Kubernetes Internal Architecture Deep Dive

Kubernetes orchestrates containerized workloads through a sophisticated control plane architecture. Understanding the internal flow is essential for debugging production issues, optimizing performance, and designing robust cluster configurations. This presentation provides a comprehensive operational refresher covering core components, networking patterns, security hardening, troubleshooting strategies, and production best practices.

Internal Request Flow: From kubectl to Running Container

When you execute a kubectl command, it triggers a carefully orchestrated sequence of events across multiple control plane components. The kube-api-server acts as the central gateway, receiving and validating all requests. After validation, the request is persisted to etcd, Kubernetes' distributed key-value store. The scheduler then evaluates cluster state and assigns workloads to appropriate nodes based on resource availability, constraints, and policies. Controller managers continuously reconcile desired state with actual state, creating and managing pods as needed. Finally, the kubelet agent on each node pulls container images and instructs the container runtime to start containers. This declarative model ensures self-healing capabilities and enables Kubernetes to maintain your desired cluster state automatically.

01

User submits manifest

kubectl sends YAML to API server over HTTPS with authentication tokens

02

API Server validates

Admission controllers enforce policies, validate schemas, and apply mutations

03

State persisted to etcd

Object definition stored in distributed database with version control

04

Scheduler assigns Node

Algorithm evaluates resources, taints, tolerations, and affinity rules

05

Kubelet starts container

Node agent pulls image from registry and instructs runtime to create container

06

CNI assigns networking

Pod receives unique IP address and network configuration from plugin

Service Exposure Patterns: NodePort, LoadBalancer, and Ingress

Choosing the Right Pattern

Kubernetes offers three primary methods for exposing applications to external traffic, each designed for specific use cases. NodePort opens a static port on every node in the cluster, routing traffic to your service. While simple to configure, this approach exposes infrastructure details and requires manual load balancing. LoadBalancer provisions cloud-specific external load balancers, providing a single entry point with automatic traffic distribution. Ingress controllers offer the most sophisticated approach, enabling domain-based routing, TLS termination, and path-based routing for HTTP/HTTPS traffic.

For production web applications, Ingress is overwhelmingly the best choice. It consolidates multiple services behind a single load balancer, reduces cloud costs, and provides advanced routing capabilities. Popular implementations include NGINX Ingress Controller, Traefik, and cloud-native solutions like GKE's built-in ingress.

Type	Use Case
NodePort	Opens port (30000-32767) on every node, useful for development and testing
LoadBalancer	Provisions cloud external load balancer, provides single IP for service
Ingress	Domain-based routing for HTTP/HTTPS, supports path-based routing and TLS

Ingress is the production-grade solution for web applications, consolidating multiple services behind intelligent routing rules while minimizing infrastructure costs.

Advanced Service Concepts and Networking Primitives

Headless Services

Services with clusterIP: None bypass kube-proxy and return Pod IPs directly via DNS. Essential for stateful applications like databases, message queues, and distributed systems that require direct pod-to-pod communication. Used extensively with StatefulSets for stable network identities.

CNI Architecture

Container Network Interface plugins handle Pod IP assignment and enable Pod-to-Pod communication across nodes. Popular implementations include Calico for network policies, Flannel for simplicity, and Cilium for eBPF-based networking with enhanced observability and security.

kube-proxy Modes

kube-proxy implements Service networking using iptables (default) or IPVS mode. IPVS offers better performance at scale with connection-based load balancing algorithms. It maintains virtual IP routing tables and handles traffic distribution to healthy backend pods.

NetworkPolicy Fundamentals

NetworkPolicies act as internal cluster firewalls, controlling which pods can communicate with each other. They're essential for implementing zero-trust security models and enforcing microsegmentation. Policies specify ingress and egress rules based on pod selectors, namespaces, and IP blocks.

Service Mesh Benefits

Service meshes like Istio and Linkerd provide advanced traffic management, mutual TLS encryption between services, and comprehensive observability. They inject sidecar proxies that handle cross-cutting concerns without modifying application code.

Health Checks and Application Lifecycle Management

Probe Type	Purpose and Behavior
Liveness	Determines if container is running properly. Failure triggers container restart. Use for detecting deadlocks or hung processes that can't self-recover.
Readiness	Determines if container is ready to serve traffic. Failure removes pod from service endpoints without restarting. Essential for gradual application initialization.
Startup	Delays liveness and readiness checks for slow-starting applications. Prevents premature restart during initialization. Set longer intervals for legacy apps with extended startup times.

Pod Disruption Budgets

PDBs ensure minimum pod availability during voluntary disruptions like node drains, cluster upgrades, and autoscaling operations. Specify either minAvailable (e.g., "2") or maxUnavailable (e.g., "25%") to control how many pods must remain running. Critical for maintaining service availability during maintenance windows.

Graceful Shutdown

Kubernetes sends SIGTERM to containers during pod termination, waiting for the grace period (default 30s) before sending SIGKILL. Applications should handle SIGTERM to complete in-flight requests, close database connections, and clean up resources properly.

High Availability and Leader Election Mechanisms

Leader Election in Control Plane

Controller Manager and Scheduler components use leader election to prevent multiple active instances from conflicting. Only one replica actively processes events while others remain on standby. This pattern uses lease objects in the coordination.k8s.io API group, with automatic failover if the leader becomes unresponsive. The mechanism prevents duplicate operations and ensures consistent cluster state management.

etcd Quorum Requirements

etcd requires a quorum (majority of nodes) for write operations, ensuring consistency in the face of failures. A 3-node etcd cluster tolerates 1 failure, while 5 nodes tolerate 2 failures. Regular backups are critical because etcd stores all cluster state, configurations, secrets, and metadata. Loss of etcd data means complete cluster state loss, requiring full cluster rebuild from backups.

Multi-Master Topologies

Production clusters deploy multiple control plane nodes for high availability. API servers run active-active behind a load balancer, while controller managers and schedulers use leader election. This topology eliminates single points of failure and enables zero-downtime control plane upgrades.

Workload Distribution Patterns: DaemonSets vs Deployments

DaemonSet Characteristics

- One pod per node: Automatically schedules exactly one pod on each node, including new nodes as they join the cluster
- Infrastructure workloads: Ideal for logging agents (Fluentd), monitoring collectors (Datadog, Prometheus Node Exporter), network plugins, and storage daemons
- Node-level operations: Perfect for tasks that need to run on every node regardless of other workloads
- Ignores taints: Can be configured to tolerate all taints and schedule even on control plane nodes

Deployment Characteristics

- Flexible replica count: Runs any number of pods distributed across available nodes based on scheduler decisions
- Application workloads: Designed for stateless applications like web servers, APIs, and microservices
- Selective scheduling: Uses node selectors, affinity, and anti-affinity rules for intelligent pod placement
- Rolling updates: Supports sophisticated update strategies with rollback capabilities

DaemonSets run everywhere by design, ensuring consistent infrastructure services across all nodes. Deployments run selectively, distributing application pods according to resource availability and placement constraints. Understanding these patterns is crucial for designing resilient, well-architected Kubernetes systems.

Container Runtimes and the CRI Evolution

1 Docker Era (Pre-1.20)

Kubernetes originally used Docker Engine through dockershim, adding complexity and maintaining dependencies on Docker-specific features.

2 CRI Standardization

Container Runtime Interface defined a standard API for Kubernetes to interact with any compliant runtime, removing Docker-specific coupling.

3 Dockershim Deprecated (1.20)

Kubernetes deprecated built-in Docker support, encouraging migration to CRI-compliant runtimes like containerd and CRI-O.

4 Current State (1.24+)

containerd and CRI-O are now the standard runtimes. Docker images still work perfectly because they're OCI-compliant container images.

containerd Benefits

containerd is a lightweight, industry-standard runtime that powers both Docker Desktop and standalone Kubernetes clusters. It offers better performance, smaller attack surface, and reduced resource overhead compared to full Docker Engine. Most cloud providers use containerd as their default runtime.

Important Clarification

Kubernetes no longer uses Docker Engine directly, but Docker images work perfectly fine. The OCI (Open Container Initiative) image specification ensures compatibility across all runtimes. You can still build with Docker and run on Kubernetes without any issues.

Autoscaling Strategies and Resource Management

Autoscaler Type	What It Scales	Typical Use Case
HPA (Horizontal Pod Autoscaler)	Number of pod replicas	Scale web apps based on CPU, memory, or custom metrics like request rate
VPA (Vertical Pod Autoscaler)	Pod resource requests/limits	Right-size containers that need more CPU/memory, especially for stateful apps
Cluster Autoscaler	Number of nodes in cluster	Add nodes when pods are pending due to insufficient resources, remove underutilized nodes

HPA Configuration

HPA monitors metrics every 15 seconds (by default) and scales deployments based on target utilization. Supports CPU, memory, and custom metrics from external sources. Configure with targetCPUUtilizationPercentage or custom metrics from Prometheus, Datadog, or other observability platforms.

VPA Limitations

VPA requires pod restarts to apply new resource requests, making it less suitable for latency-sensitive workloads. Cannot be used simultaneously with HPA on CPU/memory metrics. Best for batch jobs and applications that tolerate occasional restarts.

Cluster Autoscaler Considerations

Works with cloud provider APIs (GKE, EKS, AKS) to provision/deprovision nodes. Respects Pod Disruption Budgets during scale-down. Configure appropriate scale-down delay to prevent flapping during traffic fluctuations.

Production Security Hardening and Best Practices

Security Fundamentals

RBAC Enforcement

Enable Role-Based Access Control to restrict who can access which resources. Follow principle of least privilege.

Private API Server

Restrict API server access to private networks or VPN. Use authorized networks and firewall rules.

No Root Containers

Run containers as non-root users. Set securityContext with runAsNonRoot: true and drop unnecessary capabilities.

Network Policies

Implement microsegmentation with NetworkPolicies. Default-deny all traffic, then explicitly allow required communication.

Secrets Management

Never store secrets in plain YAML or environment variables. Kubernetes Secrets are base64-encoded but not encrypted at rest by default. For production environments, integrate external secret management systems:

- HashiCorp Vault: Industry-standard secrets management with dynamic secrets, encryption as a service, and audit logging
- Cloud KMS: AWS Secrets Manager, GCP Secret Manager, Azure Key Vault provide native cloud integration
- Sealed Secrets: Bitnami Sealed Secrets encrypt secrets with asymmetric cryptography, safe to commit to Git
- External Secrets Operator: Synchronizes secrets from external systems into Kubernetes

Image Security

Scan container images for vulnerabilities using Trivy, Anchore, or cloud-native scanners. Implement admission controllers to block vulnerable images. Use private registries and image pull secrets. Enable image signing and verification.