**Deployment Strategy in Kubernetes**

A deployment strategy in Kubernetes defines how new versions of applications are rolled out while replacing the old ones — it helps in minimizing or eliminating downtime and ensures a smooth user experience.

**Need of a Deployment Strategy**

**Without a strategy:**

You might delete old version (V1) and install new version (V2) manually.

This causes complete downtime during the transition.

For example, a payment service in a company like Amazon or Instagram being down even for 10 minutes can result in:

* Huge revenue loss
* Bad user experience
* Reputation damage

**Advantages of Using Deployment Strategies:**

* Minimizes downtime (possibly zero downtime)
* Allows rollback in case of failure
* Improves application reliability
* Supports gradual upgrades and testing in production

**Popular Deployment Strategies in Kubernetes:**

1. **Rolling Update (Default Strategy)**

**USAGE**

When you need to upgrade applications with zero or near-zero downtime

Default strategy when you use kubectl apply or kubectl set image

**Function**

Instead of uninstalling the old version and then installing the new version:

Kubernetes gradually replaces old pods (version V1) with new pods (version V2)

It starts by adding one new pod (V2)

Waits until the new pod is “Ready”

Then removes one old pod (V1)

Repeats the process until all pods are upgraded

**Key Configuration Parameters:**

MaxUnavailable: Number (or %) of pods that can be unavailable during update (default: 25%)

MaxSurge: Extra pods allowed during update (default: 25%)

**For example:**

If there are 4 pods, 25% = 1 pod

So Kubernetes adds 1 V2 pod → waits → removes 1 V1 pod → repeats

**Important:**

Always configure readinessProbe:

Helps Kubernetes know when a pod is actually ready to serve traffic

If not configured, Kubernetes might assume the pod is ready just because the container is running (which can cause downtime)

**Why isn't it exactly zero downtime?**

Application-level delays such as:

Database migration

Integration with external services

Configuration issues

These can introduce brief service interruptions despite using rolling update

**Advantage of Rolling Update**:

Despite failure of V2, 3 V1 replicas are still serving traffic.

No full downtime → application continues to serve users.

Readiness probe is crucial to validate pod health before replacement.

**Why Need for Another Strategy Like Canary if Rolling Update is Good**

**Key Limitation of Rolling Update:**

Even though pods are replaced gradually, the process

Auto-scales based on maxUnavailable & maxSurge.

Cannot delay or pause based on real user feedback.

Finishes update in minutes, not ideal for long monitoring/testing.

**Canary Deployment Strategy**

**When to Use**:

When you want to test a new version (V2) in real production with limited users.

**Ideal for**:

Critical applications

Complex integrations

Live user feedback collection

**Organizational Release Pipeline (Why Canary Exists):**

Dev/UAT → Unit tested

QA → Test data validated

Pre-prod → Close to real environment

Prod → Final target, but not tested with real prod traffic/data yet

Even if it worked in QA/pre-prod, it might fail in prod due to real-world differences:

Data size, user patterns, cluster specs, etc.

**How Canary Works:**

* Deploy V2 alongside V1 in production (e.g., 1 replica of V2, 3 replicas of V1).
* Use Ingress controller (e.g., NGINX) to configure traffic splitting:
* Send 90% traffic to V1

Send 10% traffic to V2

Monitor V2 performance, collect logs, user behavior.

Based on feedback:

Increase traffic to V2 gradually (20%, 30%, … 100%)

Or rollback by routing 100% back to V1

**Summary: Why Canary is Powerful**

* Helps test in production without full exposure.
* Reduces risk — only a subset of users sees new changes.
* Easy to control traffic flow and rollback instantly.
* Ideal for mission-critical or sensitive applications.

**3. Blue-Green Deployment**

Maintains two identical environments: one serving users (Blue), the other idle (Green). Once new version is ready in Green, switch all traffic to it.

**Why It's Used:**

* Ensures the safest and instant switchover deployment.
* Full test of the new version before users access it.
* Supports instant rollback by switching back.

**How It Works (Step-by-Step):**

* Deploy version 1 (v1) to Blue:

4 replicas

Service1 and Ingress routing to v1

* Deploy version 2 (v2) to Green:

4 replicas

Service2 and identical Ingress configuration

**When to Use:**

Highly critical applications with zero risk tolerance.

Systems where rollback must be instantaneous.

Environments where downtime is not an option.

**Pros & Cons:**

**Pros Cons**

| Safest deployment strategy | \*\*Most expensive\*\*: duplicates all infrastructure |

| Instant rollback | High memory and CPU usage (double the resources)

| Full validation before switch | Needs proper traffic switch mechanism |

| Avoids partial upgrades | Not scalable for apps with 100s of services |

<https://youtu.be/cWUbkuzc8dM?si=ygyxwCjB6iz-HKTX>

<https://youtu.be/O61HDmGUBJM?si=TBqTaxkOWwQKbsUy>

<https://youtu.be/uBhjymTV0ro?si=U9YMjIj0m9ULYkvH>