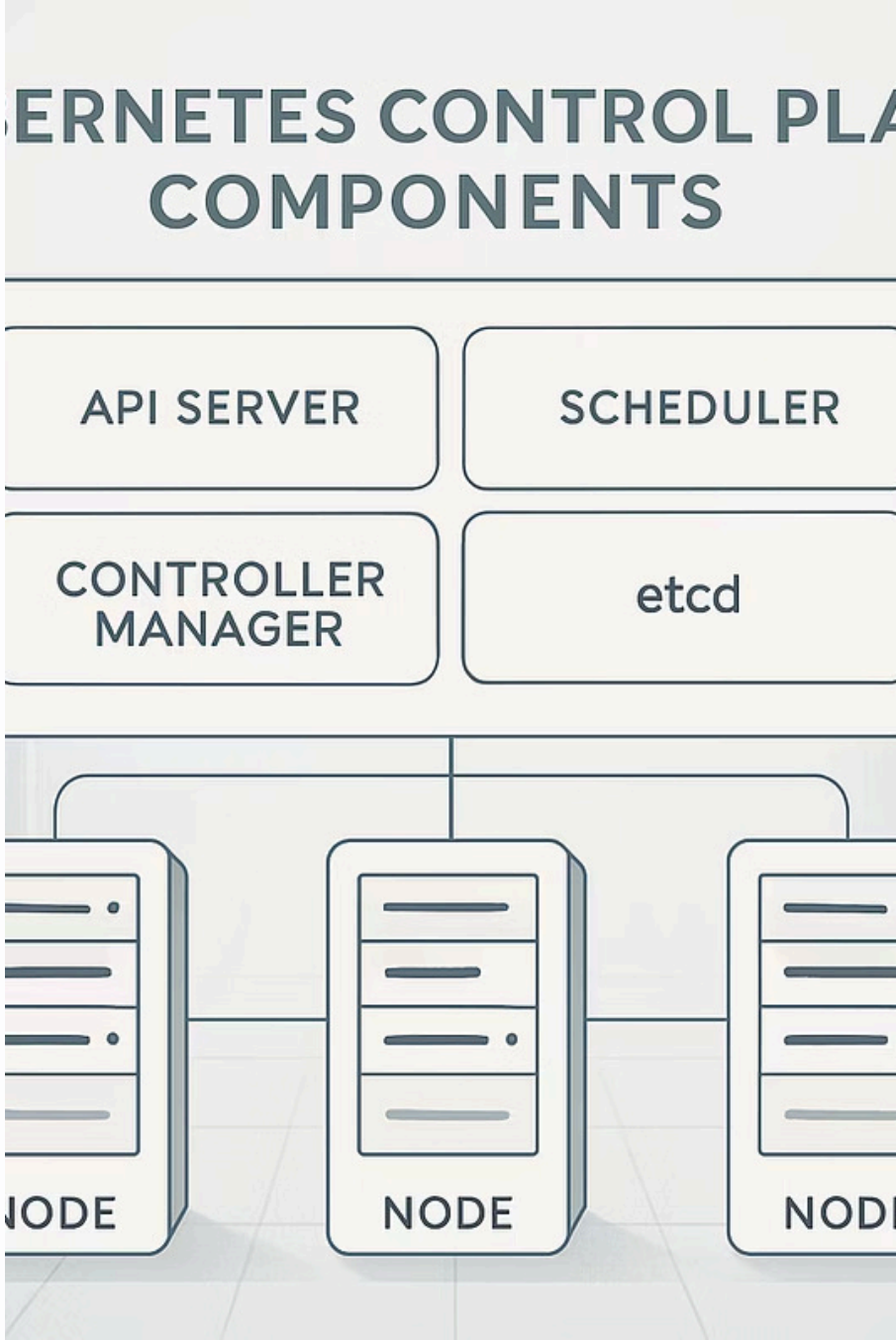
Kubernetes Cluster Maintenance and Monitoring

A technical guide for Kubernetes administrators on maintaining cluster health, performing upgrades, and implementing monitoring solutions.



Agenda

Cluster Maintenance

- OS Upgrades
- Kubernetes Version Upgrades
- Static Pods
- ETCD Backup & Restore
- Managing Cron Jobs

Logging & Monitoring

- Monitoring Applications
- Monitoring Cluster Components
- Log Analysis
- Prometheus Architecture
- Grafana Dashboards

Cluster Maintenance Overview

Cluster maintenance encompasses crucial operational tasks to ensure the reliability, security, and performance of your Kubernetes environment.

1

Routine Maintenance

OS updates, kernel patches, security fixes, and infrastructure changes

2

Version Upgrades

Kubernetes components, API versions, and feature enablement

3

Data Management

ETCD backups, restore procedures, and state management

4

Automation

Scheduled tasks, health checks, and self-healing mechanisms

Understanding OS Upgrades

Kubernetes clusters rely on stable, secure operating systems. OS upgrades must be performed carefully to maintain cluster integrity.

Node maintenance requires special handling to ensure workloads remain available during the process.

Considerations:

- Pod Disruption Budgets (PDBs)
- Node draining procedures
- Workload scheduling
- Rolling update patterns

Node Drain Process

Cordon Node

Mark node as unschedulable to prevent new pods:

```
kubectl cordon node-01
```

Drain Node

Safely evict all pods:

```
kubectl drain node-01 --ignore-daemonsets --delete-emptydir-data
```

Perform OS Updates

Apply operating system patches and updates

Uncordon Node

Mark node as schedulable again:

```
kubectl uncordon node-01
```

Pod Disruption Budgets

PDBs define how many replicas of a service must remain available during voluntary disruptions like node drains.

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: app-pdb
spec:
  minAvailable: 2    # or use maxUnavailable: 1
  selector:
    matchLabels:
      app: critical-service
```

During a node drain, Kubernetes respects PDBs and will block evictions that would violate the budget.

Upgrading a Kubernetes Cluster

Version Compatibility

Kubernetes follows n-2 version skew policy. Control plane can be up to two minor versions ahead of nodes.

Example: If nodes are at 1.24, control plane can go up to 1.26

Upgrade Sequence

1. Control plane components (apiserver, controller-manager, scheduler)

2. Node components (kubelet, kube-proxy)

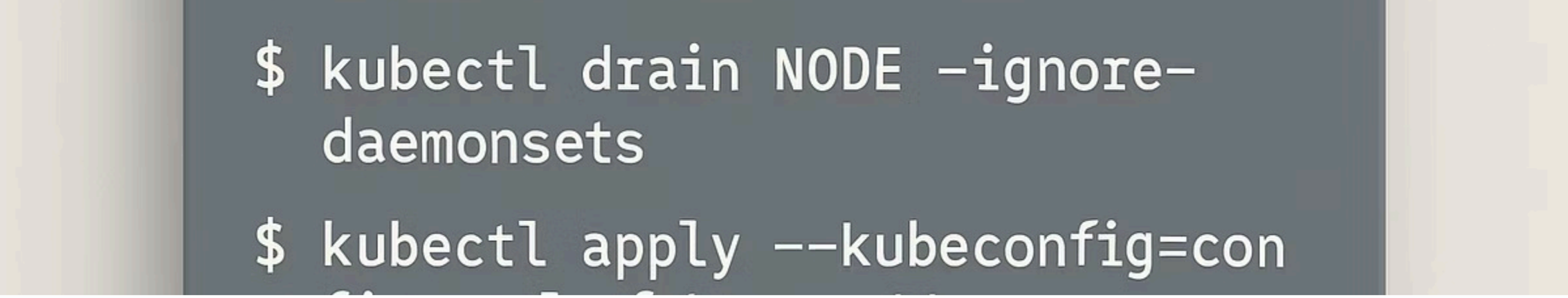
3. Add-ons and other components (CNI, CoreDNS)

Upgrade Methods

• For managed clusters: Use provider's upgrade mechanism

• For kubeadm clusters: `kubeadm upgrade`

• For custom installations: Follow component-specific upgrade paths



Kubeadm Upgrade Process

For clusters built with kubeadm, a typical control plane upgrade process:

```
# On the first control plane node
apt update
apt-cache madison kubeadm # Find available versions
apt-get install kubeadm=1.26.0-00
kubeadm version # Verify
kubeadm upgrade plan # Check upgrade paths
kubeadm upgrade apply v1.26.0 # Upgrade control plane

# Update kubelet and kubectl
apt-get install kubelet=1.26.0-00 kubectl=1.26.0-00
systemctl daemon-reload
systemctl restart kubelet
```

Upgrading Worker Nodes

Prepare Node

Drain the node to safely evict workloads:

```
kubectl drain worker-01 --ignore-daemonsets
```

Upgrade kubelet

Install new kubelet and kubeadm:

```
apt-get install -y kubelet=1.26.0-00 kubeadm=1.26.0-00
```

Apply Node Configuration

```
kubeadm upgrade node
```

Restart Services

```
systemctl daemon-reload
systemctl restart kubelet
```

Return To Service

```
kubectl uncordon worker-01
```

Upgrade Considerations

Before Upgrading:

- Review release notes for breaking changes
- Ensure adequate capacity during rolling upgrades
- Check storage driver compatibility
- Back up etcd data
- Test upgrade in non-production environment
- Verify CNI plugin compatibility

Potential Issues:

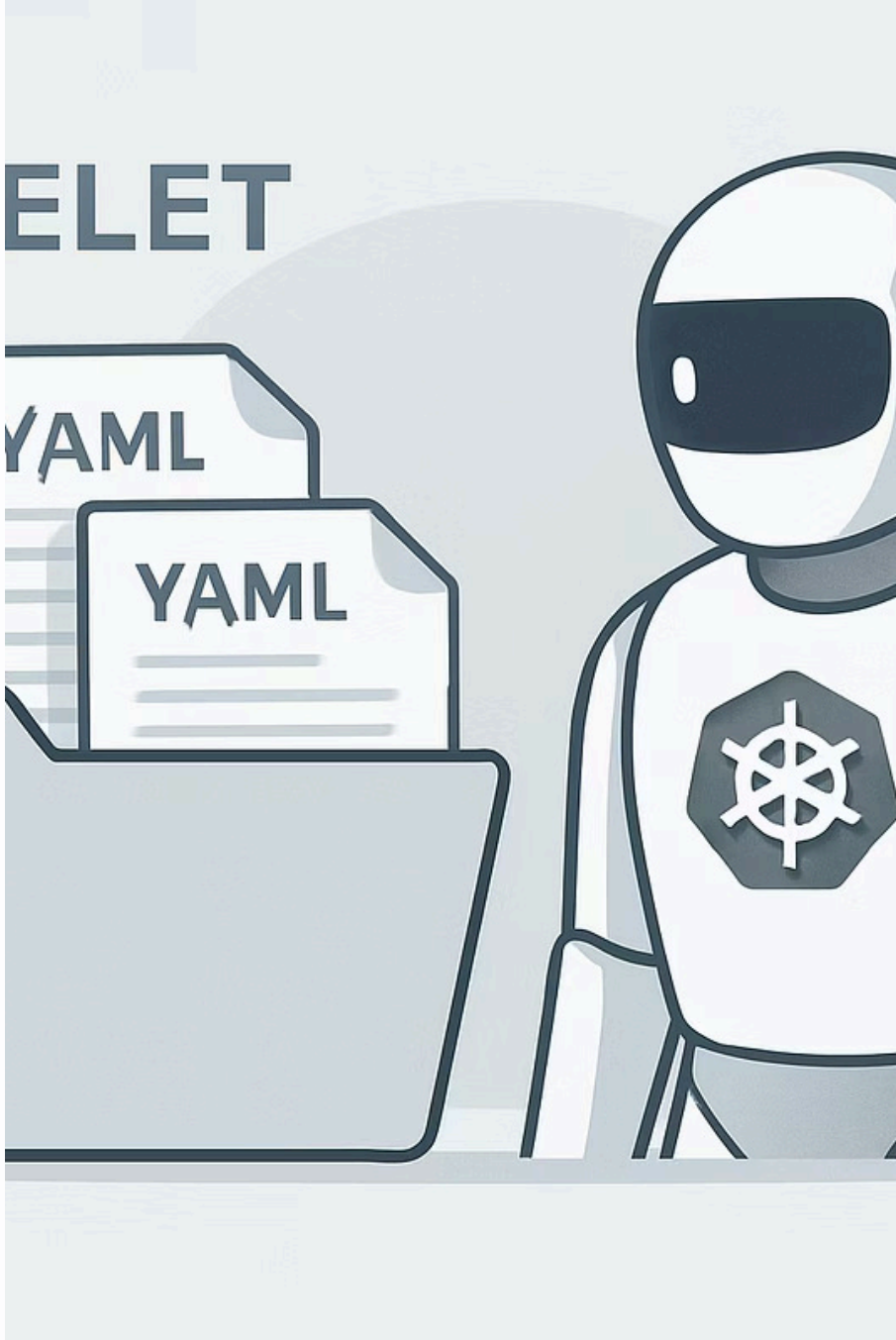
- API deprecations affecting manifests
- CRD version migrations
- Pod Security Policy replacements
- Feature gate changes
- Certificate rotation requirements
- Metrics format changes

Static Pods

Static pods are managed directly by the kubelet daemon on a specific node, not by the Kubernetes API server.

Key characteristics:

- Created from manifest files in a watched directory (typically `/etc/kubernetes/manifests/`)
- Cannot use ConfigMaps or Secrets (as they require API server)
- Named with the node hostname as a suffix
- Not controlled by Deployments, DaemonSets, or ReplicaSets
- Used for control plane components in kubeadm clusters



Creating Static Pods

Static pods are created by placing a pod manifest in the kubelet's configured static pod path.

Steps:

```
grep -r staticPodPath /etc/kubernetes/kubelet.conf
# or
ps -ef | grep kubelet | grep "\-\-pod-manifest-path"
```

```
vi /etc/kubernetes/manifests/static-web.yaml
```

1. Find the static pod path in kubelet configuration:
2. Create a pod manifest in that directory:
3. Kubelet automatically creates the pod

```
apiVersion: v1
kind: Pod
metadata:
 name: static-web
 namespace: default
spec:
 containers:
 - name: web
 image: nginx
 ports:
 - containerPort: 80
 resources:
 limits:
 memory: "128Mi"
 cpu: "500m"
```

Control Plane as Static Pods

API Server

Validates and processes API requests, manages REST operations

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

etcd

Distributed key-value store for all cluster data

`/etc/kubernetes/manifests/etcd.yaml`

Controller Manager

Runs controller processes (node, replication, endpoints, etc.)

`/etc/kubernetes/manifests/kube-controller-manager.yaml`

Scheduler

Assigns pods to nodes based on resource requirements

/etc/kubernetes/manifests/kube-scheduler.yaml

Managing Static Pods

To create or modify:

Edit or add manifest files in the static pod directory:

```
vi /etc/kubernetes/manifests/static-web.yaml
```

The kubelet will automatically apply changes.

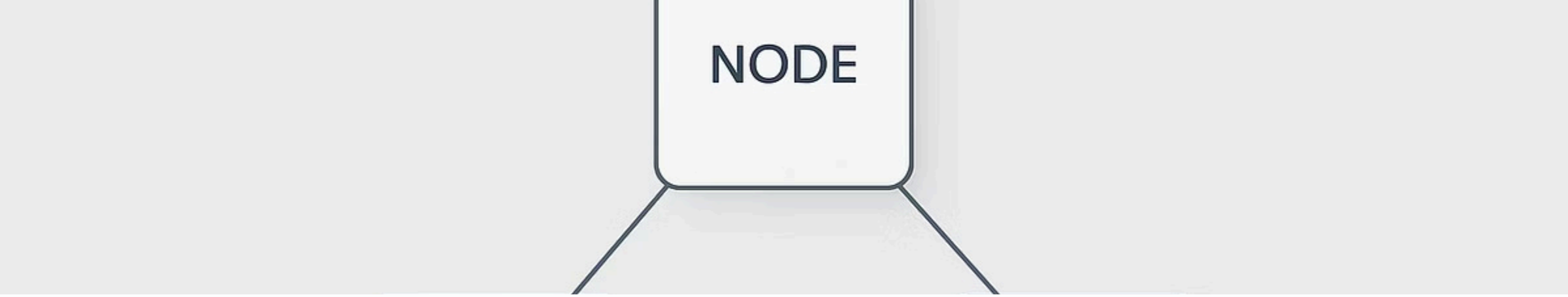
To delete:

Remove the manifest file from the directory:

```
rm /etc/kubernetes/manifests/static-web.yaml
```

Important notes:

- Static pods appear in `kubectl get pods` but are read-only to the API server
- You cannot use `kubectl delete pod` to remove them
- Static pods are recreated automatically if deleted
- They run with node-level privileges
- Changes to manifests trigger automatic restarts



ETCD Overview

etcd is a distributed, reliable key-value store used as Kubernetes' primary datastore for all cluster data.

Key Features

- Strongly consistent, distributed key-value store
- Uses Raft consensus algorithm
- Stores all API objects and cluster state
- Supports watch operations for changes

Configuration

- Can run as a static pod or systemd service
- Requires TLS certificates for secure communication
- Data directory contains all persistent state
- Typical port: 2379 for client communication

ETCD Backup

Regular backups of etcd are critical for disaster recovery. Kubernetes stores all its data in etcd, including all API objects, state, and metadata.

Backup with etcdctl:

```
ETCDCTL_API=3 etcdctl \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  snapshot save /backup/etcd-snapshot-$(date +%Y-%m-%d_%H-%M-%S).db
```

This creates a point-in-time snapshot of the etcd database that can be used for restoration.

ETCD Restore

Stop kube-apiserver

On kubeadm clusters:

```
mv /etc/kubernetes/manifests/kube-apiserver.yaml /tmp/
```

Restore the snapshot

```
ETCDCTL_API=3 etcdctl snapshot restore /backup/etcd-snapshot.db \
  --data-dir=/var/lib/etcd-restore \
  --name=master \
  --initial-cluster=master=https://127.0.0.1:2380 \
  --initial-cluster-token=etcd-cluster-1 \
  --initial-advertise-peer-urls=https://127.0.0.1:2380
```

Update etcd config

Update the etcd.yaml manifest to use the restored data directory

Restart services

Restore the API server manifest and verify cluster health

ETCD Backup Best Practices

Backup Frequency

- Production clusters: At least daily backups
- Critical systems: Consider hourly backups
- Before major changes or upgrades

Retention Policy

- Keep multiple historical backups
- Implement rotation (daily, weekly, monthly)
- Consider legal and compliance requirements

Storage Considerations

- Store backups on separate physical systems
- Use offsite or cloud storage
- Encrypt backup files
- Verify backup integrity regularly

Automation

- Implement automated backup jobs
- Monitor backup success/failure
- Regularly test restoration process

Cron Jobs in Kubernetes

CronJobs create Jobs on a schedule, allowing automated, recurring tasks in your cluster.

Common use cases include:

- Database backups
- Report generation
- Email sending
- Clean-up operations
- Metrics collection
- Scheduled API calls or data synchronization

CronJob Specification

```
apiVersion: batch/v1
kind: CronJob
metadata:
 name: backup-job
spec:
 schedule: "0 1 * * *" # Daily at 1:00 AM
 timeZone: "Etc/UTC" # Optional: timezone for schedule
 concurrencyPolicy: Forbid # Allow/Forbid/Replace
 successfulJobsHistoryLimit: 3
 failedJobsHistoryLimit: 1
 startingDeadlineSeconds: 600
 jobTemplate:
 spec:
 template:
 spec:
 containers:
 - name: backup
 image: backup-image:latest
 command: ["backup-script.sh"]
 restartPolicy: OnFailure
```

CronJob Scheduling Syntax

CronJobs use standard cron syntax with five fields:

```
* * * * *
| | | | |
| | | | └──────── Day of week (0-6 or SUN-SAT)
| | | └────────── Month (1-12 or JAN-DEC)
| | └──────────── Day of month (1-31)
| └────────────── Hour (0-23)
└──────────────── Minute (0-59)
```

Common Patterns:

`0 * * * *`	Every hour at minute 0
`0 0 * * *`	Daily at midnight
`0 0 * * 0`	Weekly on Sunday at midnight
`0 0 1 * *`	Monthly on the 1st at midnight
`*/5 * * * *`	Every 5 minutes
`0 9-17 * * 1-5`	Hourly during business hours

CronJob Management

Concurrency Policies

- **Allow**: Multiple jobs can run concurrently
- **Forbid**: Skip new job if previous still running
- **Replace**: Cancel running job and start new one

Job History

`successfulJobsHistoryLimit`: Number of completed jobs to keep

`failedJobsHistoryLimit`: Number of failed jobs to keep

Set these to prevent unbounded growth of job objects

Starting Deadline

`startingDeadlineSeconds`: Maximum time to start job after scheduled time

If job misses this window, it's counted as a failed execution

Debugging CronJobs

Common Issues:

- Incorrect cron syntax
- Container errors or misconfigurations
- Resource constraints
- Time zone inconsistencies
- Job taking too long, causing concurrency issues

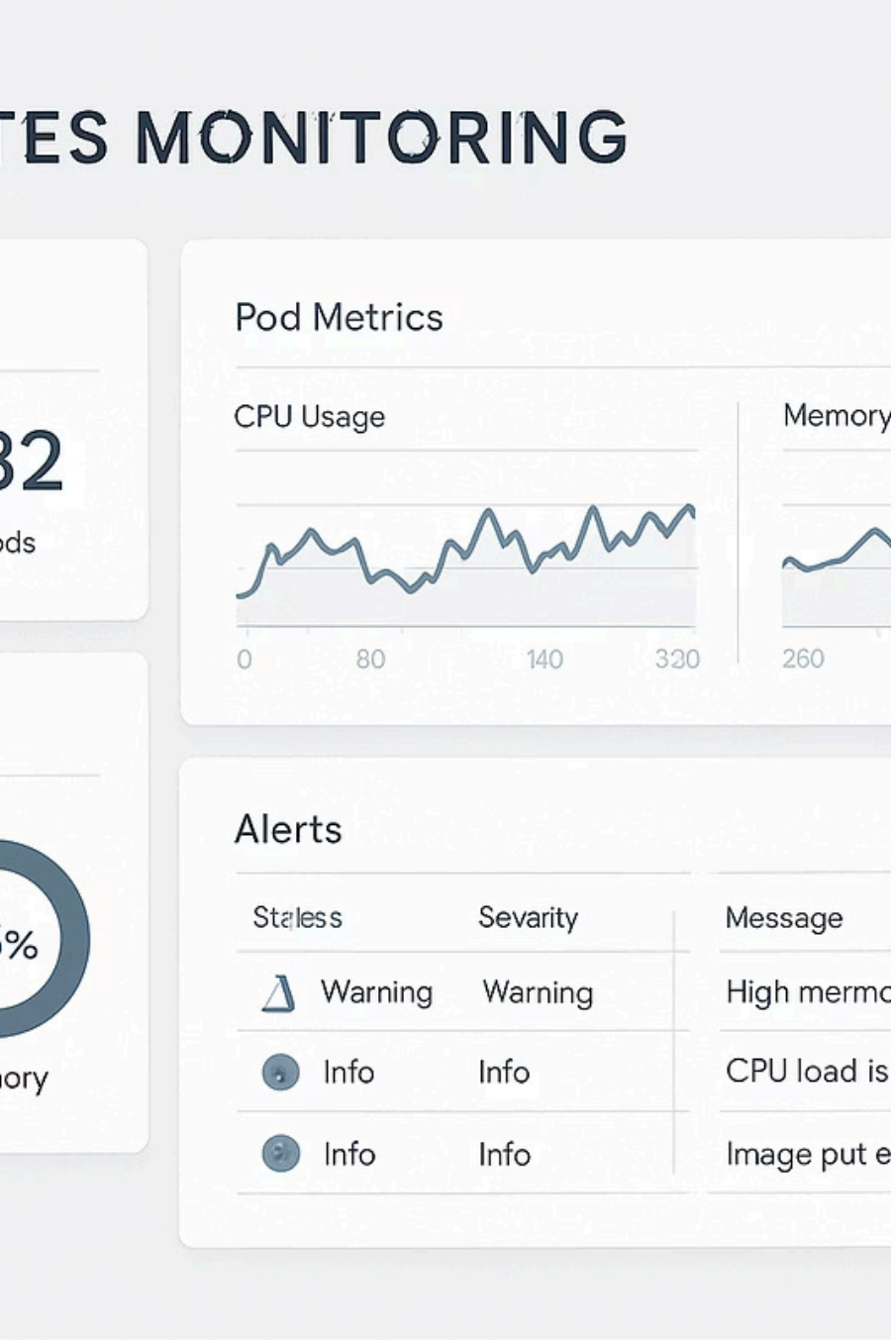
Troubleshooting Commands:

```
# Check CronJob configuration
kubectl get cronjob backup-job -o yaml

# Check status of recent jobs
kubectl get jobs --selector=job-name=backup-job

# View logs from the most recent job
kubectl logs job/backup-job-1234567890

# Check events for issues
kubectl get events --field-selector \
 involvedObject.name=backup-job
```



Monitoring in Kubernetes

Effective monitoring is essential for maintaining a healthy Kubernetes cluster and ensuring optimal application performance.

Monitoring Requirements for Kubernetes:

- Cluster-level component metrics (control plane, nodes)
- Container and pod resource utilization
- Application-level metrics and health indicators
- Alert generation for anomalies and threshold violations
- Historical data for trend analysis and capacity planning

Types of Kubernetes Metrics

1	2	3
Resource Metrics • CPU usage • Memory consumption • Disk I/O • Network traffic Exposed by kubelet's cAdvisor and Metrics Server	Kubernetes State Metrics • API object counts • Resource status • Deployment/StatefulSet/DaemonSet state Exposed by kube-state-metrics	Application Metrics • Request rates • Error rates • Response times • Business KPIs Exposed by applications using Prometheus client libraries

Monitoring Control Plane Components

Components to Monitor:

- kube-apiserver
- etcd
- kube-scheduler
- kube-controller-manager
- cloud-controller-manager (if used)

Key Metrics:

- API request rate and latency
- etcd operations and latency
- Leader election status
- Work queue depth
- Resource consumption
- Error rates

Most control plane components expose metrics on `/metrics` endpoint in Prometheus format.

Monitoring Worker Nodes

Node Status

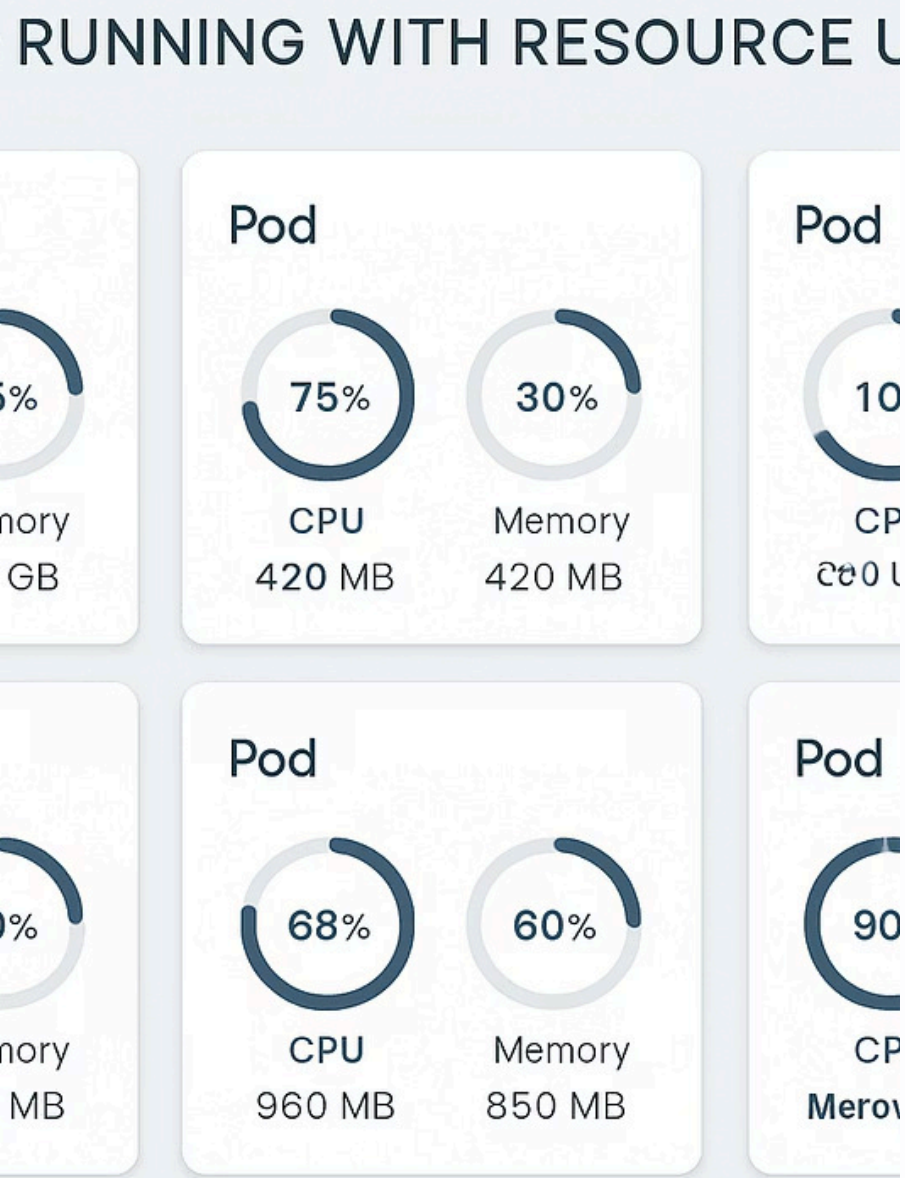
- Ready condition
- Node capacity vs allocatable resources
- Pressure conditions (memory, disk, PID)
- Kubelet status

Resource Usage

- CPU utilization
- Memory consumption
- Disk space/IO
- Network throughput

System Metrics

- Load average
- File descriptors
- System services
- Kernel parameters



Monitoring Application Pods

Key Pod-level Metrics:

- CPU and memory usage vs requests/limits
- Pod status and phase (Running, Pending, Failed)
- Restart count
- Container state
- Network I/O
- Volume usage

Additional application-specific metrics should be exposed by instrumenting your applications with Prometheus client libraries.

Kubernetes Metrics Server

Metrics Server collects resource metrics from kubelets and exposes them through the Kubernetes Metrics API for use with the Horizontal Pod Autoscaler and Vertical Pod Autoscaler.

Features:

- Lightweight, memory-only
- No historical data storage
- Cluster-wide aggregation
- Required for `kubectl top` command

Deployment:

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

Usage:

```
# View node resource usage
kubectl top nodes

# View pod resource usage
kubectl top pods -n kube-system
```

Reading Kubernetes Logs

Kubernetes provides a unified logging interface through the `kubectl logs` command, enabling access to container logs across the cluster.

Basic Commands:

```
# Get logs from a pod
kubectl logs nginx-pod

# Get logs from specific container in pod
kubectl logs nginx-pod -c nginx

# Follow logs in real-time
kubectl logs -f deployment/my-app

# Show logs from terminated containers
kubectl logs --previous nginx-pod
```

Advanced Options:

```
# Get logs with timestamps
kubectl logs --timestamps=true nginx-pod

# Get most recent logs (tail)
kubectl logs --tail=100 nginx-pod

# Get logs since specific time
kubectl logs --since=1h nginx-pod

# Get logs matching pattern
kubectl logs nginx-pod | grep ERROR
```

Control Plane Component Logs

Static Pod Logs

For kubeadm clusters where control plane runs as static pods:

```
kubectl logs -n kube-system kube-apiserver-master
```

```
kubectl logs -n kube-system kube-controller-manager-master
```

```
kubectl logs -n kube-system kube-scheduler-master
```

```
kubectl logs -n kube-system etcd-master
```

Systemd Service Logs

For components running as systemd services:

```
journalctl -u kube-apiserver
```

```
journalctl -u kube-controller-manager
```

```
journalctl -u kube-scheduler
```

```
journalctl -u etcd
```

Container Runtime Logs

```
journalctl -u containerd
```

```
journalctl -u docker
```

```
crictl logs [container-id]
```

Node Component Logs

Kubelet Logs:

```
journalctl -u kubelet
```

Kube-proxy Logs:

```
kubectl logs -n kube-system kube-proxy-xxxxx
```

Node Problem Detector (if installed):

```
kubectl logs -n kube-system node-problem-detector-xxxxx
```

Log verbosity can be adjusted in component configurations. Higher verbosity (0-9) provides more detailed logs but increases log volume.

Log Management Solutions

EFK Stack

Elasticsearch, Fluentd, and Kibana provide a comprehensive log collection, storage, and visualization solution.

PLG Stack

Promtail, Loki, and Grafana offer a lightweight, Prometheus-inspired logging stack with tight Grafana integration.

Managed Solutions

Services like Datadog, New Relic, and Splunk provide integrated observability platforms with log management capabilities.

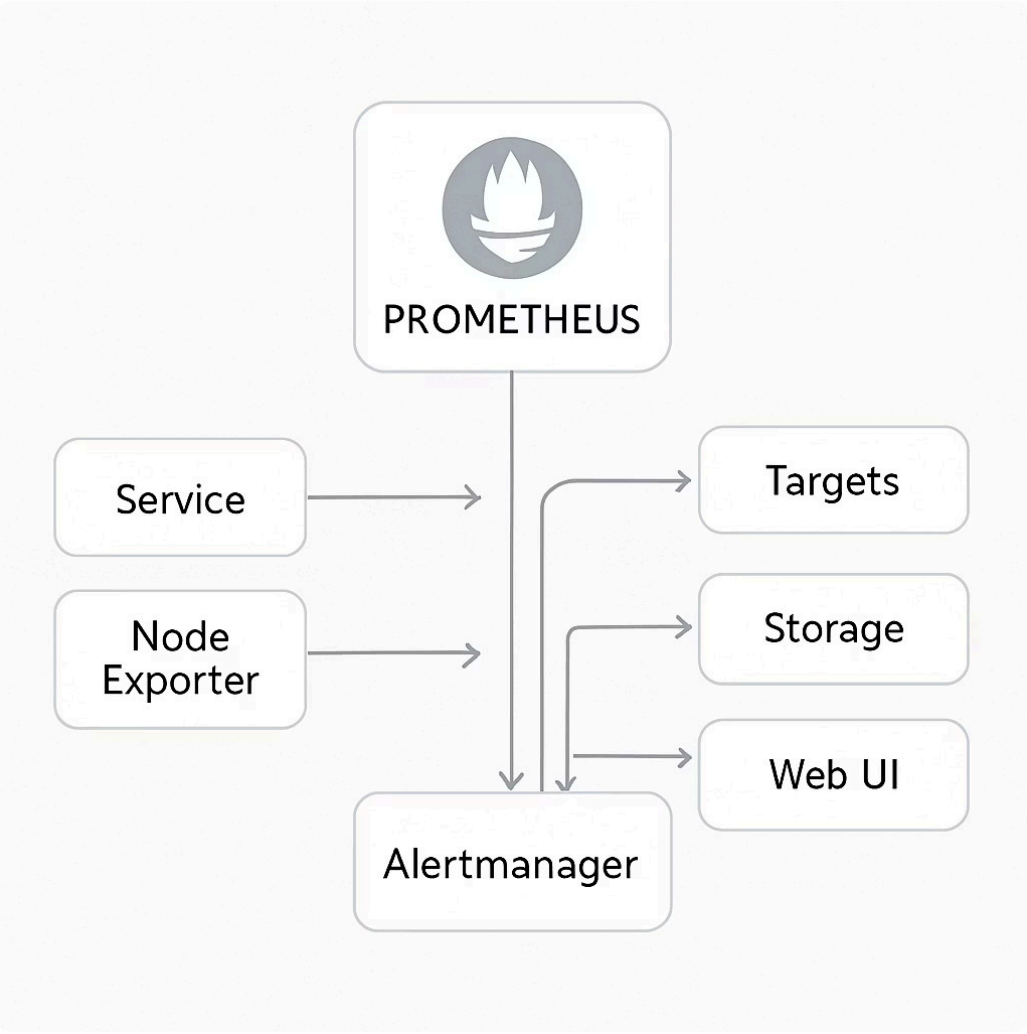
These solutions offer centralized log aggregation, indexing, search capabilities, and retention policies that overcome the limitations of `kubectl logs`.

Introduction to Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit designed for reliability and scalability.

Key Features:

- Multi-dimensional data model with time series data identified by metric name and key-value pairs
- PromQL, a flexible query language
- No reliance on distributed storage; single server nodes are autonomous
- Time series collection via a pull model over HTTP
- Pushing time series supported via an intermediary gateway



Prometheus Architecture

Data Collection

Prometheus server scrapes metrics from instrumented jobs via HTTP endpoints exposing metrics in the Prometheus format.

Storage

Collected metrics are stored locally in a custom time-series database that efficiently compresses and indexes data.

PromQL

A powerful query language that allows selection and aggregation of time series data to generate ad-hoc graphs and alerts.

Alerting

Alertmanager handles alerts sent by Prometheus server, including deduplication, grouping, and routing to the correct receiver.

Deploying Prometheus in Kubernetes

Prometheus Operator and kube-prometheus provide a complete monitoring solution for Kubernetes clusters.

Components:

- Prometheus servers
- Alertmanager
- node-exporter (for node metrics)
- kube-state-metrics
- Prometheus Adapter for metrics API
- Grafana for visualization

Installation with Helm:

```
# Add Prometheus Helm repository
helm repo add prometheus-community \
 https://prometheus-community.github.io/helm-charts
helm repo update

# Install kube-prometheus-stack
helm install prometheus prometheus-community/kube-prometheus-stack \
 --namespace monitoring \
 --create-namespace
```

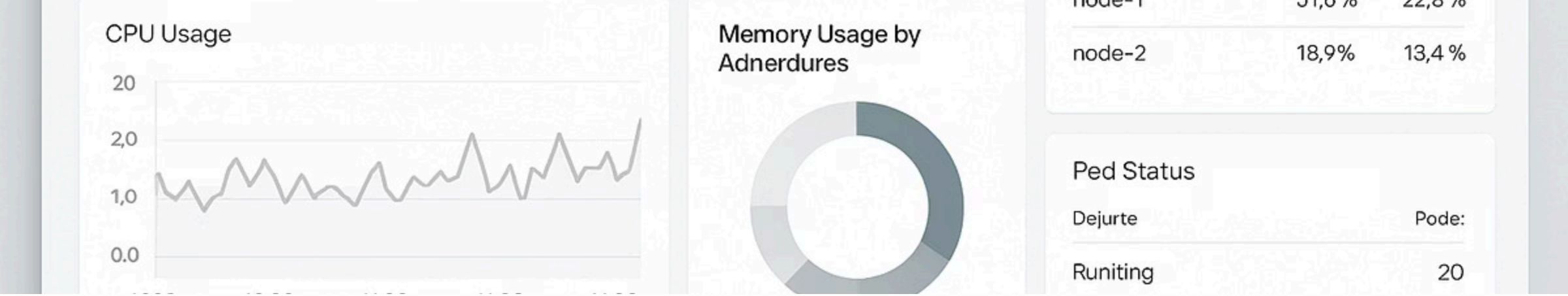
Prometheus Configuration

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
 name: prometheus
 namespace: monitoring
spec:
 serviceAccountName: prometheus
 serviceMonitorSelector:
 matchLabels:
 team: frontend
 ruleSelector:
 matchLabels:
 role: alert-rules
alerting:
alertmanagers:
- namespace: monitoring
name: alertmanager
port: web
resources:
requests:
memory: 400Mi
retention: 15d
storage:
volumeClaimTemplate:
spec:
storageClassName: standard
resources:
requests:
storage: 50Gi
```

ServiceMonitor Configuration

ServiceMonitors define how Prometheus discovers and scrapes services in Kubernetes.

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
 name: example-app
 namespace: monitoring
 labels:
 team: frontend
spec:
 selector:
 matchLabels:
 app: example-app
 namespaceSelector:
matchNames:
- default
endpoints:
- port: web
interval: 30s
path: /metrics
scheme: http
tlsConfig:
insecureSkipVerify: true
- port: metrics
interval: 15s
metricRelabelings:
- sourceLabels: [__name__]
regex: 'example_metric_total'
action: keep
```



Grafana Dashboards

Grafana provides visualization and analytics for Prometheus metrics data.

Key Features:

- Rich visualization options (graphs, tables, heatmaps)
- Dashboard templating with variables
- Alert definitions and notifications
- Authentication and role-based access control
- Annotation support for marking events

The kube-prometheus-stack includes pre-configured dashboards for Kubernetes components.

Key Takeaways

1 Cluster Maintenance

Proper node draining, following upgrade processes, and regular ETCD backups are essential for maintaining cluster health and ensuring zero-downtime maintenance.

2 Static Pods and CronJobs

Understanding static pods is critical for managing control plane components. CronJobs provide scheduled task automation for operational needs.

3 Monitoring & Logging

A comprehensive monitoring strategy includes resource metrics, component health, and application performance. Prometheus and Grafana provide powerful tools for observability.

4 Automation

Automating routine maintenance tasks, backups, and monitoring ensures reliability and reduces operational overhead for Kubernetes administrators.