Kubernetes for Production: Scaling & Monitoring

Operating, Scaling, and Observing Real-World Kubernetes Clusters

Preface

There is a moment every Kubernetes practitioner encounters—sometimes at 2 a.m., sometimes during a quarterly review—when the distance between "running Kubernetes" and "running Kubernetes *well*" becomes painfully clear. A deployment that sailed through staging collapses under real traffic. An autoscaler thrashes instead of stabilizing. An alert fires, but nobody knows what it means or who should respond. The cluster is *working*, technically. But it is not production-ready.

This book was written for that moment, and for everything you need to know to prevent it.

Why This Book Exists

Kubernetes has won. It is the dominant platform for container orchestration, adopted across startups, enterprises, and everything in between. Yet the vast majority of Kubernetes resources focus on getting started—deploying your first pod, writing your first manifest, understanding the basics of Services and Ingress. Far fewer tackle the harder, messier, and ultimately more consequential challenge: *operating Kubernetes clusters that are reliable, scalable, and observable in production*.

Kubernetes for Production: Scaling & Monitoring fills that gap. It is a practitioner's guide to the operational discipline required to run real-world Kubernetes clusters at scale—clusters where downtime has consequences, where workloads must grow and shrink with demand, and where teams need clear signals to diagnose problems before users notice them.

What You Will Find Here

This book is organized around three interconnected pillars of production Kubernetes operations:

Scaling – Chapters 3 and 4 dive deep into Horizontal Pod Autoscaling, Vertical Pod Autoscaling, and cluster-level scaling strategies. You will learn not just *how* these mechanisms work, but *when* to use each one and how to tune them for predictable behavior under real load.

Resilience – Chapters 5 and 6 address high availability, multi-node and multi-zone architectures, and the design decisions that determine whether your Kubernetes workloads survive the inevitable infrastructure failures. Chapters 11 through 14 extend this into resource management, storage and network performance, capacity planning, and the often-underestimated discipline of change management and upgrades.

Observability – Chapters 7 through 10 form the monitoring and incident response core of the book, covering metrics collection, centralized logging, alert design, and structured incident response within Kubernetes environments. These chapters emphasize *actionable* observability—not just collecting data, but building systems that help humans make better decisions faster.

The book opens with foundational context (Chapters 1 and 2) that establishes what "production-ready" truly means and revisits Kubernetes architecture through an operational lens. It closes with a candid look at common production anti-patterns (Chapter 15) and a forward-looking chapter on evolving your organization toward SRE and platform engineering practices (Chapter 16).

The appendices provide immediately usable artifacts: a **Production Readiness Checklist**, **HPA configuration examples**, an **Incident Response Playbook**, **monitoring and alerting templates**, and a **learning roadmap** for continued growth.

Who This Book Is For

This book is for Kubernetes operators, DevOps engineers, SREs, and platform teams who have moved past the basics and are now responsible for clusters that *must not fail quietly*. Prior experience with Kubernetes concepts is assumed; what this book provides is the operational depth to put that knowledge to work under pressure.

Acknowledgments

No book on production systems is written in isolation. This work draws on the collective wisdom of the Kubernetes community—the SREs who have shared their postmortems, the contributors who have refined the autoscaling APIs, and the countless engineers who have debugged failing pods at ungodly hours and then written about what they learned. I am grateful to the technical reviewers whose sharp feedback made every chapter stronger, and to the open-source maintainers whose tools form the backbone of modern Kubernetes observability.

Production is not a destination. It is a discipline—one that demands continuous learning, honest assessment, and deliberate practice. My hope is that this book becomes a trusted companion on that journey, one you reach for not just when things break, but long before they do.

Let's build Kubernetes clusters worth trusting.

Dorian Thorne

Table of Contents

Chapter 1: What Makes Kubernetes "Production-Ready"

Running Kubernetes in a development environment and running it in production are two fundamentally different challenges. In development, you might spin up a single-node cluster with Minikube, deploy a few pods, and call it a day. In production, however, the stakes are dramatically higher. Real users depend on your applications. Downtime translates directly into lost revenue, damaged reputation, and broken trust. The infrastructure must handle unpredictable traffic spikes, recover gracefully from hardware failures, and provide operators with deep visibility into every layer of the system.

This chapter establishes the foundation for everything that follows in this book. Before we dive into the mechanics of scaling, monitoring, alerting, and observability, we need to understand what "production-ready" actually means in the context of Kubernetes. We need to draw a clear line between a cluster that works and a cluster that is truly ready to serve real workloads with confidence.

The Gap Between Development and Production

When engineers first begin working with Kubernetes, they typically start with a local development setup. They install Minikube or kind (Kubernetes in Docker), run

`kubectl apply` on a few manifest files, and watch their containers come to life. The experience feels magical. Kubernetes abstracts away so much complexity that it can create a false sense of readiness. The application runs, the pods are healthy, and everything seems fine.

But consider what happens when you move that same setup into a production environment. Suddenly, you need to answer questions that never arose during development. What happens when a node fails at 3 AM? How do you ensure that a deployment rollout does not cause downtime for your users? How do you prevent one misbehaving application from consuming all the CPU and memory on a shared cluster? How do you know when something is going wrong before your customers notice?

The gap between development and production Kubernetes is not just about scale. It is about reliability, security, observability, and operational maturity. A production-ready Kubernetes cluster is one that has been deliberately configured, hardened, and instrumented to handle the realities of serving real traffic in an unpredictable world.

Let us examine a simple comparison to illustrate the differences:

Aspect	**Development Cluster**	**Production Cluster**
Number of Nodes	1 (single node)	3 or more (multi-node with high availability)
Control Plane	Single instance, no redundancy	Multiple replicas across availability zones
etcd	Single instance, no backups	Clustered, encrypted, with automated backups
Resource Requests and Limits	Often omitted	Mandatory for all workloads
Network Policies	Usually absent	Enforced to restrict pod-to-pod communication

RBAC	Permissive or disabled	Strictly configured with least-privilege access
Monitoring	None or basic kubectl commands	Full observability stack (metrics, logs, traces)
Autoscaling	Not configured	Horizontal Pod Autoscaler and Cluster Autoscaler active
Ingress and TLS	HTTP with port-forwarding	Production-grade ingress controller with TLS termination
Secrets Management	Plain text in manifests	Encrypted at rest, managed through external secret stores
Disaster Recovery	No plan	Documented runbooks, tested backup and restore procedures

This table is not exhaustive, but it captures the essential truth: production Kubernetes requires deliberate effort across every dimension of the system.

The Pillars of Production Readiness

Production readiness for Kubernetes does not emerge from a single configuration change or a single tool. It is the result of disciplined attention to several interconnected pillars. Each pillar represents a critical area that must be addressed before a cluster can be considered truly production-grade.

High Availability and Resilience

The first and most fundamental pillar is high availability. In production, no single component should be a single point of failure. This applies to the Kubernetes control plane itself, to the applications running on the cluster, and to the underlying infrastructure.

For the control plane, this means running multiple replicas of the API server, the scheduler, and the controller manager. It means deploying etcd as a clustered, replicated datastore rather than a single instance. Many managed Kubernetes services such as Amazon EKS, Google GKE, and Azure AKS handle control plane high availability automatically, but if you are running a self-managed cluster, this is your responsibility.

For applications, high availability means running multiple replicas of each workload, spreading those replicas across multiple nodes and availability zones using pod anti-affinity rules, and configuring proper health checks so that Kubernetes can automatically restart or replace unhealthy pods.

Consider this example of a Deployment manifest that incorporates several production-readiness practices:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-application
  namespace: production
  labels:
    app: web-application
    environment: production
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  selector:
    matchLabels:
      app: web-application
  template:
    metadata:
      labels:
        app: web-application
        environment: production
```

```yaml
spec:
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          podAffinityTerm:
            labelSelector:
              matchExpressions:
                - key: app
                  operator: In
                  values:
                    - web-application
            topologyKey: kubernetes.io/hostname
  containers:
    - name: web
      image: myregistry.example.com/web-application:v1.2.3
      ports:
        - containerPort: 8080
      resources:
        requests:
          cpu: "250m"
          memory: "256Mi"
        limits:
          cpu: "500m"
          memory: "512Mi"
      livenessProbe:
        httpGet:
          path: /healthz
          port: 8080
        initialDelaySeconds: 15
        periodSeconds: 10
        failureThreshold: 3
      readinessProbe:
        httpGet:
          path: /ready
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 5
        failureThreshold: 3
      startupProbe:
        httpGet:
          path: /healthz
```

```
      port: 8080
    initialDelaySeconds: 10
    periodSeconds: 5
    failureThreshold: 30
```

Let us walk through the key production-readiness features in this manifest:

Configuration Element	Purpose	Why It Matters in Production
`replicas: 3`	Runs three instances of the application	Ensures availability even if one or two pods fail
`strategy: RollingUpdate`	Gradually replaces old pods with new ones during updates	Prevents downtime during deployments
`maxUnavailable: 1`	Allows at most one pod to be unavailable during rollout	Maintains minimum capacity during updates
`podAntiAffinity`	Spreads pods across different nodes	Prevents all replicas from being lost if a single node fails
`resources.requests`	Declares minimum CPU and memory the pod needs	Enables the scheduler to make intelligent placement decisions
`resources.limits`	Declares maximum CPU and memory the pod can consume	Prevents a single pod from starving other workloads
`livenessProbe`	Checks if the application is still running	Kubernetes restarts the pod if the probe fails
`readinessProbe`	Checks if the application is ready to serve traffic	Removes the pod from service endpoints if it is not ready
`startupProbe`	Checks if the application has finished starting up	Prevents liveness probes from killing slow-starting applications

Note: *The distinction between liveness, readiness, and startup probes is critical in production. A common mistake is to use only a liveness probe, which can cause Kubernetes to restart pods that are simply slow to start. The startup probe was introduced specifically to address this issue, giving applications a generous window to initialize before liveness checks begin.*

Resource Management and Isolation

In a production cluster, multiple teams and applications typically share the same infrastructure. Without proper resource management, a single misbehaving application can consume all available CPU or memory on a node, causing other applications to be evicted or degraded.

Kubernetes provides several mechanisms for resource management. Resource requests and limits on individual containers are the first line of defense. Beyond that, Kubernetes offers LimitRange objects to set default and maximum resource constraints for a namespace, and ResourceQuota objects to cap the total resources that a namespace can consume.

Here is an example of a ResourceQuota that limits the total resources available in a namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: production-quota
  namespace: team-alpha
spec:
  hard:
    requests.cpu: "10"
    requests.memory: "20Gi"
    limits.cpu: "20"
    limits.memory: "40Gi"
    pods: "50"
    services: "10"
    persistentvolumeclaims: "20"
```

And a corresponding LimitRange that sets default values for containers that do not specify their own:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: default-limits
  namespace: team-alpha
spec:
  limits:
    - default:
        cpu: "500m"
        memory: "512Mi"
      defaultRequest:
        cpu: "250m"
        memory: "256Mi"
      type: Container
```

These objects work together to ensure that no single team or application can monopolize cluster resources. In production, this kind of guardrail is not optional. It is essential.

Security Hardening

A production Kubernetes cluster is an attractive target for attackers. It typically runs critical business applications, has access to sensitive data, and is connected to internal networks. Security must be treated as a first-class concern, not an afterthought.

At a minimum, production clusters should implement the following security measures:

Role-Based Access Control (RBAC) should be enabled and configured with the principle of least privilege. Every user and service account should have only the permissions they need and nothing more.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
```

```
  namespace: team-alpha
  name: deployment-manager
rules:
  - apiGroups: ["apps"]
    resources: ["deployments"]
    verbs: ["get", "list", "watch", "create", "update", "patch"]
  - apiGroups: [""]
    resources: ["pods", "pods/log"]
    verbs: ["get", "list", "watch"]
```

Network Policies should be used to restrict pod-to-pod communication. By default, all pods in a Kubernetes cluster can communicate with each other. In production, this is a significant security risk.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-only-frontend
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: backend-api
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: frontend
      ports:
        - protocol: TCP
          port: 8080
```

This NetworkPolicy ensures that the backend API pods only accept incoming traffic from pods labeled as the frontend application. All other ingress traffic is denied.

> ***Note:*** *Network Policies require a CNI (Container Network Interface) plugin that supports them. Not all CNI plugins enforce Network Policies. Calico,*

Cilium, and Weave Net are popular choices that provide full Network Policy support. If your CNI does not support Network Policies, the objects will be created but will have no effect, which is a dangerous false sense of security.

Observability

You cannot manage what you cannot see. Observability is the pillar that connects all the others. Without metrics, logs, and traces, you are flying blind. You will not know when resources are running low, when applications are failing, or when security incidents are occurring.

Production-ready Kubernetes clusters require a comprehensive observability stack that typically includes metrics collection with Prometheus, log aggregation with a solution like Fluentd or Fluent Bit feeding into Elasticsearch or Loki, and distributed tracing with Jaeger or Zipkin. We will explore each of these in great depth throughout this book, but it is important to establish now that observability is not a nice-to-have feature. It is a fundamental requirement for production operations.

The following `kubectl` commands are essential tools for basic cluster health assessment, but they are not a substitute for a proper monitoring stack:

```
# Check the status of all nodes in the cluster
kubectl get nodes -o wide

# View resource utilization across nodes (requires metrics-
server)
kubectl top nodes

# Check the status of all pods across all namespaces
kubectl get pods --all-namespaces

# View resource utilization of pods in a specific namespace
kubectl top pods -n production

# Describe a specific node to see conditions, capacity, and
allocatable resources
kubectl describe node worker-node-01
```

```
# Check for any events that might indicate problems
kubectl get events --sort-by='.lastTimestamp' -n production
```

Command	What It Shows	When to Use It
`kubectl get nodes -o wide`	Node status, IP addresses, OS, kernel version, container runtime	Quick cluster health check
`kubectl top nodes`	Current CPU and memory usage per node	Identifying resource pressure on nodes
`kubectl get pods --all-namespaces`	Status of all pods across the entire cluster	Broad overview of workload health
`kubectl top pods -n production`	CPU and memory consumption per pod	Finding resource-hungry pods
`kubectl describe node`	Detailed node information including conditions and events	Diagnosing node-level issues
`kubectl get events`	Recent cluster events sorted by time	Investigating recent problems or failures

> ***Note:*** *The* `kubectl top` *commands require the Metrics Server to be installed in your cluster. In managed Kubernetes services, this is often pre-installed, but in self-managed clusters, you need to deploy it yourself. Without Metrics Server, these commands will return an error, and more importantly, the Horizontal Pod Autoscaler will not function.*

A Production Readiness Checklist

To bring together everything we have discussed, here is a comprehensive checklist that you can use to evaluate whether your Kubernetes cluster is ready for produc-

tion workloads. This checklist will serve as a roadmap for the remaining chapters of this book, where we will dive deep into each of these areas.

Category	Requirement	Status
High Availability	Control plane components are replicated	Required
High Availability	etcd is clustered with at least 3 members	Required
High Availability	Workloads run with multiple replicas	Required
High Availability	Pod anti-affinity rules distribute pods across nodes	Recommended
High Availability	Nodes span multiple availability zones	Recommended
Resource Management	All containers have resource requests and limits	Required
Resource Management	Namespaces have ResourceQuotas	Recommended
Resource Management	LimitRanges set sensible defaults	Recommended
Resource Management	Horizontal Pod Autoscaler is configured for variable workloads	Recommended
Resource Management	Cluster Autoscaler is configured for node scaling	Recommended
Security	RBAC is enabled and configured with least privilege	Required
Security	Network Policies restrict pod-to-pod communication	Required
Security	Secrets are encrypted at rest	Required
Security	Pod Security Standards are enforced	Required
Security	Container images are scanned for vulnerabilities	Recommended
Security	API server audit logging is enabled	Recommended
Observability	Metrics collection is in place (Prometheus or equivalent)	Required
Observability	Log aggregation is configured	Required

Observability	Alerting rules are defined for critical conditions	Required
Observability	Dashboards provide visibility into cluster and application health	Recommended
Observability	Distributed tracing is available for microservices	Recommended
Operations	Backup and restore procedures for etcd are tested	Required
Operations	Disaster recovery plan is documented and tested	Required
Operations	Runbooks exist for common failure scenarios	Recommended
Operations	CI/CD pipelines handle deployments (not manual kubectl)	Recommended

Practical Exercise: Evaluating Your Cluster

To put this chapter into practice, perform the following exercise on an existing Kubernetes cluster. If you do not have one available, you can create a multi-node cluster using kind with the following configuration:

```
# kind-production-like.yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
  - role: worker
  - role: worker
  - role: worker
```

Create the cluster with:

```
kind create cluster --config kind-production-like.yaml --name
production-eval
```

Once your cluster is running, execute the following commands and record the results:

```
# Step 1: Check how many nodes are available
kubectl get nodes

# Step 2: Verify that RBAC is enabled
kubectl api-versions | grep rbac

# Step 3: Check if metrics-server is installed
kubectl get deployment metrics-server -n kube-system

# Step 4: Look for any pods without resource limits
kubectl get pods --all-namespaces -o json | \
  jq '.items[] | select(.spec.containers[].resources.limits ==
null) | .metadata.name'

# Step 5: Check if any Network Policies exist
kubectl get networkpolicies --all-namespaces

# Step 6: Verify etcd health (for self-managed clusters)
kubectl get pods -n kube-system -l component=etcd
```

For each check, ask yourself: would this cluster survive a node failure at 3 AM without human intervention? Would you know about it within minutes? Could you diagnose and resolve the issue using the tools and information available to you?

If the answer to any of these questions is no, then you have identified an area that needs improvement before the cluster is production-ready.

Setting the Stage for What Comes Next

This chapter has established the conceptual framework for production-ready Kubernetes. We have examined the gap between development and production clusters, explored the pillars of production readiness, and provided concrete examples of the configurations and practices that separate a hobby cluster from one that is ready to serve real users.

In the chapters that follow, we will systematically work through each of these areas in depth. We will build a complete observability stack with Prometheus, Grafana, and alerting. We will configure autoscaling at both the pod and cluster level. We will implement security hardening measures and establish operational procedures for day-two operations.

The journey from a working Kubernetes cluster to a production-ready one is not trivial, but it is well-defined. The tools exist, the patterns are proven, and the community has accumulated years of hard-won knowledge about what works and what does not. This book distills that knowledge into a practical, hands-on guide that will take you from understanding the principles to implementing them with confidence.

Production readiness is not a destination. It is a continuous practice of improvement, measurement, and adaptation. The cluster you build today will need to evolve as your applications grow, as your traffic patterns change, and as new threats emerge. What matters is that you start with a solid foundation and build deliberately from there. That foundation begins with understanding what production-ready truly means, and now you do.

Chapter 2: Kubernetes Architecture Revisited

When you first learned Kubernetes, you likely encountered its architecture as a diagram with boxes and arrows connecting control plane components to worker nodes. That initial understanding served you well for deploying your first applications and getting comfortable with kubectl commands. But running Kubernetes in production demands a much deeper comprehension of how these components interact, fail, recover, and scale. This chapter takes you back to the architectural foundations of Kubernetes, not to repeat what you already know, but to examine each component through the lens of production operations, scaling challenges, and observability requirements.

Understanding the architecture at this depth is not an academic exercise. When your cluster experiences a sudden spike in API server latency at three in the morning, or when etcd starts consuming unexpected amounts of disk I/O, or when the scheduler seems to be making poor placement decisions under load, your ability to diagnose and resolve these issues depends entirely on how well you understand what each component does, how it communicates with other components, and what its failure modes look like.

The Control Plane in Depth

The control plane is the brain of every Kubernetes cluster. It is responsible for maintaining the desired state of the entire system, making scheduling decisions,

responding to events, and exposing the API that every user and component interacts with. In production, the control plane must be treated as the most critical piece of infrastructure in your container orchestration strategy.

The API Server: The Central Nervous System

The kube-apiserver is the only component in the entire Kubernetes architecture that directly communicates with etcd. Every other component, whether it is the scheduler, the controller manager, the kubelet on each node, or your kubectl commands, interacts with the cluster state exclusively through the API server. This design decision has profound implications for production operations.

The API server performs several critical functions beyond simply serving REST requests. It handles authentication, determining who is making a request. It handles authorization, determining whether that identity is allowed to perform the requested action. It runs admission controllers, which can mutate or validate requests before they are persisted. And it manages the serialization and deserialization of objects between their internal representation and the stored format in etcd.

In production environments, the API server is typically run as multiple replicas behind a load balancer. This is not merely for high availability but also for handling the sheer volume of requests that a busy cluster generates. Consider a cluster with 200 nodes, each running a kubelet that watches for pod updates, a kube-proxy that watches for service and endpoint changes, and potentially dozens of controllers and operators, each maintaining their own watches. The number of concurrent watch connections alone can reach into the thousands.

The following table describes the key configuration parameters that affect API server performance in production:

Parameter	Description	Production Consideration
max-requests-inflight	Maximum number of non-mutating requests in flight	Default of 400 may be insufficient for large clusters. Monitor 429 responses to determine if this needs increasing.
max-mutating-requests-inflight	Maximum number of mutating requests in flight	Default of 200. Mutating requests are more expensive because they involve etcd writes.
watch-cache-sizes	Size of the watch cache for each resource type	Larger caches reduce the load on etcd but consume more memory in the API server process.
etcd-servers	Endpoints for etcd cluster	Should use dedicated etcd endpoints with proper TLS configuration. Consider separate etcd clusters for events.
audit-log-path	Path for audit log output	Essential for security and debugging, but can impact performance if the logging backend is slow.
enable-admission-plugins	List of admission controllers to enable	Each admission controller adds latency to API requests. Order matters for performance.
request-timeout	Default timeout for API requests	Default of 60 seconds. Long-running requests like watches have their own timeout handling.

You can inspect the current API server configuration on a kubeadm-based cluster by examining the static pod manifest:

```
cat /etc/kubernetes/manifests/kube-apiserver.yaml
```

To check the current health and responsiveness of your API server, use the following commands:

```
# Check API server health endpoints
kubectl get --raw /healthz
kubectl get --raw /livez
kubectl get --raw /readyz

# Detailed health check showing individual component status
kubectl get --raw '/readyz?verbose'

# Measure API server response latency
kubectl get --raw /api/v1/namespaces/default -v=6
```

Note: In production, you should always monitor the API server request latency histogram, the rate of 429 (Too Many Requests) responses, and the number of active watch connections. These metrics are exposed through the /metrics endpoint and are critical indicators of control plane health.

etcd: The Source of Truth

etcd is a distributed, consistent key-value store that serves as the single source of truth for all cluster state. Every Kubernetes object, whether it is a Pod, a Service, a ConfigMap, a Secret, or a Custom Resource, is stored as a key-value pair in etcd. Understanding etcd's behavior is essential for production Kubernetes operations because etcd performance directly constrains the performance of the entire cluster.

etcd uses the Raft consensus algorithm to maintain consistency across its cluster members. In a typical production deployment, etcd runs as a cluster of three or five members. Three members can tolerate the failure of one member, while five members can tolerate the failure of two. Running more than five members is gener-

ally not recommended because each additional member increases the time required to reach consensus on writes without providing proportional benefit.

The performance characteristics of etcd are heavily dependent on disk I/O latency. etcd must persist every write to its write-ahead log (WAL) before acknowledging the write to the API server. If the disk is slow, every create, update, and delete operation in your Kubernetes cluster will be slow. This is why production etcd deployments should always use dedicated SSD storage, ideally with low-latency NVMe drives.

You can check etcd health and performance using the etcdctl command:

```
# Set etcd environment variables
export ETCDCTL_API=3
export ETCDCTL_ENDPOINTS=https://127.0.0.1:2379
export ETCDCTL_CACERT=/etc/kubernetes/pki/etcd/ca.crt
export ETCDCTL_CERT=/etc/kubernetes/pki/etcd/server.crt
export ETCDCTL_KEY=/etc/kubernetes/pki/etcd/server.key

# Check cluster health
etcdctl endpoint health --write-out=table

# Check cluster status including database size and raft index
etcdctl endpoint status --write-out=table

# Check for slow disk performance by examining WAL sync duration
# This metric is available from etcd's /metrics endpoint
curl -s https://127.0.0.1:2379/metrics --cacert /etc/kubernetes/
pki/etcd/ca.crt \
  --cert /etc/kubernetes/pki/etcd/server.crt \
  --key /etc/kubernetes/pki/etcd/server.key | grep wal_fsync
```

Note: The etcd database has a default storage limit of 2 GB, which can be increased to a maximum of 8 GB. If your database approaches this limit, you need to investigate what is consuming the space. Common culprits include excessive events, large ConfigMaps or Secrets, and Custom Resources that are not being