Kubernetes

1. Kubernetes Architecture: The Foundation

- Kubernetes follows a client-server model, separating responsibilities into:
- **Control Plane** (Master Node): Manages cluster state and orchestrates workloads.
- **Worker Nodes** (Data Plane): Host and run containerized applications.
- This separation ensures scalability, modularity, and high availability.

2.1. Control Plane Components

The Control Plane is the "brain" of the cluster, managing its state and orchestrating workloads.

- **API Server (kube-apiserver): The Cluster's Front Door**
- Serves as the central communication hub and primary interface for all cluster interactions.
- All requests (from kubectl, REST API, or other components) pass through it.
- Crucial for validating and authorizing requests, ensuring legitimate actions.
- Maintains continuous communication with other Control Plane components and Kubelet on Worker Nodes.
- Processes requests from the Scheduler and interacts with the Controller Manager.
- **etcd: Distributed State Storage**
- A distributed, highly consistent, and fault-tolerant key-value store.
- Definitive repository for the entire Kubernetes cluster's state and configuration data.
- Stores cluster configurations, endpoint details, service definitions, and secrets.
- Reliability and consistency are paramount for cluster stability.

- All Control Plane components interact with etcd via the API Server for consistent state view.
- **Scheduler (kube-scheduler): Pod Placement Logic**
- Continuously monitors the API Server for new Pods without assigned nodes.
- Selects the optimal node for each Pod based on:
- Resource requests (CPU, memory).
- Node constraints (taints, tolerations).
- Affinity and anti-affinity rules.
- Data locality.
- Pod priority.
- Employs algorithms to maximize resource utilization without overloading nodes.
- Communicates decisions back to the API Server.
- **Controller Manager (kube-controller-manager): Orchestration Loops**
- Executes "controllers" that monitor the cluster's actual state against the desired state.
- Initiates corrective actions upon discrepancy to maintain desired configuration.
- Underpins Kubernetes' self-healing and automation capabilities.
- Notable controllers include:
- **Node Controller:** Monitors node health and redistributes workloads if a node fails.
- **Replication Controller (or ReplicaSet Controller):** Ensures the specified number of Pod replicas are running.
- **Service Controller:** Manages Services and load balancing.
- **Endpoints Controller:** Manages communication pathways among services.
- Interacts with the API Server for resource utilization adjustments and lifecycle management.

- **Cloud Controller Manager (cloud-controller-manager - optional): Cloud-Specific Integrations**
- Optional component for managing interactions with underlying cloud providers (AWS, GCP, Azure).
- Responsible for provisioning and managing cloud-specific resources (load balancers, storage volumes, node lifecycle).
- Not required for on-premises Kubernetes deployments.
- Translates Kubernetes-native requests into cloud provider API calls.

2.2. Worker Node Components

Worker Nodes (Minions/Compute Nodes) are machines that host and run containerized applications (Pods).

- **Kubelet: Node Agent for Pod Management**
- Primary agent on every Worker Node.
- Crucial communication link between Control Plane and Worker Node.
- Receives Pod definitions from the API Server and ensures containers run correctly.
- Responsibilities: pulling container images, initiating/terminating containers, monitoring health, reporting status to API Server.
- Relies on the Container Runtime for container execution.
- **Kube-proxy: Network Proxy for Services**
- Network proxy component on every Worker Node.
- Manages network rules (using iptables or IPVS) for seamless communication among Pods and with external entities.
- Vital for network connectivity and load balancing for Services.
- Creates IP routing rules to direct traffic to appropriate Pods.
- Enables Pod communication across nodes and exposes Services.
- Provides load-balancing by distributing requests across service endpoints.

- **Container Runtime: Container Execution Engine**
- Software component directly responsible for container execution on Worker Nodes.
- Supports OCI-compliant runtimes like containerd and CRI-O (Docker was an early choice).
- Interfaces with the host OS kernel to create, manage, and isolate containers.
- Pulls container images from registries and runs them based on Kubelet instructions.
- Ensures container isolation to prevent conflicts and vulnerabilities.

2.3. Interactions: How Components Communicate to Maintain Desired State

- **Declarative Model:** Users define desired application state (e.g., image, replicas, resources) in YAML, submitted to the API Server.
- **API Server:** Central communication hub, processing all interactions from kubectl, other Control Plane components, and Kubelets.
- **Scheduler:** Monitors API Server for new Pods, selects optimal Worker Node, and updates scheduling decision back to API Server.
- **Kubelet:** On selected Worker Node, receives Pod definition from API Server, instructs Container Runtime to pull images and run containers.
- **Continuous Feedback Loop:** Kubelet monitors Pod and node health, reporting status regularly to API Server, which updates etcd.
- **Controller Manager:** Continuously observes state in etcd (via API Server), detects discrepancies between desired and actual states, and takes corrective actions (e.g., restarting failed Pods, scaling deployments).
- **Kube-proxy:** Establishes and maintains network rules on Worker Nodes for inter-Pod communication and network accessibility of applications.

3. Core Kubernetes Objects: Building Blocks of Applications

Kubernetes manages applications by abstracting infrastructure into API objects representing the desired state.

- **Pods: Smallest Deployable Unit, Shared Resources**
- Fundamental building blocks and smallest deployable units in Kubernetes.
- Encapsulate one or more tightly coupled containers sharing network namespace, storage (volumes), and run specifications.
- All containers within a Pod are guaranteed to be scheduled on the same host (node).
- Provide higher abstraction than direct container management, simplifying deployment and scaling.
- Essential for efficient communication and data exchange among tightly coupled containers.
- Primary units for resource management (CPU/memory requests/limits) and security controls.
- **Pod Lifecycle:** Pods transition through Pending, Running, Succeeded, Failed, Unknown phases and Ready, Initialized, ContainersReady, PodScheduled conditions.
- **Pod Templates:** Reusable blueprints for creating new Pods, defining containers, resources, security, etc.. Changes only affect newly created Pods, enabling rolling updates.
- **Deployments: Managing Application Lifecycle and Scaling**
- Higher-level object managing a set of Pods, typically for stateless workloads.
- Provides declarative mechanism for updating Pods and ReplicaSets.
- Instructs Kubernetes on how to create, manage, and update application instances.
- Ensures a specified number of Pod replicas are running (via ReplicaSets) and offers self-healing.
- Simplifies horizontal scaling by adjusting replica count.
- Key use cases:

- Rollout of a ReplicaSet.
- Declaring new application states (e.g., image updates).
- Facilitating rollbacks to previous revisions.
- Dynamically scaling applications.
- Pausing/resuming rollouts.
- Automatically cleaning up old ReplicaSets.
- Deployments exemplify the "Controller Pattern," managing ReplicaSets which manage Pods.
- **Services: Network Abstraction for Pods**
- Crucial abstraction defining a logical set of Pods and their access policy.
- Enables loose coupling between dependent Pods, allowing interaction without ephemeral IPs.
- Provides a stable network endpoint (consistent IP, resolvable DNS name) for ephemeral Pods.
- Offers load balancing, distributing requests across healthy Pods.
- Types of Services:
- **ClusterIP:** Default, stable internal IP, accessible only within the cluster; for internal microservices.
- **NodePort:** Exposes Service on a static port across all Nodes, accessible externally.
- **LoadBalancer:** Integrates with cloud provider load balancers for external exposure; standard for cloud environments.
- **ExternalName:** Maps Service to an external DNS name (CNAME record); for accessing external services.
- **Headless Service:** No ClusterIP; for stateful applications or direct Pod access.
- Kubernetes networking is multi-layered: Pods are isolated, Services expose them, Ingress Controllers provide Layer 7 routing, and Network Policies act as firewalls.

- **Namespaces: Logical Isolation for Resources**
- Logically divide and manage cluster resources among users, teams, or projects.
- Create isolated "virtual subdivisions" within a single cluster.
- Crucial for preventing resource name collisions and controlling access.
- Primary purpose: improve cluster organization, simplify configurations, enhance security.
- Each Namespace has its own resources, policies, and RBAC.
- Predefined system namespaces: kube-system, kube-public, kube-node-lease.
- Key features: unique scope for names, RBAC enforcement, resource allocation (quotas/limit ranges), network policies, labels/annotations, scoped kubectl view.
- **ConfigMaps & Secrets: Externalizing Configuration and Sensitive Data**
- **ConfigMaps:**
- Store non-sensitive configuration data as key-value pairs.
- Decouple application code from configurations, enhancing portability.
- Use cases: storing database addresses, API URLs, feature flags.
- Pods access data via mounted files, environment variables, or command-line arguments.
- Stored in etcd in plain text; size limit of 1 MB.
- **Secrets:**
- Hold small amounts of sensitive data (passwords, tokens, keys).
- Prevent embedding confidential info in code, Pod specs, or images.
- Use cases: providing credentials, pulling images from private registries, setting sensitive environment variables.
- Stored in etcd as base64-encoded (obfuscation, not encryption).
- True encryption at rest often requires external solutions.

- Types: Opaque, TLS, Docker Registry Secrets.
- **Persistent Volumes (PVs) & Persistent Volume Claims (PVCs): Managing Durable Storage**
- **Persistent Volume (PV):**
- Represents storage capacity provisioned statically or dynamically via Storage Class.
- Lifecycle independent of Pods; data persists even if Pod is deleted.
- Access modes:
- **ReadWriteOnce (RWO):** Read-write by a single node.
- **ReadOnlyMany (ROX):** Read-only by multiple nodes.
- **ReadWriteMany (RWX):** Read-write by multiple nodes (not common for cloud disks).
- **ReadWriteOncePod:** Read-write by a single Pod only.
- **Reclaim Policy:** Determines what happens to storage after PVC deletion (Delete, Retain, Recycle).
- **Persistent Volume Claim (PVC):**
- Request for storage by a user or application.
- Defines desired size, access mode, and optional StorageClass.
- Binds to a PV that satisfies requirements.
- Pods use PVCs as volumes.
- Persistent storage is critical for stateful applications, as local storage is ephemeral.
- **Storage Classes: Dynamic Storage Provisioning**
- Define and describe different classes/tiers of storage in a cluster.
- Map to QoS levels, backup policies, etc..
- Enable **dynamic provisioning** of PVs, automatically creating storage when a PVC references a Storage Class.

- Each Storage Class specifies a provisioner (volume plugin/external storage system).
- volumeBindingMode: Delayed binding (or WaitForFirstConsumer) binds PV when Pod is scheduled; Immediate binding binds immediately.
- **Other Storage Types:**
- **Ephemeral Storage:** Temporary, not durable; for caching or temporary files.
- **Projected Storage:** Maps multiple existing sources (Secrets, ConfigMaps) into one directory within a Pod.
- **Block Storage:** Direct, raw disk access; low latency; for high-performance apps like databases.
- **File Storage:** Shared file system access across multiple pods; for shared configurations, logs, user content.
- **Object Storage:** HTTP-accessible; for large unstructured data (backups, archives, CDN, big data).

4. Kubernetes Networking: Connectivity and Control

Kubernetes implements a sophisticated networking model for communication and traffic management.

- **Service Discovery & DNS: How Services Find Each Other**
- Enables application components to communicate without hardcoding changing IP addresses.
- **DNS** is the primary mechanism, automatically assigning DNS records to Services.
- Pods communicate using stable service names (e.g., my-service.default.svc.cluster.local).
- Mechanisms:
- **Services:** Act as internal load balancers, providing stable endpoints.
- **DNS:** Built-in DNS service (CoreDNS) assigns DNS records to Services.

- **Endpoints API:** For API-aware clients, allows direct discovery of Pod IPs/ports; etcd is the service registry.
- **Environment Variables:** Kubernetes injects variables mapping Service names to IPs/ports into Pods.
- **Container Network Interface (CNI): Network Plugin Architecture**
- Standardized specification for managing network resources in a cluster.
- Simplifies interaction between Kubernetes and CNI-based software, allowing pluggable network architecture.
- Provides robust network connectivity between Pods and between Pods and hosts.
- Common CNI plugins:
- **Calico:** Comprehensive networking and network policy, supports various network options.
- **Cilium:** eBPF-based networking, observability, and security solution; can replace kube-proxy.
- **Flannel:** Overlay network provider, simplifies network setup.
- Others: Canal, Weave Net, Antrea, Multus, OVN-Kubernetes.
- **Ingress Controllers: External Access and Advanced Routing**
- Dedicated load balancer for Kubernetes clusters.
- Abstracts complexities of routing HTTP/HTTPS traffic from external sources to Services.
- Configured via Kubernetes Ingress Resources.
- Main functions:
- Traffic reception and load balancing to Pods.
- Egress traffic management.
- Automatic rule updates as Pods/Services change.
- Secure access over HTTP/HTTPS.
- Centralized routing control (host-based, path-based, traffic splitting).

- Single entry point for incoming traffic.
- SSL/TLS Termination and health checking.
- Limitations: Primarily Layer 7 (HTTP/HTTPS) traffic; typically single-namespace scoped.
- Implementations: NGINX Ingress Controller, Istio Ingress, Traefik.
- **Network Policies: Firewall Rules for Pod Communication**
- Standardized Kubernetes objects to control and restrict network traffic patterns.
- Govern communication between Pods/Namespaces and traffic in/out of the cluster.
- Implement "defense in depth" and secure multi-tenancy by defining Pod-level firewall rules.
- By default, all Pods can communicate freely ("default allow"). Network Policies allow changing to "default deny" and explicitly permitting traffic.
- Allow granular control over ingress (entering Pod) and egress (leaving Pod) rules.
- Relies on a CNI plugin that supports the Network Policy API (e.g., Calico or Cilium).
- Applied to target Pods using podSelector labels; namespaceSelector for cross-namespace communication.
- externalTrafficPolicy influences how Network Policies apply by determining source IP seen by Pods.

5. Deployment Strategies: Managing Application Updates

Kubernetes offers various strategies to balance availability, risk, and resource utilization during updates.

- **Rolling Update: Gradual, High-Availability Updates**
- Default and most common deployment strategy.

- Updates Pods gradually, one or a few at a time, while others run the previous version.
- Ensures some application instances remain available, minimizing downtime.
- Kubernetes waits for each updated Pod to become healthy before proceeding.
- **Advantages:**
- High Availability: Majority of application remains operational.
- Reduced Downtime: Continuous service maintained.
- Gradual Rollback: Issues can be reverted gradually.
- Lower Risk: Less risky than "recreate" strategy.
- **Disadvantages:**
- Configuration Complexity: Requires careful planning.
- Longer Update Process: Gradual nature takes more time.
- Transient Inconsistencies: Old and new versions run concurrently, potential for temporary issues.
- Legacy Application Challenges: Problematic for apps not designed for concurrent versions.
- **Key Parameters:**
- maxUnavailable: Max number/percentage of Pods unavailable during update (default 25%).
- maxSurge: Max number/percentage of Pods created *above* desired count (default 25%).
- **Blue/Green Deployment: Zero-Downtime, Full-Version Switch**
- Runs two identical production environments concurrently: "blue" (current) and "green" (new).
- Traffic initially to "blue"; "green" is tested. Once verified, traffic instantly switches from "blue" to "green".
- "Blue" can be retained for rollback or decommissioned.
- **Advantages:**

- Zero Downtime: Minimal to no downtime during updates.
- Immediate Rollback: Instant redirection to stable "blue" if issues arise.
- Simplified Testing: Isolated production replica for thorough testing.
- Avoids Versioning Issues: Entire application state switched at once.
- Compliance Support: Transparent and reversible changes.
- **Disadvantages:**
- Resource Overhead: Requires double compute, storage, networking resources.
- "All or Nothing" Switch: No gradual exposure; problems might surface after full promotion.
- Database Schema Upgrades: May still require database downtime.
- Increased Attack Surface: Running parallel environments.
- Automation Requirements: Needs robust automation.
- **Kubernetes Implementation:** Leverages Namespaces for isolation, Deployments for rollout, Services to direct traffic, and Ingress controllers/external load balancers for switching.
- **Canary Release: Phased Rollout to a Subset of Users**
- Progressive delivery model to reduce risk of new software versions.
- Gradually rolls out new version to a small, controlled subset of real users.
- Monitors performance and behavior before wider expansion.
- Issues contained to "canary" group, allowing quick rollback.
- **Advantages:**
- Risk Mitigation: Impact of bugs contained to limited user base.
- Real-time User Feedback: Collects production data and feedback.
- Controlled Traffic Exposure: Fine-grained control over traffic flow.
- Cost Efficiency: Uses existing production environment for testing.
- **Disadvantages:**

- Not Out-of-the-Box: Kubernetes doesn't natively support; requires external tools (service meshes, progressive delivery tools, advanced Ingress).
- Concurrent Versioning: Application must run multiple versions concurrently.
- Complex Traffic Management: Requires sophisticated Layer 7 routing.
- **Use Cases:** Feature rollouts, configuration changes, user-specific deployments, multi-region deployments, third-party service updates.
- **Kubernetes Implementation:** Typically involves two ReplicaSets (old and "canary") and traffic routing via Ingress Controller annotations or service mesh.

5.4. Table: Comparison of Kubernetes Deployment Strategies

Strategy	Description	Pros	Cons
Rolling Update	Gradually replaces old Pods with new ones, maintaining application availability. Default strategy.	High availability, minimal downtime, gradual rollback capability, less risky than recreate.	Can be complex to configure, longer update process, potential for transient inconsistencies (old/new versions running simultaneously), problematic for legacy apps not designed for concurrent versions.
Blue/Green Deployment	Runs two identical production environments ("blue" and "green"). Traffic is switched entirely to the new "green" environment after testing.	Zero downtime, immediate rollback capability, simplified testing in isolated environment, avoids versioning issues, supports compliance.	High resource overhead (double resources), "all or nothing" switch (no gradual exposure), database schema upgrades may still require downtime, increased attack surface, requires robust automation.
Canary Release	Gradually rolls out a new version to a small subset of users, monitoring	Minimizes risk (impact contained to subset), real-time user feedback, controlled traffic	Not natively supported (requires external tools/configurations), application must support concurrent versions,

	performance before wider release.	exposure, cost-efficient (uses production for testing).	requires sophisticated traffic management.

6. Kubernetes Security: Protecting Your Cluster and Workloads

Securing Kubernetes requires a multi-layered approach from access control to workload isolation.

- **Role-Based Access Control (RBAC): Fine-Grained Authorization**
- Foundational security layer regulating access to Kubernetes API and resources.
- Defines user roles with precise permissions, controlling who can view/interact with resources and what actions they can perform.
- Enforces **Principle of Least Privilege (PoLP)**: granting minimum necessary permissions.
- RBAC process:

1. **Authentication:** Verifies identity (users, groups, service accounts).
2. **Authorization:** Checks authenticated entity's permissions.
3. **Admission Control:** Intercepts API requests before persistence, ensuring policy compliance.

- RBAC rule elements: API group, verb (action), target resource.
- Permissions defined at namespace level (RoleBindings) or cluster-wide (ClusterRoleBindings).
- **Best Practices for RBAC:**
- Enforce PoLP: Grant only explicitly required permissions; avoid wildcards.
- Regular Review: Periodically review and update permissions.
- Use Namespaces for Scope: Assign permissions at namespace level to contain breaches.

- Audit and Monitor: Continuously audit/monitor RBAC events for anomalies.
- Avoid cluster-admin: Use only when absolutely necessary; avoid system:masters group.
- Minimize Privileged Tokens: Limit powerful service accounts on Pods.
- Secure Sensitive Operations: Implement additional verification.
- Automate with Policy as Code: Integrate RBAC into GitOps pipelines.
- Be Aware of Privilege Escalation Risks: Certain permissions can lead to escalation.
- **Pod Security Standards (PSS): Enforcing Workload Isolation**
- Define and enforce security restrictions for Kubernetes workloads.
- Introduced in v1.23, replaced Pod Security Policies (PSPs) from v1.25.
- PSS categorizes workloads into security levels; PSAs describe requirements for Pod security contexts.
- Migration from PSPs requires transitioning to another workload security mechanism and mapping to PSS.
- Security levels:
- **Privileged:** Unrestricted policies.
- **Baseline:** Minimally restrictive, prevents known privilege escalations.
- **Restricted:** Heavily restricted, follows Pod hardening best practices.
- Ensure Pods and containers are appropriately isolated.
- **Secrets Management Best Practices: Handling Sensitive Data Securely**
- Crucial for security posture and operational integrity.
- Kubernetes Secrets provide basic storage, but base64 encoding is obfuscation, not encryption.
- **Best Practices:**
- Avoid Hardcoding: Never embed secrets in code, images, or Pod specs.

- Encryption at Rest and in Transit: Enable encryption for Secret data in etcd and during transit (TLS).
- Implement RBAC: Tightly control access to secrets.
- Least Privilege Principle: Grant only necessary secret access.
- Regular Rotation: Change and update secrets frequently, automate rotation.
- Audit and Monitor Access: Maintain detailed logs of secret interactions.
- Isolate with Namespaces and Anti-Affinity: Limit secret access and minimize storage on single nodes.
- Use Dedicated Secret Management Tools: Integrate external systems (HashiCorp Vault, Azure Key Vault, AWS Secrets Manager) for enhanced features.
- Resource Limits: Set size limits (under 1MB) and enforce namespace quotas for Secrets.

7. Monitoring and Logging: Gaining Visibility into Your Cluster

Effective monitoring and logging are indispensable for maintaining cluster health, performance, and security.

- **Monitoring Tools: Prometheus & Grafana**
- **Prometheus:**
- Open-source systems monitoring and alerting toolkit.
- Collects and stores metrics as time series data using a pull-based model.
- Designed for reliability, allowing diagnosis during outages.
- Excels in dynamic service-oriented architectures with multi-dimensional data.
- Key Components: Prometheus server, client libraries, Pushgateway, exporters, Alertmanager.
- Common Metrics: Node CPU/memory/disk/network, Pod CPU/memory, network traffic, app performance.

- **Grafana:**
- Open-source visualization and alerting platform with out-of-the-box Prometheus support.
- Creates rich, interactive dashboards for time series data visualization.
- Supports alerting rules and notification channels based on Prometheus metrics.
- **Best Practices for Monitoring:**
- Collect Comprehensive Metrics: Resource utilization, app performance, network stats.
- Set Up Alerts and Thresholds: Proactive alerts for potential issues.
- Monitor Control Plane: Health and performance of API Server, Scheduler, Controller Manager, etcd, Kube-DNS.
- Instrument Applications: Inject instrumentation libraries into containers.
- Dig Deep: Go beyond surface-level to granular data (process interactions, ports, files, memory, network).
- Historical Data: Capture historical data beyond metrics for debugging.
- **Logging Solutions: Centralized Log Aggregation**
- Essential due to ephemeral nature of Pods/containers (local logs disappear).
- Aggregates logs from various sources into a persistent datastore for analysis, troubleshooting, and auditing.
- **Log Sources:**
- Application Logs: From non-system containers (stdout/stderr).
- System Logs: From kubelet, container runtime, kube-proxy, kube-dns, etc..
- Audit Logs: Security-relevant records from Kubernetes API server.
- Control Plane Logs: From kube-apiserver, kube-scheduler, kube-controller-manager.
- Events: Real-time notifications of significant occurrences.
- **Aggregation Mechanisms:**

- GKE Cloud Logging Integration: Default integration with per-node logging agent.
- Fluentd/Fluent Bit: Open-source log collectors deployed as DaemonSets to forward logs to backends (Elasticsearch, Splunk).
- Oracle Logging Analytics: Collects metrics, object info, and logs from Kubernetes components, OS, containers.
- **Best Practices for Logging:**
- Structured Logging: Use JSON for easier parsing and searching.
- Metadata Tagging: Add context (Pod name, namespace, container) for filtering.
- Filtering and Rate Limiting: Filter noisy messages to manage volume and costs.
- Centralized Storage: Ensure logs are collected centrally.
- Alerting on Logs: Set up alerts based on log patterns.

8. Troubleshooting: Diagnosing and Resolving Issues

Effective troubleshooting requires a systematic approach using kubectl commands.

- **General Troubleshooting Steps:**

1. **Check Pod Status:** kubectl get pods for high-level overview (e.g., Pending, CrashLoopBackOff, ImagePullBackOff).
2. **Describe the Resource:** kubectl describe <resource_type> <resource_name> for detailed info, events, warnings.
3. **Review Logs:** kubectl logs <pod-name> or kubectl logs <pod-name> -c <container-name> for application errors. Use --previous for past instances.
4. **Check Events:** kubectl get events --sort-by='.lastTimestamp' for cluster-wide clues.
5. **Verify Configuration:** kubectl apply --dry-run=client -f <file>.yaml or kubectl get <resource> -o yaml to validate YAML.

- **Common Troubleshooting Scenarios & kubectl Commands:**
- **Pod Stuck in Pending State:**
- **Cause:** Cannot be scheduled due to insufficient resources, node selectors, taints/tolerations, hostPort conflicts.
- **Commands:**
- kubectl describe pod <pod-name>: Check scheduler messages in "Events".
- kubectl top nodes / kubectl top pods: Check resource utilization.
- kubectl get nodes -o wide: Check node status/resources.
- kubectl describe node <node-name> | grep -i taints: Check for taints.
- Adjust resource requests/limits or add nodes.
- **CrashLoopBackOff or Unhealthy Pods:**
- **Cause:** Container repeatedly crashing due to app bugs, misconfiguration, OOMKilled, resource starvation, improper commands, liveness probe failures.
- **Commands:**
- kubectl logs <pod-name> / kubectl logs <pod-name> -c <container-name>: Identify app errors.
- kubectl logs <pod-name> --previous: View logs from previous instance.
- kubectl describe pod <pod-name>: Check "Last State" and events (e.g., OOMKilled).
- kubectl get pod <pod-name> -o yaml: Inspect pod configuration.
- kubectl exec -it <pod-name> -- /bin/sh: Get shell inside container.
- **ImagePullBackOff or ErrImagePull:**
- **Cause:** Cannot pull container image (incorrect name/tag, auth issues, network problems).
- **Commands:**
- kubectl describe pod <pod-name>: Details on failed pull.
- kubectl get secret: Check registry credentials.

- Manually pull image on host (docker pull <image>).
- Test network connectivity from busybox pod (kubectl run -it --rm busybox --image=busybox -- /bin/sh then curl, ping, nslookup).
- **Service Inaccessibility / DNS Issues:**
- **Cause:** Service misconfigured, endpoints unhealthy, DNS resolution failures, Network Policies blocking traffic.
- **Commands:**
- kubectl get svc <service-name>: Verify service existence/type.
- kubectl describe service <service-name>: Detailed info, endpoints, selectors.
- kubectl get endpoints <service-name>: Directly check endpoints.
- kubectl run -it --rm --image=busybox dns-test -- nslookup <service-name>: Test DNS resolution from within cluster.
- kubectl logs -n kube-system -l k8s-app=kube-dns: Check DNS service logs.
- kubectl get networkpolicy: Verify Network Policies.
- **kubectl Connectivity Issues:**
- **Cause:** kubectl unable to connect to API server (misconfigured kubeconfig, VPN, auth/authz, API server unreachability).
- **Commands:**
- kubectl version: Check client/server versions (look for "Server Version").
- kubectl config view: Inspect kubeconfig context.
- kubectl config get-contexts / kubectl config use-context <context-name>: Verify/switch contexts.
- Check ~/.kube/config and $KUBECONFIG.
- Ping API server host, check network/firewall.
- Verify VPN connection.
- Check cloud provider health status for API server/load balancer.
- **Deployment Failures (General):**

- **Cause:** Misconfigured application, resource constraints, scheduling, networking, invalid configurations.
- **Commands:** Follow general troubleshooting steps. Pay attention to RESTARTS and READY columns in kubectl get pods.
- kubectl describe deployments <deployment-name>: Provides overall deployment details.