py`):

```
# Database credentials for each service
kubectl create secret generic auth-db-secret \
  --from-literal=DB_USER=auth_user \
  --from-literal=DB_PASSWORD=$(openssl rand -hex 16) \
  --from-literal=DB_NAME=uvote \
  --from-literal=DB_HOST=postgres-service \
  -n uvote

# JWT signing key for auth-service
kubectl create secret generic jwt-secret \
  --from-literal=JWT_SECRET_KEY=$(openssl rand -hex 32) \
  --from-literal=JWT_ALGORITHM=HS256 \
  -n uvote

# Encryption key for voting-service
kubectl create secret generic voting-encryption-secret \
  --from-literal=ENCRYPTION_KEY=$(openssl rand -hex 32) \
  -n uvote

```

Secret consumption in Deployment manifests:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-service
spec:
  template:
    spec:
      containers:
        - name: auth-service
          env:
            - name: DB_USER
              valueFrom:
                secretKeyRef:
                  name: auth-db-secret
                  key: DB_USER
            - name: DB_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: auth-db-secret
                  key: DB_PASSWORD
            - name: JWT_SECRET_KEY
              valueFrom:
                secretKeyRef:
                  name: jwt-secret
                  key: JWT_SECRET_KEY

```

Least-privilege secret mounting:

Each Deployment references only the specific Secret keys required by that service. The auth-service receives JWT keys and its own database credentials. The voting-service receives database credentials and the encryption key. The results-service receives read-only database credentials only. No service receives secrets it does not need.

Secret rotation:

Fresh secrets are generated by `openssl rand` on every `deploy_platform.py` run. Because the cluster is ephemeral, natural rotation occurs on every cluster recreation without requiring an explicit rotation mechanism.

Integration points:

- `deploy_platform.py`: creates all Secret objects during cluster setup
- Deployment manifests: reference secrets via `secretKeyRef` in `env` blocks
- Application code: reads from environment variables via `os.environ`
- ADR-010 (Zero-Trust): namespace-scoped Secrets complement network policy isolation

Considerations

Hardcoded environment variables (not selected): Defining secrets directly in Kubernetes Deployment manifests or `docker-compose.yml` as plaintext values requires no additional tooling and is immediately visible for debugging. It was not selected because secrets stored in version-controlled YAML are visible in Git history permanently, even if later removed, and anyone with repository access can read all credentials. This was the approach used during early prototyping and is what this ADR replaces.

HashiCorp Vault (not selected): Vault provides encryption at rest, dynamic secrets with automatic rotation and TTL, comprehensive audit logging, and fine-grained access policies. It is the industry standard for production secret management. It was not selected for Stage 1 because the Vault server alone requires 256 to 512MB of RAM, the Vault Agent Injector adds a sidecar to every pod, and the initial bootstrapping and unsealing process adds significant operational complexity. A migration to Vault is planned for Stage 2. Application code will not need to change for this migration, as secrets will continue to be consumed as environment variables.

Bitnami Sealed Secrets (not selected): Sealed Secrets encrypts Kubernetes Secrets client-side using a public key, producing a `SealedSecret` resource that is safe to commit to Git. It was not selected because the ephemeral Kind cluster lifecycle creates a key management problem: the controller's private key changes on every cluster recreation, requiring all secrets to be re-sealed each time. Since secrets are generated programmatically by `deploy_platform.py` on every deployment and are never committed to Git, the Git-safe storage benefit does not apply here.

Base64 encoding is not encryption: Kubernetes Secrets store values as base64-encoded strings in etcd, not encrypted. Anyone with etcd access can decode them. For a local Kind cluster that is not exposed to external networks, this is an accepted limitation. The primary threat being mitigated here is accidental exposure in source code or Git history, not etcd compromise.

No audit logging: Kubernetes does not log secret reads by default. It is not possible to determine which pods accessed which secrets without enabling additional audit logging. This is acceptable for a single-developer project at this stage.

Decision

Kubernetes Secrets were selected as the secret management mechanism for UVote Stage 1.

The decision is shaped by the deployment context. The Kind cluster is local, ephemeral, and not exposed to external networks. The primary threat is accidental secret leakage into Git history or Docker image layers, not etcd compromise or insider attacks at the infrastructure level. Kubernetes Secrets eliminate this threat entirely by storing all sensitive values outside the codebase, while requiring zero additional infrastructure.

The ephemeral cluster lifecycle makes `deploy_platform.py` the natural place to generate and provision secrets. Running `openssl rand` during bootstrapping means every cluster creation produces fresh credentials, providing natural rotation without an explicit rotation mechanism. This turns a potential limitation of the Kind setup into a security property.

Each service receives only the secrets it requires, enforced at the Deployment level via individual `secretKeyRef` references. This implements least-privilege at the secret layer, complementing the per-service database users (ADR-014) and network policies (ADR-010).

The limitations of Kubernetes Secrets relative to Vault are accepted for Stage 1. The solution requires no additional pods, no additional RAM, and no learning investment beyond standard Kubernetes knowledge. A clear migration path to Vault exists for Stage 2 and requires no changes to application code.

A review is scheduled for the start of Stage 2 (February 2026) to evaluate migration to HashiCorp Vault for production deployment.

Other Related ADRs

- ADR-001: Python FastAPI Backend - environment variable consumption in services
- ADR-002: PostgreSQL Database - database credentials managed as Secrets
- ADR-010: Zero-Trust Network Policies - complementary isolation layer
- ADR-012: Kind Distribution - ephemeral cluster lifecycle driving the rotation approach

References

- Kubernetes Secrets Documentation
- Kubernetes RBAC for Secrets
- HashiCorp Vault on Kubernetes
- Bitnami Sealed Secrets

- EdgeX Foundry ADR Template