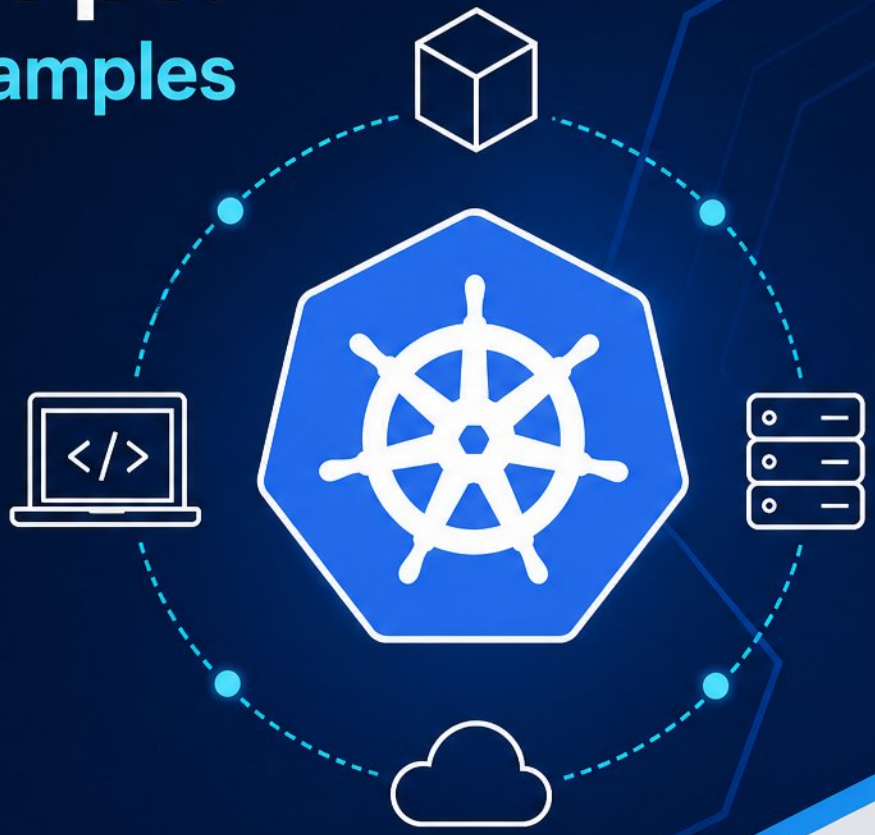
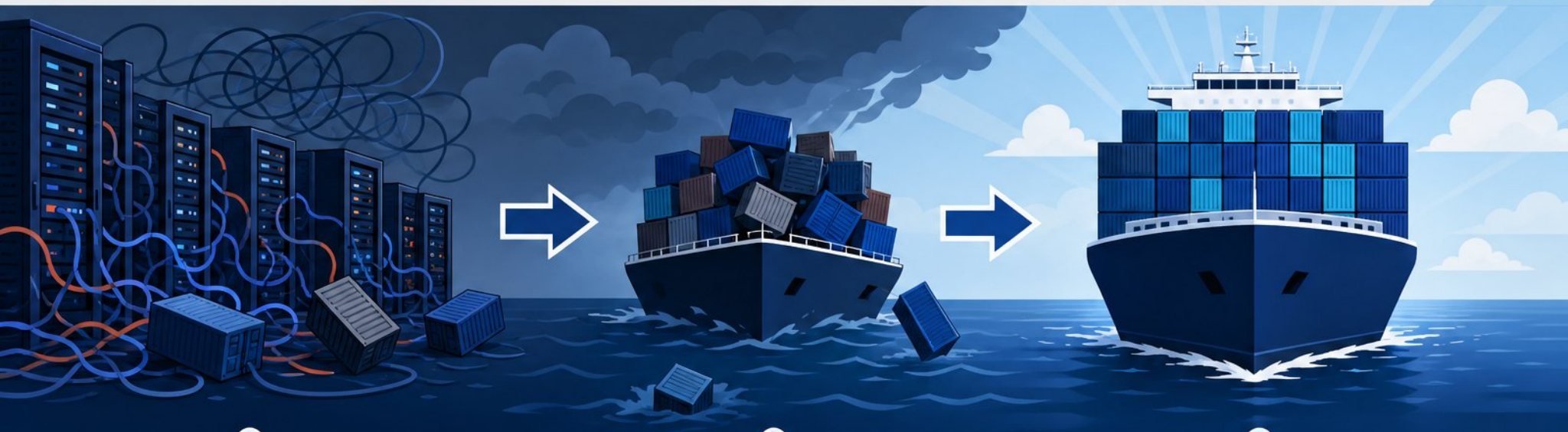
Kubernetes in DevOps:
Concepts, Purpose, and Examples

- Core concepts explained: what Kubernetes is and how it works
- Purpose: why Kubernetes became the standard for DevOps
- Real-world examples: deploying, scaling, and managing applications
- Balance of theory, business value, and hands-on practice

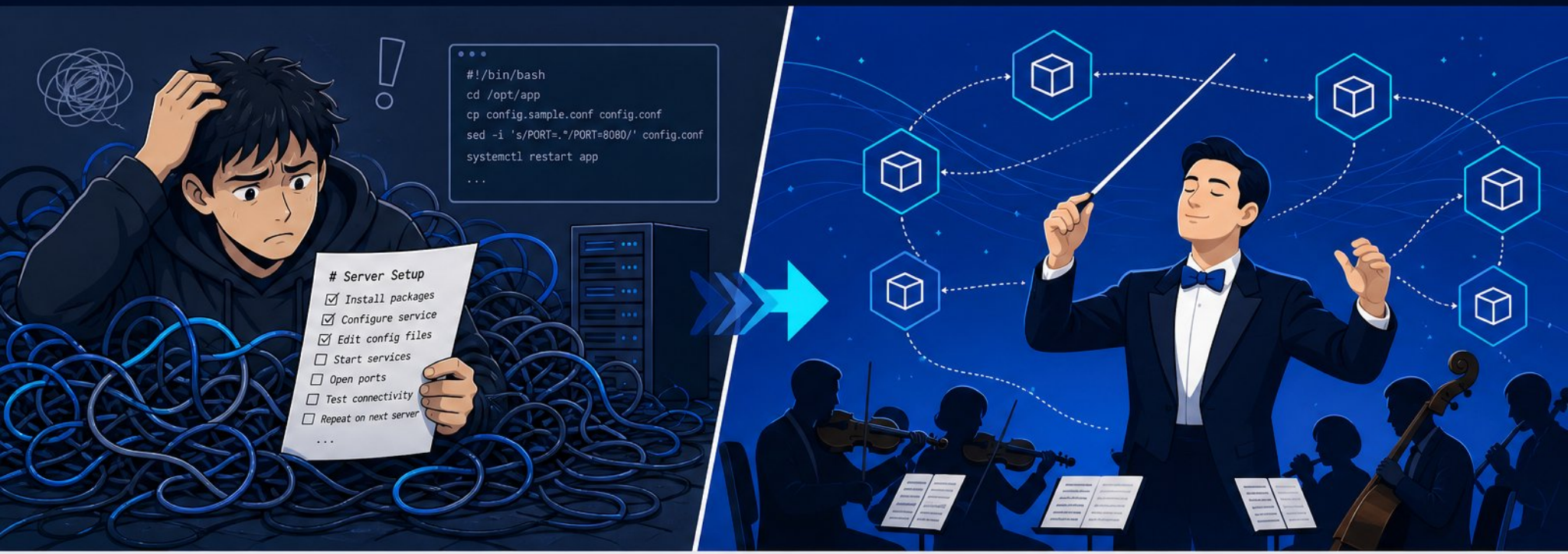


What Kubernetes Is and Why It Became the Standard

- Kubernetes is an open-source platform for automating container deployment, scaling, and management
- Originated from Google's internal Borg system; now a Cloud Native Computing Foundation (CNCF) project
- Containers solved the "works on my machine" problem; Kubernetes solved "how do I run this at scale"



The Problem: From Manual Deployments to Orchestration



- Pre-Kubernetes era: manual server config, error-prone shell scripts, inconsistent environments
- Containers alone aren't enough when managing dozens or hundreds of services
- Kubernetes provides declarative automation: define desired state, let the system reconcile to it

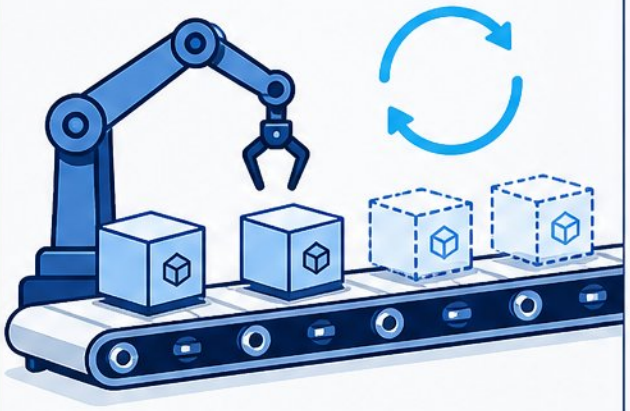
Kubernetes Core Concepts: Cluster, Nodes, and Pods

- A cluster has a control plane and worker nodes
- Pods are the smallest deployable unit in Kubernetes
- Control plane runs the API server, scheduler, and etcd
- Declarative configs describe the desired cluster state
- Kubernetes continuously works to reach that state

BUILDING BLOCKS

Deployments, Services, and Namespaces

DEPLOYMENT

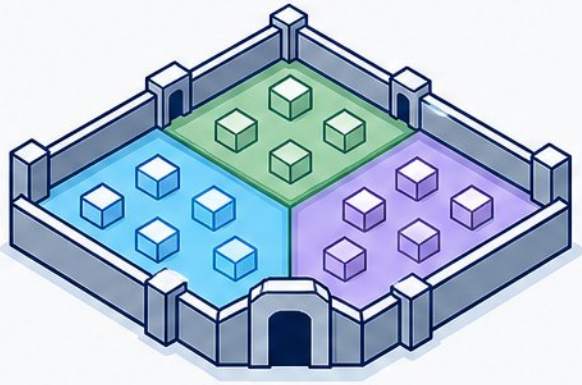


Manages replicas, enabling rolling updates and rollbacks

SERVICE

Provides stable networking and load balancing for pods

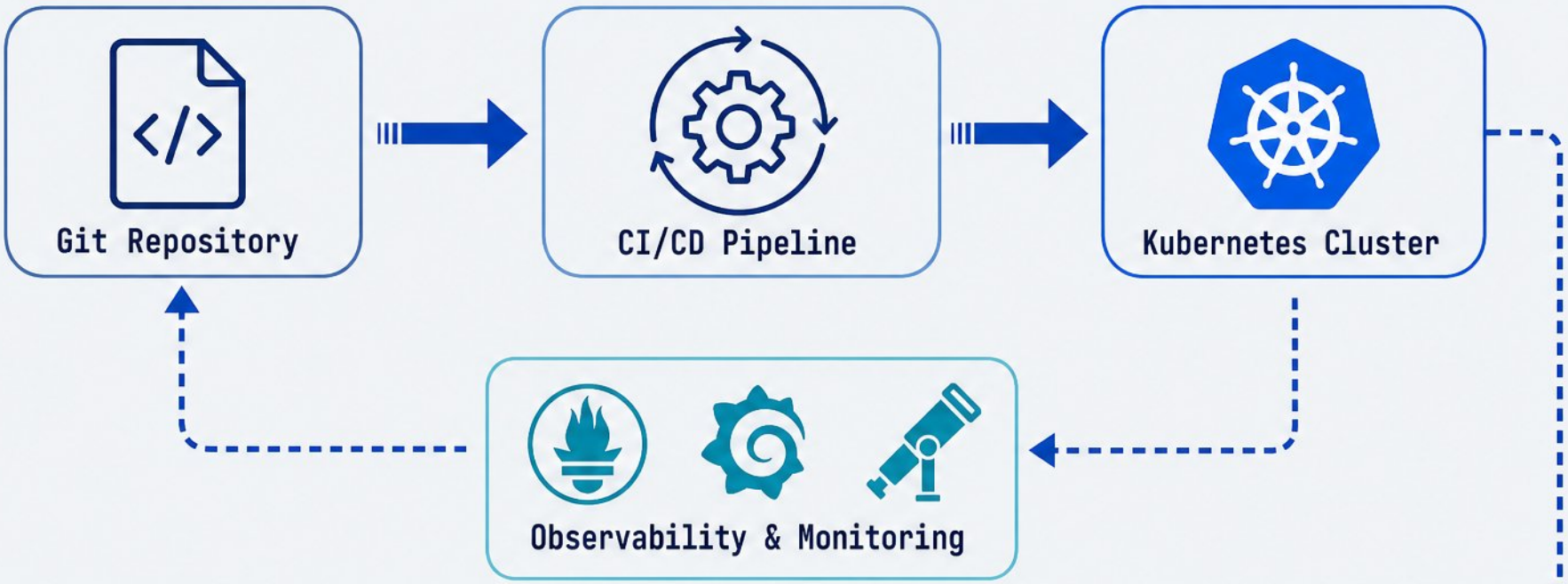
NAMESPACE



Creates logical isolation and resource boundaries

- Deployments manage replicas, enabling rolling updates and rollbacks
- Services provide stable networking and load balancing for pods
- Namespaces create logical isolation and resource boundaries
- These Kubernetes objects form the core of application management

Kubernetes in the DevOps Lifecycle



- - Kubernetes is the deployment target for CI/CD pipelines
- - GitOps: Git as source of truth with Argo CD/Flux
- - Prometheus, Grafana, OpenTelemetry for observability
- - Declarative automation, self-healing, and auto-scaling

Purpose and Benefits for Developers

- Applications behave identically from laptop to production
- Rolling updates, rollbacks, and self-healing guard quality
- Horizontal scaling handles traffic spikes automatically
- Self-service workflows give developers autonomy without risk

Purpose and Benefits
for Platform Engineers

- Shared infrastructure via namespaces, RBAC, and resource quotas
- Standard patterns with Helm charts, operators, and admission controllers
- Policy-as-code for security using Kyverno or OPA Gatekeeper
- Cost allocation and observability wired in by default
- “Golden paths” make the easy way the safe, standard way

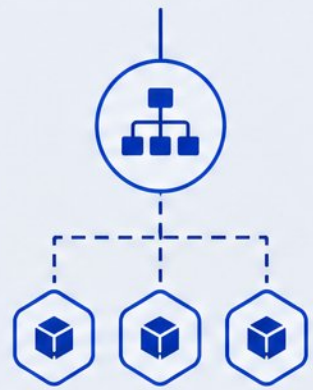
Example: Deploying and Scaling a Service

- - Create a Deployment manifest (YAML) for a sample app
- - Expose the app with a Service for internal access
- - kubectl scale adjusts the replica count on demand
- - Update the image to trigger a rolling update

Deployment (YAML)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
       - name: app
         image: sample-app:1.0
```

Service



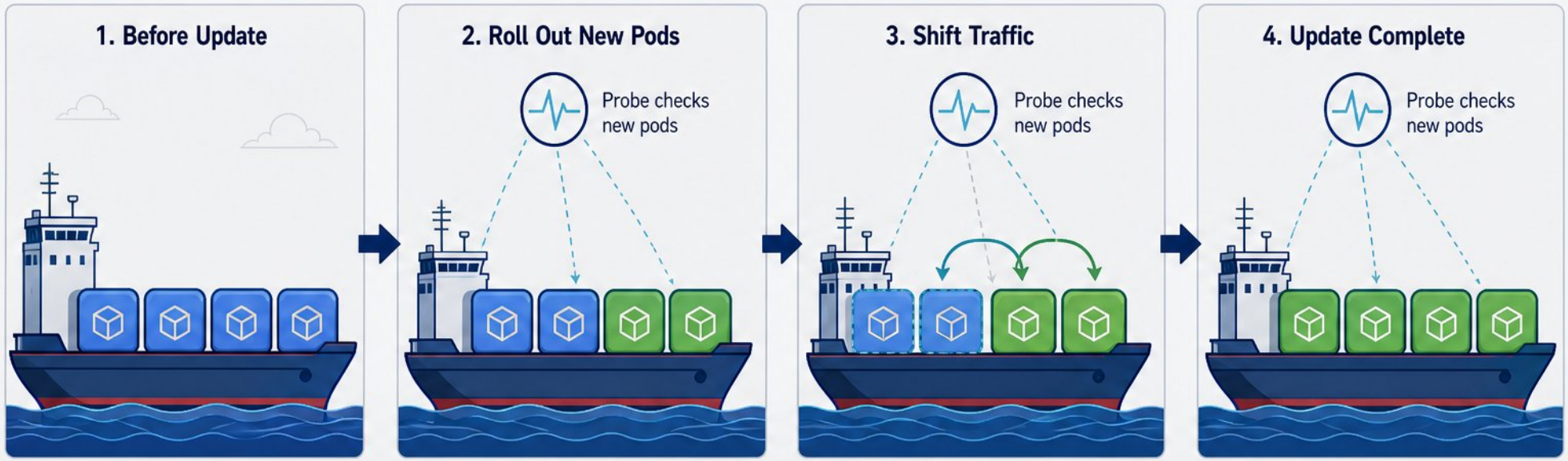
Internal Access

```
> $ kubectl scale deployment sample-app
  --replicas=<count>
```

Update the image
to trigger a rolling update

Example: Rolling Updates, Rollbacks, and Probes

- - Rolling updates start new pods before terminating old ones, avoiding downtime
- - kubectl rollout status tracks progress; use rollout undo for failed releases
- - Readiness probes route traffic only to healthy pods
- - Liveness probes automatically restart unhealthy containers



All traffic is served by existing (old) pods.

New pods are started and probes verify their health.

Traffic is routed to healthy new pods; old pods are drained.

Old pods are terminated. Only new pods remain.

```
kubectl rollout status tracks progress; use rollout undo for failed releases
```

Common DevOps Patterns and Pitfalls

- GitOps: Git as source of truth, Argo CD or Flux syncs clusters
- Set resource requests and limits to prevent noisy-neighbor issues
- Avoid latest tags; mutability breaks rollbacks and causes split-brain
- Add health probes to correctly route traffic and enable self-healing
- Use labels and namespaces to reduce operational complexity

When Not to Use Kubernetes

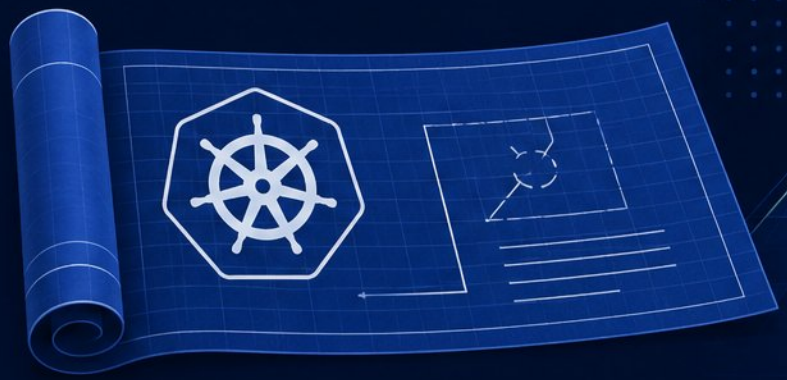
- Small teams with 1-3 services may not need Kubernetes
- Simple apps without high availability needs can use managed container services
- Containers alone may suffice for basic deployments
- Avoid over-engineering from day one
- Start simple and add complexity only when needed

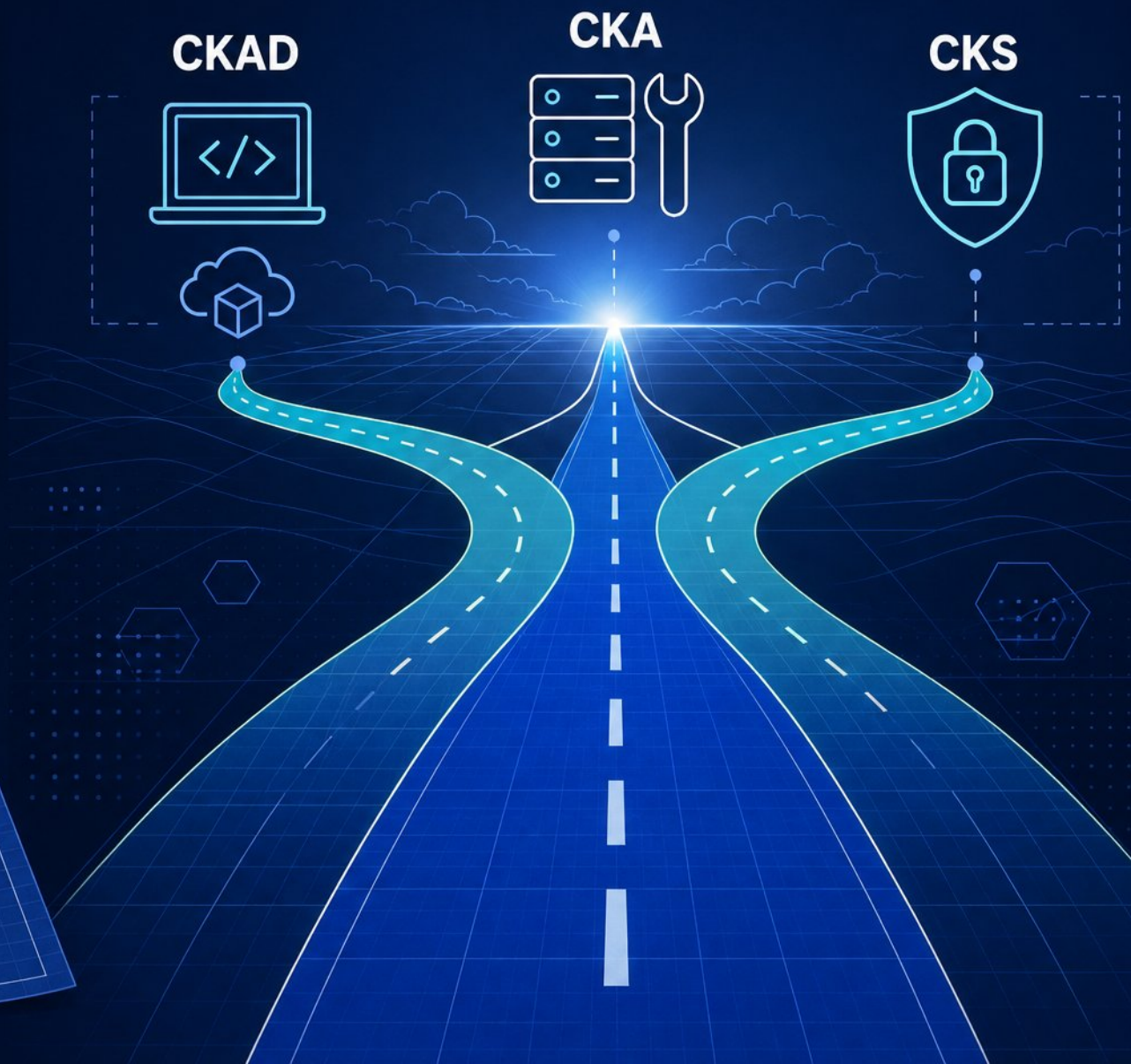
Platform Engineering: Hiding Kubernetes from Developers

- Platform engineering builds Internal Developer Platforms (IDPs) as products
- Golden paths provide paved, opinionated routes to production
- Tools like Backstage, Argo CD, and Crossplane abstract Kubernetes complexity
- Guardrails and observability are built into every layer
- Goal: developers ship code without knowing what a Pod is

Wrap-Up and Next Steps

- Recap: declarative automation, orchestration, and GitOps patterns
- CKAD for developers deploying applications on Kubernetes
- CKA for operators: cluster setup, maintenance, and troubleshooting
- CKS for security: requires CKA, covers cluster hardening and runtime security
- Next: Helm charts, Argo CD/Flux GitOps, observability, and security





PersonWise.

PersonWise • AI digital-human course platform

Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/cf2409ab/public