Devops Shack

Mastering Microservices Deployment on Kubernetes

BY DEVOPS SHACK

DevOps Shack

Mastering Microservices Deployment on Kubernetes

◇ What is a Service Mesh and Why Use It?

- ◇ Installing Istio or Linkerd into Your Cluster
- ◇ mTLS, Traffic Routing, and Observability with Sidecars

6. Enabling Auto-Scaling and Resilience

- ◇ Horizontal Pod Autoscaler (HPA) with CPU/Memory Metrics
- ◇ Vertical Pod Autoscaler and KEDA for Event Scaling
- ◇ Liveness, Readiness, and Startup Probes for Recovery

7. Observability and Monitoring

- ◇ Centralized Logging with Fluent Bit, Loki, or ELK
- ◇ Metrics with Prometheus and Grafana Dashboards
- ◇ Tracing and Service Graphs with Jaeger or Istio

8. Securing and Hardening Microservices

- ◇ Role-Based Access Control (RBAC) and Network Policies
- ◇ Securing Secrets with Vault or External Secrets Operator
- ◇ Image Scanning and Supply Chain Protection (Trivy, Cosign, SBOMs)

☑ 1. Introduction to Microservices on Kubernetes

Deploying microservices on Kubernetes has become a de facto standard in modern cloud-native architecture. But before we start configuring services, scaling replicas, or wiring up Ingress, it's important to understand **why Kubernetes fits microservices so well**, what core concepts are involved, and what unique challenges this architecture introduces.

o 1.1 What Makes Kubernetes Ideal for Microservices?

Microservices break applications into smaller, independent services that:

- Can be developed and deployed separately
- Scale independently
- Use different languages or tech stacks
- Communicate over APIs (usually REST/gRPC)

Kubernetes aligns perfectly with this model because:

Microservices Need	Kubernetes Capability
Independent deployment	Each service is its own Deployment/Pod
Auto-scaling	HPA, VPA, and Cluster Autoscaler
Service discovery	DNS-based discovery via Kubernetes Services
Resilience & restarts	Self-healing via probes and restart policies
Resource isolation	Namespaces, resource quotas, and limits
Environment abstraction	ConfigMaps, Secrets, and VolumeMounts

💡 Kubernetes provides **platform-level primitives** to solve common microservice concerns like availability, isolation, scaling, and discovery — without reinventing the wheel.

o 1.2 Core Concepts: Pods, Services, and Namespaces

Before diving deeper, it's critical to understand **three foundational Kubernetes building blocks** that apply to all microservice setups.

📦 **Pod**

The smallest deployable unit in Kubernetes.

- Usually runs a single container (or tightly coupled containers).
- Each microservice instance = **one or more**

Pods. apiVersion: v1

kind: Pod

metadata:

  name: payment-pod

spec:

  containers:

    - name: payment

      image: myapp/payment:1.0

🔗 Service

- Abstracts access to a group of Pods (usually those of one microservice).
- Provides **stable DNS** and **load balancing** across

Pods. apiVersion: v1

kind: Service

metadata:

  name: payment-svc

spec:

  selector:

    app: payment

  ports:

    - port: 80

      targetPort: 8080

🧱 Namespace

- Logical grouping for isolating resources (e.g., dev, staging, prod)
- Helps organize teams, environments, or

applications kubectl create namespace orders

You can scope resources like:

metadata:

  namespace: orders

🧠 **Best Practice**: Use one namespace per environment or per microservice group, especially when combined with RBAC.

o **1.3 Challenges in Managing Distributed Services**

While Kubernetes **solves infrastructure-level problems**, microservices introduce **complex system-level challenges** that engineers must still address:

◇ **Service Communication**

- Internal APIs need DNS-based routing (service- name.namespace.svc.cluster.local)
- Load balancing and retries become critical

o **Observability**

- Debugging across 10+ services means:
  - Distributed tracing
  - Centralized logging
  - Service-level dashboards

o **Deployment Coordination**

- Each service might be on a different release cycle
- Inter-service dependencies require contract testing or versioning

◇ **Security**

- Each microservice must be secured independently

- Secrets, tokens, and service-to-service encryption (e.g., mTLS) must be managed

🚨 As you scale from 2 → 20 → 200 microservices, orchestration, visibility, and governance become exponentially harder — and Kubernetes gives you the base, but **you need the patterns, tools, and discipline** to handle the rest.

☑ 2. Designing Microservices for Kubernetes

Deploying microservices to Kubernetes without proper design can lead to poor scalability, tightly coupled services, and unnecessary complexity. This section covers how to structure your codebases, containerize services effectively, and manage configuration cleanly across environments.

2.1 Structuring Your Codebase: APIs, DBs, and Shared Services

Before deploying to Kubernetes, microservices should be architected with **clear domain boundaries** and **minimal dependencies** between services.

💡 **Design Patterns to Follow:**

- **One repo per service** (recommended for larger orgs) or **monorepo with CI control** (for startups).
- Each service should:
  - Own its **database** (no shared DBs!)
  - Expose **REST/gRPC APIs**
  - Avoid tight coupling with other services

🧱 **Example Directory Layout (Monorepo):**

```
/services
  /orders-service
    /src
    Dockerfile
    helm-chart/
  /payments-service
    /src
    Dockerfile
    helm-chart/
/common-libs
```

🔐 **Separate Databases Per Service:**

Microservice	DB Type	Ownership
Orders	PostgreSQL	orders-db
Payments	MongoDB	payments-db

📌 Avoid using a **shared schema** between services — schema changes become a nightmare across teams.

2.2 Containerizing Each Microservice with Best Practices

Each microservice must be independently **containerized**, with a lightweight, secure, and optimized Dockerfile.

✅ **Example: Dockerfile for Node.js Microservice**

```
FROM node:20-alpine

WORKDIR /app

COPY package*.json ./
RUN npm ci --omit=dev

EXPOSE 3000

CMD ["node", "server.js"]
```

Best Practices:

- Use **multi-stage builds** for languages like Java or Go
- Set a proper HEALTHCHECK for Kubernetes
- **Run as non-root** inside the container
- Keep image sizes small (Alpine, Distroless)

🧠 Every container image should include **only what's needed for runtime** — no source files, compilers, or test tools.

2.3 Managing Configuration per Environment (ConfigMap, Secret)

Hardcoding environment-specific config into your image is a **production anti-pattern**.

Instead, use:

- **ConfigMaps** for general app configuration (URLs, log levels)
- **Secrets** for sensitive data (passwords, tokens, keys)

☑ **Example: ConfigMap for Non-Sensitive Env Vars**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: orders-config
data:
  LOG_LEVEL: debug
  API_URL: https://api.orders.dev
```

☑ **Example: Secret for Sensitive Data**

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
type: Opaque
data:
  username: b3JkZXJzX3VzZXI=      # base64
  encoded password: c3VwZXJzZWNyZXQxMjM=
```

☑ **Injecting into Pods:**

```
env:
  - name: DB_USER
    valueFrom:
    secretKeyRef:
```

```
name: db-credentials
      key: username
     - name: LOG_LEVEL
       valueFrom:
    configMapKeyRef:
      name: orders-config key:
          LOG_LEVEL
```

🔒 **Never bake secrets into Docker images** or Helm charts — always inject them via Kubernetes primitives.

📦 **Design Takeaways:**

Focus Area	Best Practice
Code Structure	Domain-focused, independently deployable
Containerization	Lightweight, secure, multi-stage Dockerfiles
Configuration	Use ConfigMaps & Secrets — avoid hardcoding

☑ 3. Deploying Microservices to Kubernetes

Deployment is where all your effort in designing and containerizing microservices comes to life. Kubernetes provides powerful abstractions, but without the right approach, your services can become hard to manage and scale. This section guides you through how to deploy microservices cleanly and reproducibly across environments.

3.1 Creating Kubernetes Manifests: Deployment, Service, Ingress

Every microservice deployed on Kubernetes needs (at minimum):

- A **Deployment** to manage Pods
- A **Service** for internal communication
- An **Ingress** (optional) to expose it externally

☑ **Minimal Working Set of YAMLs:**

1. Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders-deployment
spec:
  replicas: 3
  selector:
  matchLabels:
      app: orders
  template:
  metadata:
      labels:
        app: orders
    spec:
      containers:
        - name: orders
          image: myapp/orders:1.0
          ports:
```

```yaml
        containerPort: 3000
          envFrom:
            - configMapRef:
                name: orders-config
            - secretRef:
                name: db-credentials
```

2. Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: orders-service
spec:
  selector:
    app: orders
  ports:
    - port: 80
      targetPort: 3000
  type: ClusterIP
```

3. Ingress (if needed)

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: orders-ingress
  annotations:
```

```
nginx.ingress.kubernetes.io/rewrite-target: /
 spec:
  rules:
   - host:
    orders.myapp.com http:
     paths:
      - path: /
        pathType: Prefix
        backend:
         service:
          name: orders-service
          port:
           number: 80
```

🌐 Ingress enables HTTP routing across multiple services through a single domain or LoadBalancer IP.

3.2 Managing Multiple Environments (Dev, Staging, Prod)

Different environments need:

- Different resource limits
- Different DB/API URLs
- Secrets and credentials isolation

☑ **Recommended Structure:**

```
/k8s
 /dev
  orders-deployment.yaml
  orders-service.yaml
 /staging
```

/prod

☑ **Helm (or Kustomize) for Overrides:**

Use Helm values files:

helm upgrade --install orders ./orders-chart -f values-dev.yaml This allows you to **reuse the same templates** with:

- Different replica counts
- Different container images
- Different ConfigMaps/Secrets

3.3 GitOps and Helm: Automating Deployments

Manually applying YAMLs (kubectl apply) doesn't scale. Use **GitOps** to:

- Treat manifests as code
- Trigger deployments via Git pushes
- Promote environments with PRs

☑ **Tools:**

Tool	Use Case
Helm	Templating and managing charts
ArgoCD	GitOps-based deployment controller
Flux	Lightweight GitOps engine for clusters
Kustomize	Overlay-based YAML configuration

☑ **Sample ArgoCD Workflow:**

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
```

```
  name: orders
spec:
  source:
    repoURL: https://github.com/myorg/microservices-k8s
    path: k8s/prod/orders
    targetRevision: main
  destination:
    server: https://kubernetes.default.svc
    namespace: prod
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

📌 ArgoCD watches your Git repo and keeps your cluster in sync with declared state. No more manual kubectl.

📦 **Deployment Best Practices Summary:**

Component	Best Practice
Deployment	Versioned image tags, health probes, env injection
Service	Always define internal Services, even if using Ingress
Multi-Env	Use Helm or Kustomize for environment-specific values
GitOps	Automate via ArgoCD or Flux, not kubectl apply

☑ 4. Exposing Microservices with Ingress Controllers

In microservice architecture, **exposing APIs and UIs** through a clean, secure, and scalable interface is essential. Kubernetes provides several ways to expose services — from simple NodePort to advanced Ingress Controllers with TLS, rewrites, and authentication.

4.1 Understanding ClusterIP, NodePort, LoadBalancer, and Ingress

Each type of Kubernetes Service exposes your microservices in different ways. Picking the right one matters.

Type	Use Case	Accessible From
ClusterIP	Default, for internal-only services	Inside the cluster only

Type	Use Case	Accessible From
NodePort	Quick testing; exposes on each node's IP	Outside, via node IP:port
LoadBalancer	External access via cloud-managed LB (AWS, GCP)	Public internet
Ingress	Routes HTTP(S) traffic with rules and TLS support	Domain-based access

🌐 **Example:**

ClusterIP – Internal DB access:

```
type: ClusterIP
```

NodePort – Basic test setup:

```
type: NodePort
nodePort: 30080
```

LoadBalancer – AWS/GCP Production Setup:

```
type: LoadBalancer
```

Ingress – Domain-based routing:

```
kind: Ingress
```

⚠ Only Ingress lets you expose **multiple services on the same IP** with **clean domain paths** and **TLS support**.

4.2 Setting Up NGINX or Traefik Ingress Controller

An Ingress **resource** does nothing by itself — it needs an **Ingress Controller** to read the rules and route traffic accordingly.

☑ **Install NGINX Ingress via Helm:**

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
```

```
helm repo update

helm install nginx-ingress ingress-nginx/ingress-nginx \
  --namespace ingress-nginx --create-
namespace
```

This sets up:

- A **DaemonSet or LoadBalancer Service** for traffic entry
- A controller that listens for Ingress rules

✅ **Traefik Alternative (More CRD-Driven):**

```
helm repo add traefik https://helm.traefik.io/traefik

helm install traefik traefik/traefik --namespace traefik --create-namespace
```

✨ Traefik supports CRDs, Let's Encrypt, mTLS, and fine-grained routing out of the box.

o 4.3 Using Ingress Annotations, TLS, and Rewrite Rules

Once the controller is ready, you can define advanced rules for routing, SSL, and URL manipulation.

✅ **Basic Ingress Example:**

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: orders-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
```

```yaml
  host: orders.mycompany.com
    http:
      paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: orders-service
              port:
                number: 80
```

☑ **TLS Example with Secret:**

```yaml
spec:
  tls:
    - hosts:
        - orders.mycompany.com
      secretName: tls-orders-secret
```

Generate the secret:

```
kubectl create secret tls tls-orders-secret \
  --cert=cert.pem --key=key.pem
```

☑ **Auth Example (NGINX Basic Auth):**

```yaml
annotations:
nginx.ingress.kubernetes.io/auth-type: basic
  nginx.ingress.kubernetes.io/auth-secret: basic-auth
  nginx.ingress.kubernetes.io/auth-realm: "Protected Area"
```

🔐 This allows per-route auth — ideal for internal tools, dashboards, or pre-production UIs.

📦 **Ingress Exposure Summary:**

Feature	Tool/Component	Purpose
Public Access	LoadBalancer + Ingress	Domain-level exposure
TLS Encryption	Ingress + Secret	HTTPS with custom or Let's Encrypt certs
Route Control	Ingress annotations	Rewrite, prefix stripping, redirects
Auth/Rate Limits	NGINX or Traefik configs	Built-in middlewares

☑ 5. Adding a Service Mesh for Observability and Control

As microservices grow, so does the complexity of **service-to-service communication**. A service mesh provides a transparent way to add **retries, timeouts, traffic shifting, encryption, and observability** across all your services with minimal application changes.

5.1 What is a Service Mesh and Why Use It?

A **service mesh** is a dedicated infrastructure layer that handles **network communication** between services. Instead of writing retry logic, metrics, or encryption inside the app, you offload that responsibility to a **sidecar proxy** (usually Envoy).

🔍 **Key Features of a Service Mesh:**

🔒 **mTLS (mutual TLS)**: Encrypted service-to-service communication

- 🔁 **Traffic Control**: Canary, A/B, blue-green deployments
- 📈 **Metrics and Tracing**: Prometheus, Jaeger, Grafana
- ⚙️ **Policy Enforcement**: Rate limits, access control, retries

Without Service Mesh:

- You must code logic for retries, timeouts, and telemetry.
- TLS is manually configured in each microservice.
- Observability and metrics require custom instrumentation.

With Service Mesh:

- All this is handled automatically via **sidecars** injected into each Pod.
- You configure behavior via **CRDs**, not code.

🔥 Popular Meshes: **Istio**, **Linkerd**, **Consul Connect**, **Kuma**

- **5.2 Installing Istio or Linkerd into Your Cluster**

✅ **Installing Istio (Recommended for Complex Use Cases)**

```
curl -L https://istio.io/downloadIstio | sh -
cd istio-*
export PATH=$PWD/bin:$PATH

istioctl install --set profile=demo -y
kubectl label namespace default istio-injection=enabled
```

- The above enables **automatic sidecar injection** into all Pods in default namespace.

✅ **Installing Linkerd (Lightweight Alternative)**

```
curl -sL https://run.linkerd.io/install | sh
```

linkerd install | kubectl apply -f -

linkerd check

Then:

linkerd inject deployment.yaml | kubectl apply -f -

📦 **Istio = more features** (mTLS, routing, gateway, telemetry)
🧩 **Linkerd = easier to use**, best for small/medium clusters

o 5.3 mTLS, Traffic Routing, and Observability with Sidecars

Once a service mesh is installed and sidecars are injected, your cluster gains **powerful capabilities** instantly.

🔒 Mutual TLS (mTLS)

Enabled by default in Istio:

kubectl get peerauthentication -A

Create a strict policy:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
namespace: orders
spec:
  mtls:
    mode: STRICT
```

mTLS ensures that **only trusted services** inside the mesh can communicate — prevents spoofing.

🔁 Canary Deployment with VirtualService

You can **split traffic** between two versions of a service with no code change.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: orders-route
spec:
  hosts:
    - orders.myapp.com
  http:
    - route:
        - destination:
            host: orders-v1
          weight: 80
        - destination:
            host: orders-v2
          weight: 20
```

Deploy orders-v2 and **watch 20% of real traffic flow to it** — then ramp it up.

Observability: Prometheus + Grafana + Jaeger

Istio installs telemetry out-of-the-box:

- **Prometheus**: collects mesh metrics (latency, errors, throughput)
- **Grafana**: pre-built dashboards for services, workloads, proxies
- **Jaeger**: distributed tracing across all

services Access them:

```
istioctl dashboard prometheus
istioctl dashboard grafana
```

```
istioctl dashboard jaeger
```

📦 Service Mesh Capabilities Summary:

Feature	Description	Benefit
mTLS	Encrypted service-to-service traffic	Secure by default
Traffic Routing	Canary, A/B testing, version shifting	Safer deployments
Observability	Built-in metrics and tracing with zero app code	Debug faster, see service flow clearly
Sidecar Injection	Transparent proxy per Pod (e.g., Envoy)	No code change needed

☑ 6. Enabling Auto-Scaling and Resilience

Your microservices must **scale on demand** and **recover gracefully** from failure. Kubernetes offers powerful primitives like **HPA**, **KEDA**, and **probes** to help your services survive traffic spikes, crashes, and restarts — all automatically.

o 6.1 Horizontal Pod Autoscaler (HPA) with CPU/Memory Metrics

The **Horizontal Pod Autoscaler** automatically increases or decreases the number of Pod replicas based on CPU/memory utilization or custom metrics.

☑ Basic HPA Example (CPU-based):

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: orders-hpa
```

```yaml
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: orders-deployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
```

This keeps CPU usage around 70%. If demand spikes, Kubernetes adds replicas automatically.

☑ **Enabling Metrics Server:**

To use HPA, make sure the **Metrics Server** is installed:

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

- **6.2 Vertical Pod Autoscaler and KEDA for Event Scaling**

While HPA scales **horizontally**, you can also scale based on **event-driven** workloads or **adjust CPU/memory** automatically per Pod.

🔁 Vertical Pod Autoscaler (VPA):

Adjusts resource requests/limits per Pod based on actual usage.

```yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: orders-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment
    name:       orders-deployment
  updatePolicy:
    updateMode: "Auto"
```

⚡ KEDA (Kubernetes Event-Driven Autoscaling):

KEDA enables autoscaling based on **events**, not just metrics — like queue length, database activity, or Kafka lag.

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: orders-keda
spec:
scaleTargetRef:
    name: orders-deployment
  triggers:
    - type: aws-sqs
      metadata:
```

```yaml
      queueURL: https://sqs.ap-south-1.amazonaws.com/123456/orders
    awsRegion: ap-south-1
    queueLength: "10"
```

🚀 When your queue length crosses 10 messages, KEDA will trigger more Pods.

6.3 Liveness, Readiness, and Startup Probes for Recovery

Kubernetes uses **probes** to decide when to:

- Restart your container (liveness)
- Send traffic to it (readiness)
- Wait before probing (startup)

✅ **Example: All 3 Probes**

```yaml
livenessProbe:
  httpGet:
    path: /health
    port: 3000
  initialDelaySeconds: 10
  periodSeconds: 5

readinessProbe:
  httpGet:
    path: /ready
    port: 3000
  initialDelaySeconds: 5
  periodSeconds: 5
```

```yaml
startupProbe:
  httpGet:
    path: /startup
    port: 3000
  failureThreshold: 30
  periodSeconds: 5
```

💡 **Startup Probe** is crucial for apps with slow boot times (e.g., Spring Boot or .NET apps).

📦 **Resilience and Scaling Summary:**

Feature	Use Case	Benefit
HPA	Scale based on CPU/memory	Responds to live resource demand
VPA	Auto-adjust Pod resources	No need to guess CPU/memory limits
KEDA	Event-driven scaling (SQS, Kafka, etc.)	Reactive to business workloads
Probes	Detect startup, readiness, crashes	Ensures uptime and safe traffic

☑ 7. Observability and Monitoring

In a distributed system like microservices, **you can't fix what you can't see**. Observability is about having **metrics, logs, and traces** that let you detect issues, understand behavior, and debug failures across services — all in real time.

7.1 Centralized Logging with Fluent Bit, Loki, or ELK

Kubernetes logs are ephemeral — when a Pod dies, its logs go with it. That's why we need **log aggregation** and centralized storage.

☑ Fluent Bit → Loki (Lightweight & Fast)

- **Fluent Bit**: log collector agent (lightweight, runs as DaemonSet)
- **Loki**: log storage engine (Grafana's project)

helm repo add grafana https://grafana.github.io/helm-charts

helm upgrade --install loki grafana/loki-stack -n monitoring --create-namespace Then configure Fluent Bit to send logs to Loki via Helm or ConfigMap.

☑ ELK Stack (ElasticSearch, Logstash, Kibana)

More heavyweight, but powerful for large-scale setups.

```
helm repo add elastic https://helm.elastic.co
helm install elasticsearch elastic/elasticsearch -n logging
helm install kibana elastic/kibana -n logging
```

Fluent Bit or Filebeat can forward logs from /var/log/containers to Logstash or Elasticsearch.

📝 Standardize all app logs to **JSON** format for easy parsing and indexing.

7.2 Metrics with Prometheus and Grafana Dashboards

Prometheus is the **default monitoring engine** in Kubernetes. It scrapes metrics from services, nodes, and service meshes.

☑ **Install Prometheus + Grafana:**

```
helm repo add prometheus-community
https://prometheus- community.github.io/helm-charts
helm upgrade --install kube-prometheus
prometheus-community/kube- prometheus-stack -n monitoring --
create-namespace
```

This deploys:

- **Prometheus** (metrics storage & scraper)
- **Grafana** (dashboards)
- **Alertmanager** (email/slack/pager

alerts) Access dashboards:

```
kubectl port-forward svc/kube-prometheus-stack-grafana -n monitoring
3000:80
Login: admin / prom-operator
```

☑ **Custom App Metrics**

Expose Prometheus metrics in your app using client libraries:

Language	Library
Node.js	prom-client
Python	prometheus_client
Java	micrometer + prometheus-registry

🎯 Expose at /metrics, and Prometheus will scrape it via ServiceMonitor.

7.3 Tracing and Service Graphs with Jaeger or Istio

Tracing lets you **visualize request flow** through services and pinpoint latency, errors, and bottlenecks.

☑ **Jaeger (Distributed Tracing)**

```
kubectl create namespace tracing
helm install jaeger jaegertracing/jaeger -n tracing
```

Instrument your services using:

- **OpenTelemetry SDKs**
- **Istio or Linkerd** (automatic if sidecars are used)

☑ **Visualizing Trace:**

```
kubectl port-forward svc/jaeger-query -n tracing 16686:16686
```

Visit http://localhost:16686 to:

- Search by service name
- See end-to-end traces
- Identify slow spans and failures

☑ **Istio Service Graph:**

```
istioctl dashboard kiali
```

- See real-time **service-to-service topology**
- Metrics, error rates, and traffic volume

Click into a service → view latency, traces, logs

📦 Observability Stack Summary:

Layer	Tool	Purpose
Logs	Fluent Bit + Loki	View logs across Pods/nodes/services
Metrics	Prometheus + Grafana	View CPU/memory, requests, latency
Traces	Jaeger, OpenTelemetry	Visualize end-to-end request paths
Alerts	Alertmanager	Send notifications based on metrics

☑ 8. Securing and Hardening Microservices

Security is **non-negotiable** in production. In a microservices environment, you must ensure every service has the **least privilege**, secrets are handled safely, and your software supply chain is tamper-proof. Kubernetes provides the tools — you just need to apply them properly.

- **8.1 Role-Based Access Control (RBAC) and Network Policies**

🔐 Role-Based Access Control (RBAC)

RBAC lets you **control what users, pods, or services can do** in a namespace or cluster.

☑ Example: Read-Only Role for Pods

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: dev
rules:
- apiGroups: [""]
```

```
resources: ["pods"]
    verbs: ["get", "watch", "list"]
```

Bind it to a ServiceAccount:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods-binding
  namespace: dev
subjects:
  - kind: ServiceAccount
    name: dev-service-account
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

🔐 Use RBAC to **prevent privilege escalation**, accidental deletes, or namespace crossovers.

🛡 Network Policies

Network Policies restrict **Pod-to-Pod communication** by default.

✅ Example: Deny All Except Internal DB

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-orders-to-db
  namespace: payments
```

```yaml
spec:
  podSelector:
    matchLabels:
      app: db
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: orders
```

🚫 Without this, **any Pod can talk to any Pod** — not safe for production.

8.2 Securing Secrets with Vault or External Secrets Operator

🔑 **Kubernetes Secrets (Basic, but Not Encrypted-at-Rest by Default)**

- Base64 encoded, not encrypted
- Visible via kubectl unless locked down

☑ **HashiCorp Vault (Dynamic Secrets, Strong Encryption)**

- Run Vault in the cluster (HA with Raft backend)
- Enable Kubernetes Auth
- Inject secrets securely via **Vault Agent Injector**

```yaml
annotations:
  vault.hashicorp.com/agent-inject: "true"
  vault.hashicorp.com/role: "orders-app"
  vault.hashicorp.com/agent-inject-secret-db-creds: "internal/data/db/orders"
```

☑ **External Secrets Operator (ESO)**

Sync secrets from:

AWS Secrets Manager

- HashiCorp Vault
- GCP Secret Manager

```yaml
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: orders-db-secret
spec:
  secretStoreRef:
    name: vault-backend
  kind: SecretStore
  target:
    name: orders-secret
  data:
    - secretKey:
      password
      remoteRef:
        key: orders-db-password
```

🎯 **Never store secrets in Git**. Always use an external provider + automated sync.

8.3 Image Scanning and Supply Chain Protection (Trivy, Cosign, SBOMs)

Attackers now target your **build process**, **containers**, and **dependencies**. Secure the entire chain.

✅ **Trivy: Vulnerability & Misconfig Scanner**

```
trivy image
```

myapp/orders:1.0 trivy

config ./k8s/

Finds:

- CVEs in packages and OS
- Hardcoded secrets
- Misconfigured RBAC or open Ingress

✅ SBOM (Software Bill of Materials)

Generate a full dependency list:

```
syft myapp/orders:1.0 -o cyclonedx-json > sbom.json
```

Scan it:

```
grype sbom:sbom.json
```

✅ Cosign: Sign and Verify Images

```
cosign sign --key cosign.key myapp/orders:1.0
cosign verify --key cosign.pub myapp/orders:1.0
```

🔐 Signed images = trustable, tamper-proof containers in the supply chain.

📦 Security & Hardening Summary:

Area	Tool/Practice	Benefit
Access Control	RBAC, ServiceAccounts	Prevents privilege misuse
Network Isolation	NetworkPolicies	Restricts Pod communication
Secret Management	Vault, External Secrets	Keeps secrets encrypted and dynamic
Image Scanning	Trivy, Grype	Detects vulnerabilities early
Supply Chain Security	SBOM + Cosign	Protects against tampering & malware