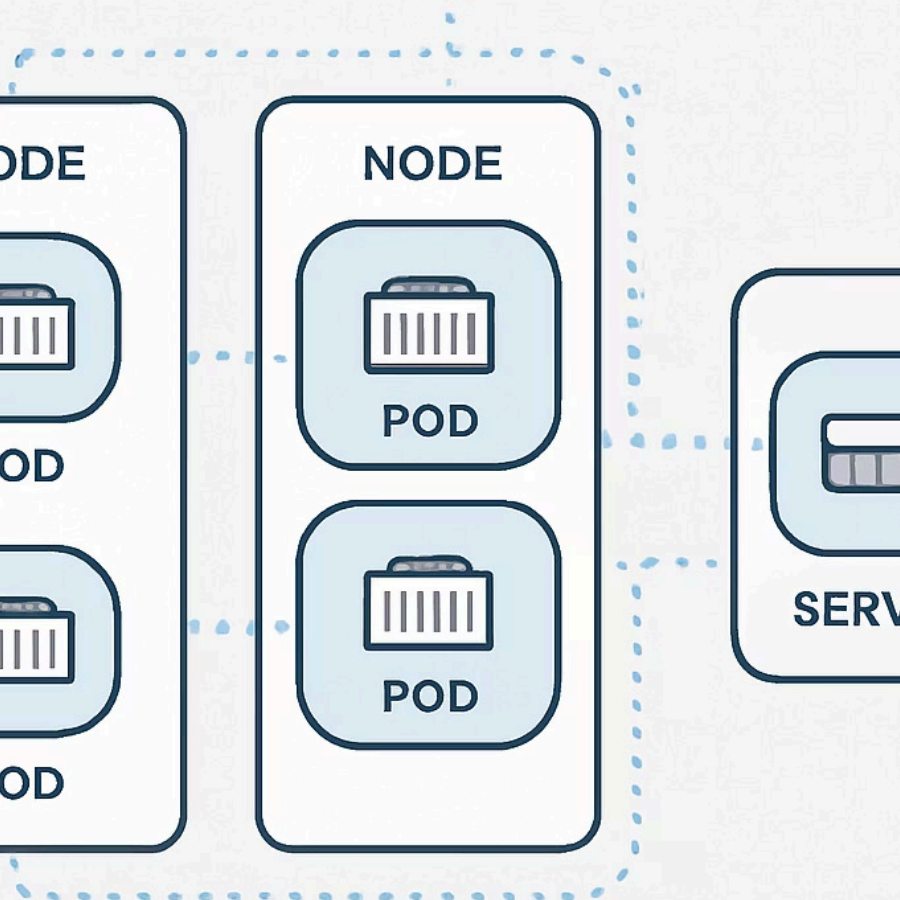




# KUBERNETES



## Mastering Kubernetes: From Pods to Production

Welcome to your comprehensive journey through Kubernetes resource creation and scheduling. We'll transform complex container orchestration concepts into practical, real-world knowledge that you can immediately apply in your projects.

# Chapter 1

## Understanding Kubernetes Resources

Building blocks of container orchestration

# What is a Pod? Think of it as an Apartment

## Real-World Analogy

A Pod is like an apartment building where containers live together. Just as roommates in an apartment share utilities, Wi-Fi, and storage space, containers in a Pod share network interfaces, storage volumes, and computing resources.

Every Pod gets its own IP address, like each apartment has its own address. Containers inside communicate using localhost, just like roommates talking face-to-face.



# Creating Your First Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: my-web-app
  labels:
    app: frontend
    tier: web
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
      ports:
        - containerPort: 80
```

This Pod is like booking a studio apartment for your web application. The nginx container lives alone but could have roommates (other containers) if needed. The labels act like apartment tags - "frontend apartment in the web tier."



# Labels and Selectors: The Kubernetes Address System

## Labels Are Like Tags

Think of labels as sticky notes you put on moving boxes. You might label boxes as "kitchen," "bedroom," or "fragile." Similarly, you label Pods with "app=frontend," "environment=production," or "version=v2.1."

## Selectors Are Like Search Filters

Selectors work like filters on a shopping website. Want all "red shirts in size medium"? That's a selector. In Kubernetes, you might select all Pods with "app=frontend AND environment=production."

# Practical Example: E-commerce Labels

```
# Frontend Pods
```

```
labels:
```

```
  app: ecommerce
```

```
  tier: frontend
```

```
  version: v2.1
```

```
# Database Pods
```

```
labels:
```

```
  app: ecommerce
```

```
  tier: database
```

```
  version: v1.5
```

```
# Payment Service Pods
```

```
labels:
```

```
  app: ecommerce
```

```
  tier: payment
```

```
  version: v3.0
```

Imagine managing an online store with hundreds of services. Labels help you instantly find "all payment services running version 3.0" or "all frontend components that need updating." It's like having a perfectly organized warehouse where everything has a clear, searchable label.



# Replication Controller: The Original Guardian

Think of a Security Guard

A Replication Controller is like a security guard who ensures exactly 3 people are always in the building. If someone leaves, the guard immediately calls a replacement. If too many people show up, the guard politely asks extras to leave.

It continuously watches and maintains the desired number of Pod replicas, but it's the older, less flexible approach.



# Replica Set: The Modern Manager

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend-replicas
spec:
  replicas: 3
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: web-server
          image: nginx:1.20
```

A ReplicaSet is like an evolved security system with smart cards. It can recognize different types of employees (using advanced selectors) and has more sophisticated rules. It's the modern replacement for Replication Controllers, offering better flexibility and label matching capabilities.



# Replica Set vs Replication Controller

## Replication Controller

- Basic equality-based selectors
- Simple label matching (app=frontend)
- Legacy approach
- Limited flexibility

## Replica Set

- Advanced set-based selectors
- Complex matching (app in [frontend, backend])
- Modern standard
- Used by Deployments

Think of upgrading from a basic key lock (Replication Controller) to a smart lock with multiple access methods (ReplicaSet). Both secure your house, but the smart lock offers more features and flexibility.

# Chapter 2

## Kubernetes Scheduling

Placing workloads where they belong



# How Kubernetes Scheduling Works

Kubernetes scheduling is like an air traffic control system. The scheduler (air traffic controller) decides which runway (Node) each plane (Pod) should land on based on available space, weather conditions (resource availability), and special requirements.

By default, the scheduler automatically picks the best Node for your Pod. But sometimes, you need to take manual control, just like when a pilot requests a specific runway due to special cargo or weather conditions.

# Manual Scheduling: Taking Control

## Why Manual Scheduling?

- Compliance requirements (data must stay in specific regions)
- Hardware dependencies (GPU-intensive workloads)
- Network proximity (reduce latency)
- Testing and debugging specific nodes

```
apiVersion: v1
kind: Pod
metadata:
  name: manual-pod
spec:
  nodeName: worker-node-2
  containers:
  - name: app
    image: my-app:latest
```



It's like being a pilot who specifically requests to land at Terminal B because you're carrying medical supplies that need immediate ground transport available only at that terminal.

# Node Selector: The Preferred Method

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-workload
spec:
  nodeSelector:
    hardware: gpu-enabled
    zone: us-west-1
  containers:
  - name: ml-training
    image: tensorflow:gpu
```

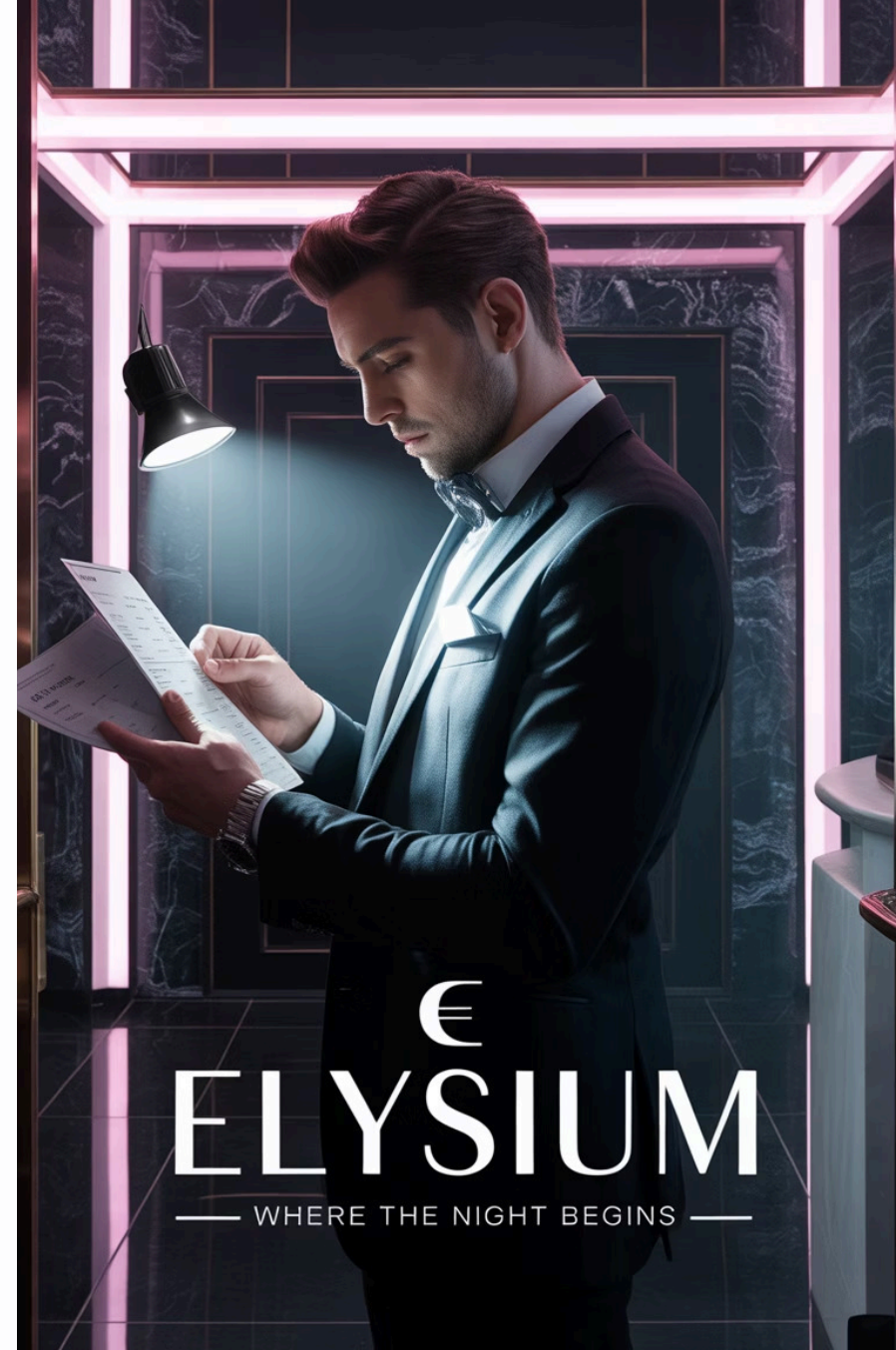
NodeSelector is like booking a hotel room with specific amenities. You don't care about the exact room number, but you need a room with a gym, pool, and business center. Kubernetes finds any node with labels "hardware=gpu-enabled" and "zone=us-west-1."

First, label your nodes: `kubectl label nodes worker-1 hardware=gpu-enabled`

# Taints and Tolerations: The Bouncer System

Imagine an exclusive nightclub where the bouncer (taint) only allows people with special passes (tolerations). Nodes can have taints that repel Pods, unless those Pods have matching tolerations.

This system ensures that only specific workloads run on dedicated nodes, like keeping production workloads separate from development ones, or reserving high-performance nodes for critical applications.





# Applying Taints and Tolerations

## Step 1: Taint the Node

```
kubectl taint nodes node1 \
dedicated=gpu:NoSchedule
```

This is like putting a "VIP Only" sign on a node. Regular Pods can't schedule here anymore.

## Step 2: Add Toleration to Pod

```
tolerations:
- key: "dedicated"
  operator: "Equal"
  value: "gpu"
  effect: "NoSchedule"
```

### NoSchedule

Prevent new Pods from scheduling (but don't evict existing ones)

### PreferNoSchedule

Try to avoid scheduling here, but allow if necessary

### NoExecute

Evict existing Pods and prevent new ones

# Real-World Taint Example: Database Isolation

```
# Taint database nodes
kubectl taint nodes db-node-1 db-node-2 \
  workload=database:NoSchedule
```

```
# Database Pod with toleration
```

```
apiVersion: v1
kind: Pod
metadata:
  name: mysql-primary
spec:
  tolerations:
    - key: "workload"
      operator: "Equal"
      value: "database"
      effect: "NoSchedule"
  containers:
    - name: mysql
      image: mysql:8.0
```

Think of this like a hospital where certain floors are reserved for specific departments. The cardiac unit has restricted access - only cardiac staff (Pods with cardiac tolerations) can work there, ensuring resources aren't consumed by unrelated activities.



# DaemonSets: The Night Security Patrol

A DaemonSet ensures exactly one Pod runs on every node in your cluster, like having a security guard patrol every floor of a building. Whether you have 3 floors or 300 floors, each gets exactly one guard.

Common uses include log collection agents, monitoring agents, network proxies, and storage daemons. As nodes join or leave the cluster, DaemonSets automatically adjust, ensuring comprehensive coverage.

# Creating a DaemonSet: Log Collection

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: log-collector
spec:
  selector:
    matchLabels:
      app: log-collector
  template:
    metadata:
      labels:
        app: log-collector
    spec:
      containers:
        - name: fluent-bit
          image: fluent/fluent-bit:latest
          volumeMounts:
            - name: varlog
              mountPath: /var/log
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
```

This DaemonSet deploys a log collector to every node, like installing security cameras in every room. Each collector monitors local logs and forwards them to a central system. When you add new nodes, they automatically get their own log collector.

# Chapter 3

## Kubernetes Services

Connecting and exposing your applications

# Why Do We Need Services?

## The Pod Problem

Pods are like people at a conference - they get temporary name tags (IP addresses) that change when they leave and come back. If you want to find "Sarah from Marketing," you can't rely on her seat number because it changes.

Services solve this by providing a permanent "reception desk" that always knows how to reach the right Pods, regardless of their changing IP addresses.





# Service Types: Three Ways to Connect



## ClusterIP

Internal communication only. Like an intercom system that only works within the building.



## NodePort

External access through node ports. Like opening a specific door to the building that outsiders can use.



## LoadBalancer

Cloud provider integration. Like having a professional doorman service manage multiple entrances.



# ClusterIP Service: Internal Communication

```
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  type: ClusterIP # Default type
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

ClusterIP is like the phone system in a large office building. Employees can call each other using extensions (backend-service:80), but people outside the building can't reach these internal numbers. This service creates a stable internal endpoint that frontend applications can use to reach backend Pods.

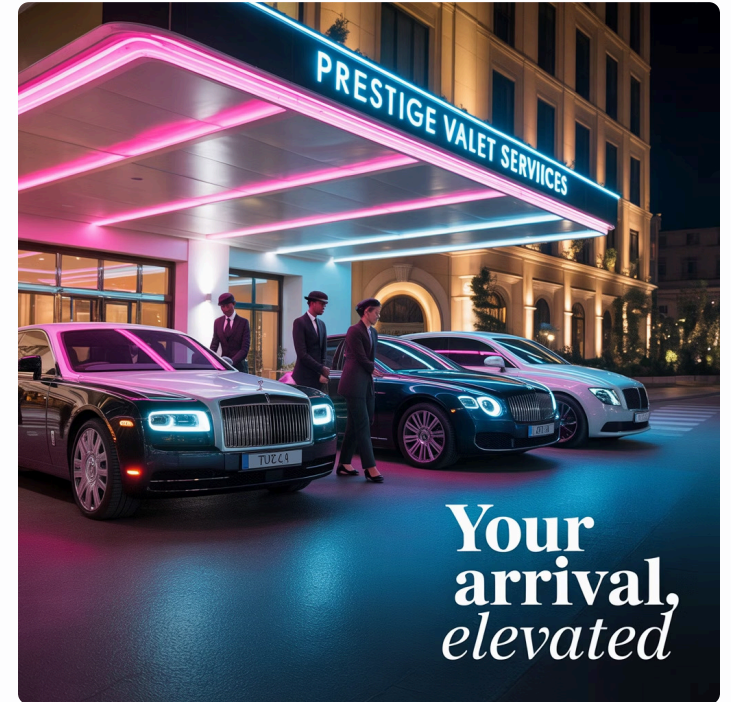
# NodePort Service: External Access

```
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  type: NodePort
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      nodePort: 30080 # External port on each node
```

NodePort is like opening the same door number (30080) on every building in your campus. Visitors can reach your service by going to any building's door 30080. Kubernetes automatically routes traffic from any node's port 30080 to the correct Pods, regardless of which node they're actually running on.

# LoadBalancer Service: Cloud Integration

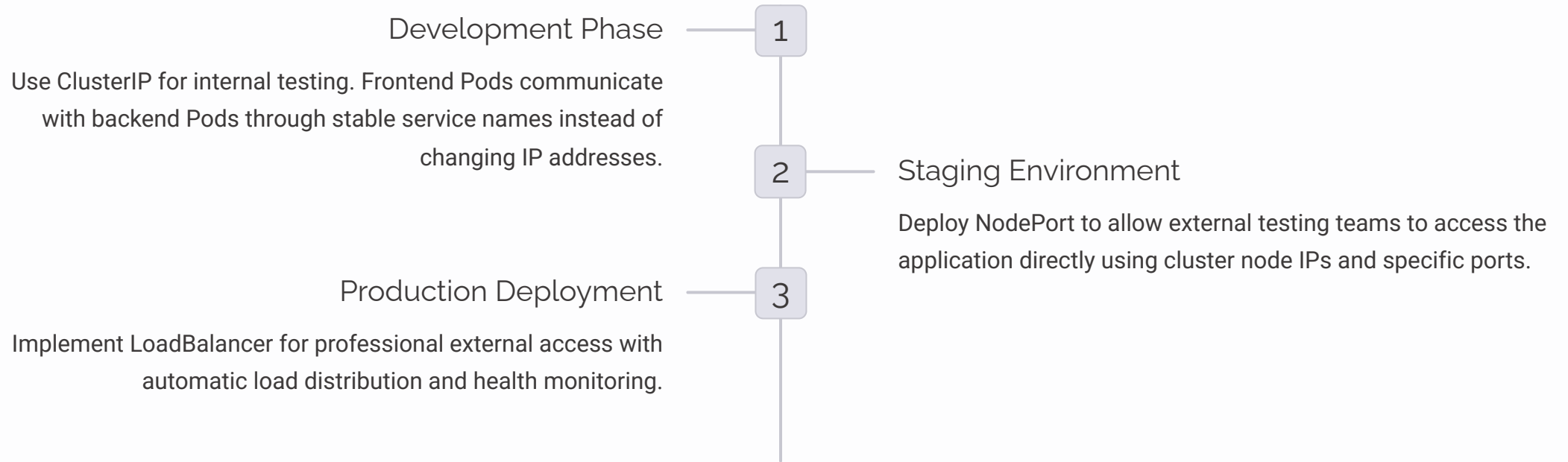
```
apiVersion: v1
kind: Service
metadata:
  name: public-web-service
spec:
  type: LoadBalancer
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```



LoadBalancer is like hiring a professional valet service. Instead of giving visitors confusing directions to multiple building entrances, you provide one elegant address. The cloud provider creates a load balancer that distributes traffic across all your nodes automatically.

Perfect for production applications that need a clean, professional entry point with automatic traffic distribution and health checking.

# Service Selection in Action



# Complete E-commerce Example

```
# Frontend (LoadBalancer for public access)
```

```
apiVersion: v1
```

```
kind: Service
```

```
metadata:
```

```
  name: ecommerce-frontend
```

```
spec:
```

```
  type: LoadBalancer
```

```
  selector:
```

```
    app: frontend
```

```
  ports:
```

```
  - port: 80
```

```
    targetPort: 3000
```

```
---
```

```
# Backend API (ClusterIP for internal use)
```

```
apiVersion: v1
```

```
kind: Service
```

```
metadata:
```

```
  name: api-service
```

```
spec:
```

```
  type: ClusterIP
```

```
  selector:
```

```
    app: backend-api
```


```
  ports:
```

```
  - port: 8080
```

```
    targetPort: 8080
```

This setup creates a complete e-commerce architecture: customers access the frontend through a clean LoadBalancer URL, while the frontend internally communicates with the backend API through the reliable ClusterIP service.





# Putting It All Together: Complete Application

## Pods

Run your application containers (frontend, backend, database)

## ReplicaSets

Ensure high availability with multiple Pod replicas

## DaemonSets

Deploy monitoring and logging to every node

## Services

Connect everything with stable networking

Like building a city: Pods are the buildings, ReplicaSets ensure neighborhoods stay populated, DaemonSets provide city-wide utilities, and Services create the road network that connects everything together.



# Your Kubernetes Journey

You've learned the essential building blocks of Kubernetes: Pods as your application homes, ReplicaSets as your availability guardians, scheduling as your resource optimization strategy, and Services as your networking foundation.

**Remember:** Every complex Kubernetes deployment starts with these fundamentals. Practice with simple examples, gradually building complexity. The concepts you've learned today - thinking of Pods as apartments, Services as reception desks, and DaemonSets as security patrols - will help you architect robust, scalable applications.