



KUBERNETES

A HANDS-ON GUIDE



BY DEVOPS SHACK

[Click here for DevSecOps & Cloud DevOps Course](#)

DevOps Shack

Kubernetes:

ConfigMaps, Secrets & Storage: A Hands-On Guide

Table of Contents

1. Introduction

- Overview of configuration management in Kubernetes
- Why ConfigMaps, Secrets, and Storage matter

2. Understanding ConfigMaps

- What is a ConfigMap?
- Use cases and best practices
- Creating ConfigMaps (via YAML, CLI, from files)
- Mounting ConfigMaps to Pods (env vars vs. volume)
- Live demo: Injecting configuration into an app

3. Secrets Management

- What is a Secret in Kubernetes?
- Difference between ConfigMap and Secret
- Types of Secrets (Opaque, TLS, Docker Registry, etc.)
- Creating and using Secrets (CLI, YAML, from files)
- Security considerations (RBAC, encryption, HashiCorp Vault intro)
- Live demo: Storing and using DB credentials

4. Storage in Kubernetes

- Introduction to Volumes
- Volume types overview (emptyDir, hostPath, NFS, PVC)
- Understanding Persistent Volumes (PV) and Claims (PVC)
- Dynamic vs. static provisioning
- StorageClasses and CSI drivers
- Live demo: Attaching persistent storage to a Pod

5. Real-World Use Cases

- App with external configuration and secrets
- Stateful applications with persistent volumes
- Environment-specific deployments using ConfigMaps/Secrets

6. Security & Best Practices

- Encrypting Secrets at rest
- RBAC for secrets and configmaps
- Avoiding config drift
- Centralized secret management (Vault, SOPS)

7. Troubleshooting and Debugging

- Common mistakes (volume mount paths, key not found, etc.)
- Using kubectl describe and logs
- Verifying mounted configs/secrets

8. CI/CD Integration

- Injecting configs/secrets via Helm or Kustomize

- Sample pipeline with GitHub Actions + Kubernetes

9. Final Hands-On Lab

- Deploy a production-ready app using:
 - ConfigMap for config
 - Secret for credentials
 - PVC for data persistence

10. Conclusion & Next Steps

- Summary of key takeaways
- Recommended tools and resources
- What's next: Sealed Secrets, External Secrets Operator, Vault



1. Introduction



Overview of Configuration Management in Kubernetes

In modern cloud-native applications, decoupling configuration from application code is a core principle. This allows teams to:

- Deploy the same container image across environments (dev, test, prod)
- Update settings without rebuilding the application
- Handle sensitive information securely
- Persist application state and data across pod restarts

Kubernetes provides robust built-in mechanisms to manage application configuration, secrets, and persistent storage through:

- **ConfigMaps** – for non-sensitive configuration data
- **Secrets** – for sensitive information like passwords and API keys
- **Persistent Volumes (PV)** and **Persistent Volume Claims (PVC)** – for managing storage in a portable and dynamic way

These tools help manage your applications more flexibly, allowing seamless updates, improved security, and better scalability.



Why ConfigMaps, Secrets, and Storage Matter

In any real-world deployment, your application will need:

- Environment-specific settings (e.g., API endpoints)
- Credentials to access external systems (e.g., DB, cache)
- Persistent storage to keep data even if pods restart

Without proper management of these components, applications become brittle, insecure, and difficult to scale or migrate. Kubernetes addresses this through a declarative model that lets you define configuration and storage as part of your infrastructure-as-code (IaC) workflow.

What You'll Learn in This Guide

By the end of this guide, you'll be able to:

- Create and manage ConfigMaps and Secrets
- Mount configurations and secrets into running pods
- Set up persistent storage for both stateless and stateful applications
- Securely handle sensitive data in production environments
- Build a complete CI/CD pipeline that includes config and storage management



2. Understanding ConfigMaps

○ What is a ConfigMap?

A **ConfigMap** is a Kubernetes API object used to **store non-sensitive key-value pairs** for application configuration. It allows you to decouple configuration artifacts from your image content to keep your application portable.

Use cases include:

- Environment variables
- Configuration files
- Command-line arguments

Creating ConfigMaps

Kubernetes offers multiple ways to create ConfigMaps:

☒ A. From Literal Key-Value Pairs (CLI)

```
kubectl create configmap app-config \
  --from-literal=APP_ENV=production \
  --from-literal=APP_PORT=8080
```

Check the result:

```
kubectl describe configmap app-config
```

☒ B. From a File

If you have a config file:

```
# app.properties APP_ENV=staging
APP_PORT=9090
```

Create the ConfigMap:

```
kubectl create configmap app-config --from-file=app.properties
```

☒ C. From a YAML Manifest

```
# configmap.yaml
```

```
apiVersion: v1
```

```
kind: ConfigMap
```

```
metadata:
```

```
  name: app-config
```

```
data:
```

```
  APP_ENV: "development"
```

```
  APP_PORT: "3000"
```

Apply it:

```
kubectl apply -f configmap.yaml
```

Mounting ConfigMaps to Pods

You can use a ConfigMap in two main ways:

o 1. As Environment Variables

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: demo-pod
```

```
spec:
```

```
  containers:
```

```
    - name: demo-container
```

```
      image: nginx
```

```
      envFrom:
```

```
        - configMapRef:
```


name: app-config

This will inject APP_ENV and APP_PORT into the container's environment.

o 2. As Volumes (Mounted Files)

apiVersion: v1

kind: Pod

metadata:

name: demo-pod

spec:

containers:

- name: demo-container

image: nginx

volumeMounts:

- name: config-volume

mountPath: /etc/config

volumes:

- name: config-volume

configMap:

name: app-config

This will mount each key in the ConfigMap as a file inside /etc/config/, and the file contents will be the values.

Best Practices for ConfigMaps

- Use **environment-specific ConfigMaps** (e.g., app-config-dev, app-config-prod)
- Avoid putting **sensitive data** in ConfigMaps (use Secrets instead)
- Use `kubectl edit configmap <name>` carefully—manual edits can be risky in GitOps environments

G Live Demo Idea (*Optional hands-on task*)

You can try the following to test ConfigMaps:

1. Create a ConfigMap with a custom welcome message.
2. Deploy an NGINX container that mounts the ConfigMap into /usr/share/nginx/html/index.html.
3. Visit the NGINX service in your browser to see the message rendered dynamically.

3. Secrets Management

- What is a Secret in Kubernetes?


A **Secret** is a Kubernetes object used to store **sensitive data**, such as:

- Passwords
- API keys
- SSH keys
- TLS certificates

Unlike ConfigMaps, Secrets are **base64-encoded by default** (note: not encrypted) and are designed to be more secure.

ConfigMap vs Secret: Key Differences

Feature	ConfigMap	Secret
Purpose	Non-sensitive data	Sensitive data
Encoding	Plaintext	Base64
Storage	etcd (unencrypted)	etcd (can be encrypted)
RBAC Required	Optional	Strongly recommended
Usage Methods	Env var / Volume	Env var / Volume

 **Important:** Base64 is not encryption. Always encrypt secrets at rest and restrict access using RBAC.

Creating Kubernetes Secrets

☒ A. From Literal Key-Value Pairs

```
kubectl create secret generic db-secret \
  --from-literal=username=admin \
  --from-literal=password='S3cr3tP@ss'
```

To view (base64 encoded):

```
kubectl get secret db-secret -o yaml
```

To decode:

```
echo 'U2VjcmV0UGFzcw==' | base64 --decode
```

☒ B. From a YAML Manifest

```
apiVersion: v1
```

```
kind: Secret
```

```
metadata:
```

```
  name: db-secret
```

```
type: Opaque
```

```
data:
```

```
  username: YWRtaW4=      # base64 of "admin"
```

```
  password: UzNjcjN0UEBzcw== # base64 of "S3cr3tP@ss"
```

Use `echo -n "value" | base64` to generate encoded

strings. Apply it:

```
kubectl apply -f db-secret.yaml
```

Mounting Secrets into Pods

○ 1. As Environment Variables

```
env:
```

```
  - name: DB_USER
```

```
    valueFrom:
```

```
      secretKeyRef:
```

```
        name: db-secret
```

```
        key: username
```

```
  - name: DB_PASS
```

```
    valueFrom:
```

```
      secretKeyRef:
```

name: db-secret

key: password

Ideal for apps expecting credentials from environment variables.

o 2. As Mounted Files

volumes:

- name: secret-volume

secret:

secretName: db-secret

containers:

- name: app

image: myapp:latest

volumeMounts:

- name: secret-volume

mountPath: "/etc/secrets"

readOnly: true

This will mount files like:

/etc/secrets/username

/etc/secrets/password

Each file will contain the decoded secret.

Security Considerations

1. RBAC Permissions

- o Use Role or ClusterRole to limit access to secrets.
- o Example:

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: default

name: secret-reader

rules:

- apiGroups: [""]

resources: ["secrets"]

verbs: ["get"]

2. Encryption at Rest

- Configure your Kubernetes distribution to **encrypt secrets in etcd** (default is plaintext).
- Usually involves setting up a EncryptionConfiguration file and restarting kube-apiserver.

3. Avoid Logging Secrets

- Never log secrets via kubectl logs or include them in output accidentally.

4. Use External Secret Managers

- Integrate with tools like **HashiCorp Vault**, **AWS Secrets Manager**, or **Sealed Secrets** for production-grade secret management.

G Live Demo Idea

Deploy a Node.js or Python app that:

- Reads DB credentials from environment variables (via Secret)
- Connects to a fake DB or logs the credentials to stdout
- (Optional) Update the secret and roll the pod to show dynamic reloading



4. Storage in Kubernetes

Kubernetes was originally designed for stateless workloads, but real-world applications often need to **store persistent data**—for logs, databases, sessions,

and user uploads. Kubernetes solves this with **Volumes, Persistent Volumes**

(PV), and Persistent Volume Claims (PVC).

○ What Is a Volume?

A **Volume** in Kubernetes is a directory accessible to containers in a Pod. Unlike container filesystems, a Volume's data can persist across container restarts **within the same Pod**.

There are different types of volumes for different use cases:

Volume Type	Use Case
emptyDir	Temporary scratch space, cleared when pod dies
hostPath	Access host machine's file system
configMap	Mount ConfigMap data as files
secret	Mount Secret data securely
persistentVolumeClaim	Use persistent storage from a volume provider

○ Example: emptyDir Volume

volumes:

```
- name: temp-storage  
  emptyDir: {}
```

volumeMounts:

```
- mountPath: /tmp/cache  
  name: temp-storage
```

Used for caching or temporary storage between container processes.

Persistent Storage: PV and PVC

Kubernetes uses a **two-step process** to manage persistent storage:

1. **PersistentVolume (PV):** Represents a piece of storage in the cluster (e.g., EBS, NFS, local disk).
2. **PersistentVolumeClaim (PVC):** A user's request for storage by size and access mode.

Static Provisioning Example

Create a PV

apiVersion: v1

kind: PersistentVolume

metadata:

name: my-pv

spec:

capacity:

storage: 1Gi

accessModes:

- ReadWriteOnce

hostPath:

path: /mnt/data

hostPath is only suitable for single-node testing.

Create a PVC

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: my-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 1Gi

Use PVC in a Pod

volumes:

- name: persistent-storage

persistentVolumeClaim:

claimName: my-pvc

volumeMounts:

- mountPath: "/data"

name: persistent-storage

Now, the application will have /data backed by persistent storage.

Dynamic Provisioning with StorageClasses

With cloud-native clusters (e.g., AWS EKS, GKE, AKS), you can let Kubernetes **automatically provision storage** using a StorageClass.

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: dynamic-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 5Gi

storageClassName: standard

Kubernetes talks to the cloud provider and provisions a disk (like AWS EBS) automatically.

Volume Access Modes

- **ReadWriteOnce (RWO):** One node can read/write
- **ReadOnlyMany (ROX):** Many nodes can read
- **ReadWriteMany (RWX):** Many nodes can read/write (NFS, GlusterFS)

CSI Drivers (Container Storage Interface)

Kubernetes supports pluggable storage backends via CSI. Examples:

- ebs.csi.aws.com (AWS)
- pd.csi.storage.gke.io (GCP)
- nfs.csi.k8s.io

(NFS) To use one, you need:

- CSI driver installed
- StorageClass defined
- PVC referencing the StorageClass

G Live Demo Idea

Deploy a database (e.g., PostgreSQL or MongoDB) using:

- PVC for data storage
- ConfigMap for DB settings
- Secret for DB password

Restart the pod and show that **data is retained**, proving persistent storage is working.

Best Practices for Storage in Kubernetes

- Always use **PVCs** instead of hardcoding volume paths
- Choose appropriate access mode (RWO/RWX) based on app requirements
- Set **resource quotas** to avoid over-allocation
- For databases, use **stateful sets** instead of deployments

5. Real-World Use Cases of ConfigMaps, Secrets, and Storage

Understanding individual components is great—but seeing how they come together in real-world applications brings the concepts to life. This section explores **how ConfigMaps, Secrets, and Storage work together in actual deployments**.

Use Case 1: Deploying a Web Application with Environment Configs and Secrets

Imagine a Node.js or Python web application that connects to a PostgreSQL database. You need to:

- Store DB connection details securely (Secrets)
- Configure app mode, logging level, or custom variables (ConfigMap)
- Ensure database data persists across pod restarts (PersistentVolumeClaim)

Resources Involved:

- ConfigMap: For APP_MODE, LOG_LEVEL
- Secret: For DB_USER, DB_PASS
- PVC: For PostgreSQL data

Deployment Sample:

apiVersion: apps/v1

kind: Deployment

metadata:

name: webapp

spec:

replicas: 1

selector:

matchLabels:

app:

webapp

template:

metadata:

labels:

app: webapp

```
spec:
  containers:
  - name: webapp
    image: mycompany/webapp:latest
    env:
    - name: APP_MODE
      valueFrom:
        configMapKeyRef:
          name: app-config
          key: APP_MODE
    - name: DB_USER
      valueFrom:
        secretKeyRef:
          name: db-secret
          key: username
    - name: DB_PASS
      valueFrom:
        secretKeyRef:
          name: db-secret
          key: password
```

Use Case 2: Stateful Database with Persistent Storage

Let's say you're deploying **MongoDB**:

- Each replica needs its own persistent volume
- Passwords must be hidden
- App config should be flexible

☒ StatefulSet + PVC Example:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo
spec:
  serviceName: "mongo" replicas: 1
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
        - name: mongo
          image:
            mongo:4.4 env:
              - name: MONGO_INITDB_ROOT_USERNAME
                valueFrom:
                  secretKeyRef:
                    name: mongo-secret
                    key: username
              - name: MONGO_INITDB_ROOT_PASSWORD
```



```
    valueFrom:
    secretKeyRef:
      name: mongo-secret
      key: password
  volumeMounts:
    - name: mongo-data
      mountPath: /data/db
  volumeClaimTemplates:
    - metadata:
        name: mongo-data
      spec:
        accessModes: [ "ReadWriteOnce" ]
        resources:
          requests:
            storage: 5Gi
```

Use Case 3: External Secrets + ConfigMap Driven Frontend

A frontend React/Angular app might:

- Use a ConfigMap to inject base URLs or feature flags (API_BASE_URL, ENABLE_CHAT)
- Pull API keys from a **Secret Manager** (like AWS Secrets Manager or HashiCorp Vault) and inject via sidecars or admission controllers

This architecture decouples config and secrets even for static apps.

G Use Case 4: Rolling Updates with ConfigMap Changes

You can trigger **rolling updates** in a Deployment when a ConfigMap or Secret is updated by mounting them as volumes and setting subPath to load specific files. Alternatively, use checksum annotations to detect changes.

Real-World Tips

- Use **CI/CD tools (e.g., ArgoCD, Flux, Helm)** to manage and template your configs/secrets.
- **Don't hardcode credentials** in your manifests—always inject them via Secrets.
- For shared configs, use **namespace-specific ConfigMaps** or **ConfigMap generators**.
- Use **StorageClasses** for dynamic provisioning, especially in cloud-native apps.


6. Best Practices for Managing Configuration Data in Kubernetes

While ConfigMaps and Secrets are powerful, improper handling can lead to

security issues, deployment failures, and poor observability. This section outlines practical **best practices** to ensure your configuration management in Kubernetes is reliable, secure, and maintainable.

1. Keep Secrets Secret

- Never store secrets in plain-text ConfigMaps.
- Use Secrets for **credentials, tokens, API keys, and sensitive configs.**
- Use **encryption at rest** (EncryptionConfiguration in kube-apiserver).
- Enable **RBAC** to control who can access or edit Secrets.

 **Pro Tip:** Use external secret managers like **HashiCorp Vault, AWS Secrets Manager, or Google Secret Manager** with Kubernetes integrations.

2. Use Volume Mounts for Large Configs

If your config files are large (e.g., .yaml, .json, or .conf files), use ConfigMaps and mount them as volumes:

volumeMounts:

```
- name: config-volume  
  mountPath: /etc/app/config
```

volumes:

```
- name: config-volume  
  configMap:  
    name: app-config
```

This keeps your app code clean and avoids cluttering environment variables.

3. Trigger Pod Restarts on Config Changes

By default, Kubernetes **does not restart pods** when ConfigMaps or Secrets change.

Solutions:

- Mount as volume (K8s will update the file, but app must re-read it)
- Use checksum annotations in the

Deployment: [annotations](#):

[config-checksum: "{{ .Values.config | sha256sum }}"](#)

- Use **sidecars** or **reloader controllers** (e.g., [kube-reload](#)) to detect and trigger redeploy.

4. Avoid Bloating ConfigMaps/Secrets

- Keep ConfigMaps/Secrets concise—**each value ≤1MB**.
- Don't store large binaries or blobs.
- Split into multiple resources when config size grows.

5. Use Namespaces and Labels

Use **namespaces** and **labels** to logically isolate environments and categorize resources.

[metadata](#):

[namespace: dev](#)

[labels](#):

[app: myapp](#)

[env: dev](#)

This improves management across environments like dev, staging, and production.

G 6. Validate Before Applying

Use tools to **validate manifests** before applying:

- `kubectl apply --dry-run=client -f file.yaml`

Schema validation tools like

- Policy engines like [OPA Gatekeeper](#)

7. Automate with GitOps

Use tools like **ArgoCD** or **FluxCD** to version and deploy ConfigMaps/Secrets:

- Store config values in Git
- Template with **Helm**, **Kustomize**, or **Jsonnet**
- Monitor for drifts and auto-sync

8. Audit and Monitor Usage

Track who accesses or modifies ConfigMaps/Secrets:

- Use Audit Logs in cloud providers or custom admission controllers.
- Scan clusters for exposed secrets or bad practices using tools like:
 - [kubesecc](#)
 - [kube-hunter](#)
 - [Trivy](#)

9. Use Template Engines Wisely

Use **Helm** or **Kustomize** to generate ConfigMaps and Secrets from .env or YAML templates:

`apiVersion: v1`

`kind: ConfigMap`

`metadata:`

`name: {{ .Release.Name }}-config`

`data:`

`ENV_MODE: {{ .Values.env.mode }} SERVICE_URL: {{
 .Values.env.serviceUrl }}`

This ensures consistency across environments and reduces duplication.

10. Set Resource Quotas and Limits

Avoid accidental bloat by setting:

`apiVersion: v1`

`kind: ResourceQuota`

`spec:`

`hard:`

`count/configmaps: "50"`

`count/secrets: "30"`

This keeps your cluster manageable and avoids surprise resource overuse.

7. Security Considerations

Security is paramount when managing ConfigMaps, Secrets, and Storage in Kubernetes. This section covers key security practices to protect sensitive data and ensure your cluster remains secure.

1. Protect Secrets at Rest and in Transit

- Enable **Encryption at Rest** for Secrets by configuring the kube-apiserver with encryption providers.
- Use **TLS** for all Kubernetes API communications.
- Use **Network Policies** to restrict pod-to-pod and pod-to-service communication.

2. Limit Access with RBAC

- Define **fine-grained Role-Based Access Control (RBAC)** rules for who can view, edit, or delete Secrets and ConfigMaps.
- Use **least privilege principle** — only grant access when absolutely necessary.

Example Role:

kind: Role

apiVersion: rbac.authorization.k8s.io/v1

metadata:

namespace: production

name: secret-reader

rules:

- apiGroups: [""]

resources: ["secrets"]

verbs: ["get", "list"]

3. Avoid Secrets in Logs and Environment Variables

- Avoid printing secrets in application logs.
- If possible, prefer mounting secrets as **files** instead of environment variables, which can be exposed in process lists.

4. Use External Secret Management Solutions

- Integrate Kubernetes with **Vault**, **AWS Secrets Manager**, or **Azure Key Vault**.
- Use tools like **External Secrets Operator** to sync external secrets into Kubernetes as native Secrets.

5. Prevent Secret Sprawl

- Avoid creating too many duplicate Secrets.
- Regularly audit and remove unused Secrets.
- Rotate secrets periodically.

6. Implement Secret Rotation

- Use automated tools or scripts to rotate secrets without downtime.
- Ensure applications handle secret reloads gracefully.

7. Secure Persistent Storage

- Use storage encryption (e.g., encrypted EBS volumes).
- Restrict access to PVs and PVCs via **Pod Security Policies** or **Security Contexts**.
- Limit volume mounts to trusted pods only.

8. Use Pod Security Policies and Security Contexts

- Use **Pod Security Policies** to control permissions, volume types, and capabilities.
- Set **Security Contexts** to drop unnecessary Linux capabilities in pods.

9. Audit and Monitor

- Enable Kubernetes audit logs for tracking access to ConfigMaps and Secrets.
- Use monitoring tools like **Falco**, **Kube-bench**, or **Open Policy Agent** to detect suspicious activities.

10. Backup and Disaster Recovery

- Regularly backup Secrets and ConfigMaps along with cluster state.
- Test restore processes to ensure rapid recovery in case of incidents.

8. Tools and Utilities to Manage ConfigMaps, Secrets & Storage

Managing ConfigMaps, Secrets, and Storage can get complex in larger clusters. Fortunately, there are several tools and utilities designed to simplify tasks like creation, synchronization, encryption, and monitoring.

1. kubectl

The default Kubernetes CLI tool:

- Create/update ConfigMaps and Secrets:

```
kubectl create configmap my-config --from-file=app.conf
```

```
kubectl create secret generic db-secret --from-literal=username=admin --from-literal=password=pass123
```

- View and edit

resources: [kubectl get secrets](#)

```
kubectl edit configmap my-config
```

2. Sealed Secrets (Bitnami)

- Encrypt Secrets into “sealed secrets” safe to store in Git.
- Only the controller in the cluster can decrypt and create native Secrets.
- Enables **GitOps-friendly** secret

management. [GitHub: Sealed Secrets](#)

3. External Secrets Operator

- Synchronizes secrets from external secret managers (AWS Secrets Manager, HashiCorp Vault, Azure Key Vault) into Kubernetes Secrets automatically.
- Keeps secrets up to date with external

providers. [GitHub: External Secrets Operator](#)

4. Helm

- Package manager for Kubernetes.

- Use helm template or helm install to generate and deploy ConfigMaps and Secrets from templates.
- Supports value overrides for different environments.

5. Kustomize

- Native Kubernetes configuration customization tool.
- Overlay and patch ConfigMaps and Secrets.
- Built-in to kubectl (kubectl kustomize).

6. KubeVault

- Kubernetes operator to manage secret lifecycle, encryption, and auto-rotation using HashiCorp Vault.
- Simplifies Vault integration for apps in Kubernetes.

7. Reloader

- Watches ConfigMaps and Secrets for changes and triggers rolling updates of pods automatically.
- Useful for apps that don't support hot-reloading config files.

D 8. Kube-ops-view / Lens

- Visual cluster dashboards for monitoring ConfigMaps, Secrets, and Storage usage.
- Helps with cluster resource management.

9. Trivy / Kube-bench / Kube-hunter

- Security scanning tools to detect misconfigurations and vulnerable secrets.
- Run scans on clusters or container images.

G 10. kubectl plugins

- Extend kubectl with plugins like kubectl-plugins/config-viewer to easily inspect ConfigMaps and Secrets.
- Check kubectl plugin index for available tools.

9. Troubleshooting Common Issues

Working with ConfigMaps, Secrets, and Storage in Kubernetes can sometimes lead to unexpected issues. This section covers frequent problems and how to resolve them.

1. Pods Not Updating When ConfigMap or Secret Changes

- **Cause:** Kubernetes doesn't restart pods automatically when ConfigMaps or Secrets change.
- **Solution:**
 - Use volume mounts and make your app watch for file changes.
 - Use annotations with checksums in your Deployment spec to trigger pod rollout on change.
 - Use a reloader tool like [stakater/Reloader](#).

2. Secret Not Found or Permission Denied

- **Cause:**
 - Secret does not exist in the pod's namespace.
 - RBAC rules restrict access.
 - Typo in the Secret name/key.
- **Solution:**
 - Verify secret exists: `kubectl get secret <secret-name> -n <namespace>`
 - Check pod's namespace matches Secret's namespace.
 - Check RBAC roles and bindings.
 - Check spelling in manifest.

3. ConfigMap or Secret Size Limits Exceeded

- **Cause:** Kubernetes limits size to ~1MB for ConfigMaps and Secrets.
- **Solution:**
 - Split large configs into multiple ConfigMaps or Secrets.
 - Store large data externally (e.g., in a volume or external service).

4. PVC Bound Status is Pending

- **Cause:** StorageClass may not be available or misconfigured.
- **Solution:**
 - Check available StorageClasses: `kubectl get storageclass`
 - Ensure your PVC references a valid StorageClass.
 - Check the underlying cloud provider or storage system for issues.

5. Data Not Persisting After Pod Restart

- **Cause:** PVC not mounted properly or using ephemeral storage.
- **Solution:**
 - Check your Pod spec mounts the PVC to the correct path.
 - Ensure PVC is bound to a PersistentVolume.
 - Verify StorageClass supports `ReadWriteOnce` or appropriate access modes.

6. Environment Variables from ConfigMap/Secret Not Available in Pod

- **Cause:** Misspelled keys or missing references in deployment manifest.
- **Solution:**
 - Double-check keys in ConfigMap or Secret.
 - Ensure correct references using `valueFrom.configMapKeyRef` or `valueFrom.secretKeyRef`.

7. Pod CrashLoopBackOff Due to Secret Misconfiguration

- **Cause:** Application expects secret file/variable but it's missing or malformed.
- **Solution:**
 - Inspect pod logs with `kubectl logs <pod-name>`.

Confirm secret keys and mount paths.

- Check app startup scripts handle missing secrets gracefully.

8. ConfigMap Data Not Loaded in Application

- **Cause:** Application does not reload config after pod starts.
- **Solution:**
 - Restart pod to pick up ConfigMap changes.
 - Modify app to watch mounted config files or restart on changes.

9. PersistentVolumeClaim Cannot Be Deleted

- **Cause:** PVC still bound to a PV, or PV is stuck in Terminating.
- **Solution:**
 - Delete associated PV first if appropriate.
 - Use `kubectl patch pvc <pvc-name> -p '{"metadata":{"finalizers":null}}'` to force delete.

10. Secrets Exposed in Logs or CLI

- **Cause:** Environment variables or command output reveal secrets.
- **Solution:**
 - Use mounted files instead of env vars for sensitive data.
 - Avoid logging secret values.
 - Restrict access to logs and audit CLI usage.

10. Summary and Further Learning Resources

Summary

- **ConfigMaps** and **Secrets** are essential Kubernetes primitives to manage configuration and sensitive data separately from application

code.

Use **ConfigMaps** for non-sensitive config and **Secrets** for sensitive data.

- Properly mount them as environment variables or files inside pods.
- Choose the right **storage** options and understand PersistentVolumes and PersistentVolumeClaims.
- Follow **best practices** for security, access control, and lifecycle management.
- Utilize tools like **kubectl**, **Helm**, **Sealed Secrets**, and **External Secrets Operator** to simplify management.
- Troubleshoot common issues related to configuration updates, access, storage, and pod failures efficiently.
- Always secure your data with encryption, RBAC, and audit policies.

Further Learning Resources

- **Kubernetes Official Documentation**
<https://kubernetes.io/docs/concepts/configuration/configmap/>
<https://kubernetes.io/docs/concepts/configuration/secret/>
<https://kubernetes.io/docs/concepts/storage/persistent-volumes/>
- **Books**
 - *Kubernetes Up & Running* by Kelsey Hightower et al.
 - *The Kubernetes Book* by Nigel Poulton.
- **Tools and Projects**
 - [Sealed Secrets \(Bitnami\)](#)
 - [External Secrets Operator](#)
 - [Reloader](#)
 - [Helm](#)
- **Security Best Practices**
<https://kubernetes.io/docs/concepts/security/overview/>
<https://www.cncf.io/blog/2020/07/02/securing-kubernetes-best-practices/>

Community and Tutorials

- Kubernetes Slack: <https://slack.k8s.io/>
- Katacoda Kubernetes Scenarios:
<https://www.katacoda.com/courses/kubernetes>
- CNCF Tutorials and Webinars: <https://www.cncf.io/learning/>