Platform Engineering UNIVERSITY

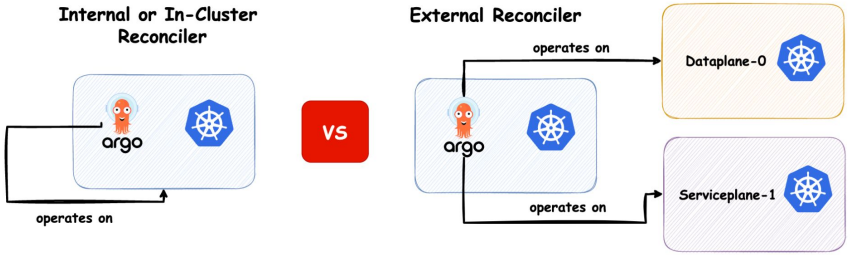
GITOPS FOR PLATFORM ENGINEERING

# GitOps Architecture, Patterns and Anti-Patterns

MODULE 03

# GitOps Architecture

# GitOps Architecture

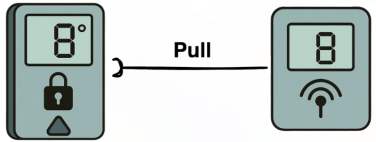



Remember this.

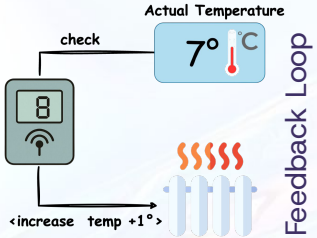
## Declarative

## Versioned and Immutable

## Pull-Based

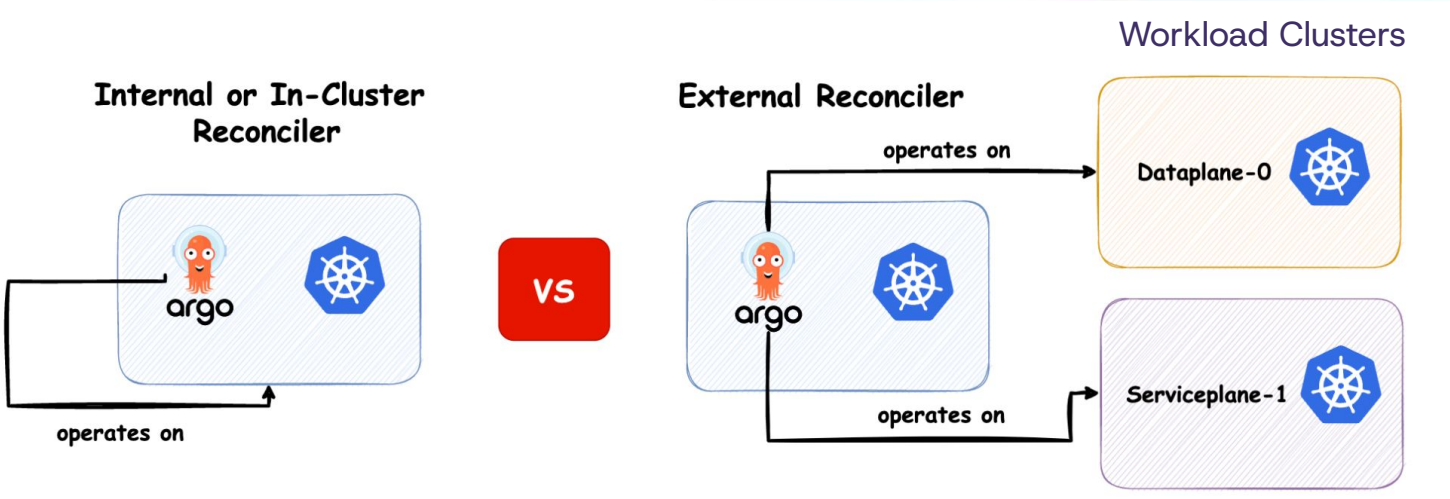



## Continuously Reconciled

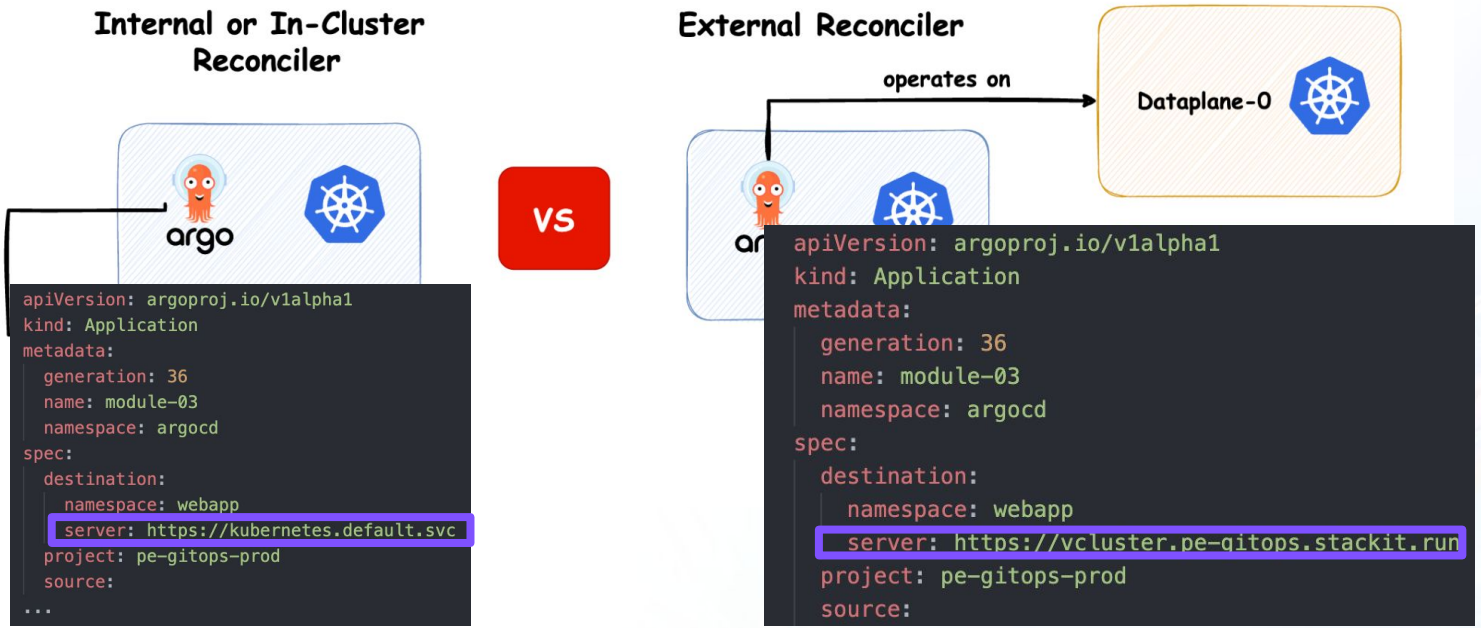



Just four simple principles—no more, no less.

# Internal vs External Reconciler



# Internal vs External Reconciler



# Declarative

Declarative means defining the desired "what" (the end state) and letting the system handle the "how" (the steps to get there).

**Definition (YAML):** `replicas: 3`

Imperative is a list of commands; Declarative is a description of the destination

```
kubectl scale deployment my-app --replicas=3
```

> Imperative is like giving a driver turn-by-turn directions; Declarative is giving the driver an address and letting them find the best route

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  namespace: webapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: ghcr.io/stefanprodan/podinfo:6.9.4
        ports:
        - containerPort: 9898
        resources:
          requests:
            cpu: 100m
            memory: 32Mi
          limits:
            cpu: 1000m
            memory: 128Mi
```

# Versioned and Immutable

Versioned and immutable means **every change** is identified by a **unique**, permanent tag (like a Git hash or Docker digest) so that once a version is deployed, it **can never** be altered—only **replaced** by a newer version.

❌ The Bad Way (Mutable)

```
- name: frontend
  # DANGEROUS: You don't know what's inside
  image: ghcr.io/stefanprodan/podinfo:latest
```

✅ The Good Way (Versioned & Immutable)

```
- name: frontend
  # Versioned: Clear release tag
  image: ghcr.io/stefanprodan/podinfo:6.9.4
  # OR even better: Immutable via Digest (SHA)
  # image: ghcr.io/stefanprodan/podinfo@sha256:82138e053dc84...
```

Tags are for humans (versioned); Hashes are for machines (immutable); **Never** use 'latest' in production.

# Pull-Based

In a Pull-based system, an agent inside the cluster continuously monitors Git and 'pulls' changes to synchronize the state, ensuring the cluster always matches the source of truth.

### ❌ The Push Model (Traditional CI/CD)

- **Code:** `git push` -> CI/CD Server runs `kubectl apply -f deployment.yaml`
- **Risk:** You have to store sensitive Kuberentes credentials (Kubeconfig) inside your CI tool.

### ✅ The Pull Model (GitOps)

- **Code:** `git push` -> **Argo CD** (inside the cluster) detects the change and pulls it.
- **Benefit:** No cluster credentials leave the cluster. Even if someone manually deletes a pod, the Pull-based agent will "pull" the correct state back.

Push-based is 'Fire and Forget' from the outside; Pull-based is 'Watch and Sync' from the inside.

# Continuous Reconciled

Continuously Reconciled means the system constantly compares the observed state of the cluster with the desired state in Git and automatically fixes any discrepancies (drift) without human intervention.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-cm
  namespace: argocd
  labels:
    app.kubernetes.io/name: argocd-cm
    app.kubernetes.io/part-of: argocd
data:
  timeout.reconciliation: 180s #3 min
```

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: frontend-app
spec:
  syncPolicy:
    automated: # This enables Continuous Reconciliation
      prune: true    # Automatically delete resources removed from Git
      selfHeal: true # Automatically fix manual changes in the cluster (Drift)
```

### The "Loop" Logic

- **Observe:** Argo CD looks at the running Pods in your `webapp` namespace.
- **Diff:** Argo CD looks at the YAML in your Git repo.
- **Act:** If a Pod was manually deleted or a replica count was changed via `kubectl`, Argo CD "reconciles" it by re-applying the Git state.

Continuous Reconciliation turns 'I hope it's running' into 'I know it's running' by constantly killing the drift.

```
Context: ske-pe-pro [RW]
Cluster: ske-pe-pro
User:    ske-pe-pro
K9s Rev: v0.50.6 ⚡v0.50.16
K8s Rev: v1.34.2
CPU:     3%
MEM:     43%
```

<0> all
<1> webapp
<2> external-dns
<3> kyverno
<4> mgmt
<5> projectsveltos

pods(webapp)[3]

| NAME↑ | PF | READY | STATUS | RESTARTS | CPU | MEM | %CPU/R | %CPU/L | %MEM/R | %MEM/L | IP |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frontend-b5fd785df-jkd5j | ● | 1/1 | Running | 0 | 2 | 14 | 2 | 0 | 44 | 11 | 100.6 |
| frontend-b5fd785df-xkfw2 | ● | 1/1 | Running | 0 | 2 | 18 | 2 | 0 | 59 | 14 | 100.6 |
| frontend-b5fd785df-zm5br | ● | 1/1 | Running | 0 | 2 | 16 | 2 | 0 | 52 | 13 | 100.6 |

<namespace> <pod>

argo v3.2.1+8c4ab63

Applications
Settings
User Info
Documentation

Resource filters

NAME
KINDS
SYNC STATUS
- Synced 4
- OutOfSync 0

HEALTH STATUS
- Progressing 0
- Suspended 0
- Healthy 6
- Degraded 0
- Missing 0
- Unknown 0

Applications / module-03

APPLICATION DETAILS TREE

DETAILS | DIFF | SYNC | SYNC STATUS | HISTORY AND ROLLBACK | DELETE | REFRESH

Log out

APP HEALTH
Healthy

SYNC STATUS
Synced to HEAD (55a9693)
Auto sync is enabled.
Author: Artem Lajko <110809333+la-cc@u...

LAST SYNC
Sync OK to 55a9693
Succeeded 4 minutes ago (Tue Dec 23 2025 01:23:46 GMT+0100)
Author: Artem Lajko <110809333+la-cc@u...

100%

module-03
7 minutes

webapp
ns
7 minutes

frontend
svc
5 minutes

webapp
sa
5 minutes

frontend
deploy
4 minutes rev:1

# Progressive Delivery

# Progressive Delivery

Is the umbrella term. It's the overarching philosophy or discipline of modern software releases.


## Rolling Update

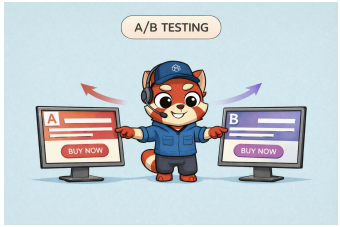
Replaces pods one by one (incrementally).

**Standard availability** with minimal resource overhead.


## Shadow/Mirroring

Mirrors live traffic to the new version without the user seeing the result.

**Performance & Load** testing under real conditions.




## A/B Testing

Distributes traffic based on user attributes (region, browser, ID).

**Business metrics** (Which version converts better?).


## Blue/Green

Two identical environments; switches 100% traffic from Blue (old) to Green (new).

**Zero Downtime** and instant rollbacks.


## Canary

Routes a small percentage of traffic (e.g., 5%) to the new version to test it.

**Risk mitigation** by testing on **real** users.

While "**Continuous Delivery**" focuses on moving code from the developer to production as fast as possible, **Progressive Delivery** focuses on **how** that code is exposed to users to minimize risk

# Progressive Delivery

Is the umbrella term. It's the overarching philosophy or discipline of modern software releases.


## Rolling Update

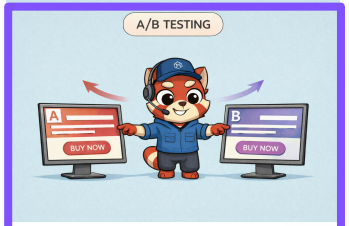
Replaces pods one by one (incrementally).

**Standard availability** with minimal resource overhead.


## Shadow/Mirroring

Mirrors live traffic to the new version without the user seeing the result.

**Performance & Load** testing under real conditions.




## A/B Testing

Distributes traffic based on user attributes (region, browser, ID).

**Business metrics** (Which version converts better?).


## Blue/Green

Two identical environments; switches 100% traffic from Blue (old) to Green (new).

**Zero Downtime** and instant rollbacks.


## Canary

Routes a small percentage of traffic (e.g., 5%) to the new version to test it.

**Risk mitigation** by testing on **real** users.

While "**Continuous Delivery**" focuses on moving code from the developer to production as fast as possible, **Progressive Delivery** focuses on **how** that code is exposed to users to minimize risk

# Trunk-Based vs Branch-Based Development

# Trunk-Based vs Branch-Based Development

**Trunk-Based Development**
In this model, developers collaborate on a single branch (usually main or master). Short-lived feature branches are merged as quickly as possible.

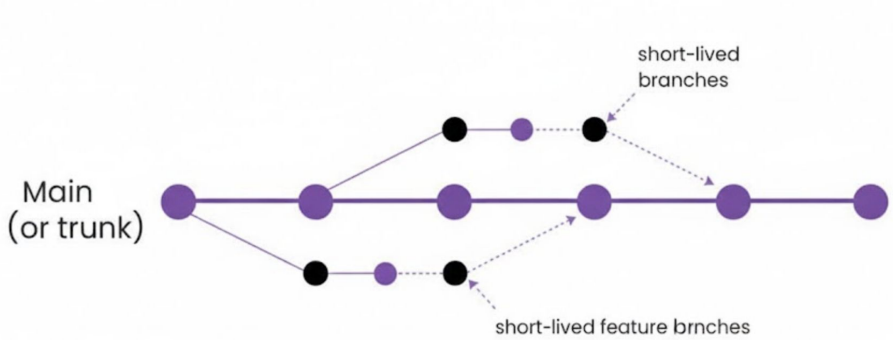
- **How it works in GitOps:** Argo CD tracks the main branch. As soon as a commit is pushed or merged into main, Argo CD automatically syncs those changes to the cluster.
- **Focus:** Speed, Continuous Integration, and avoiding "merge hell."
- **Best for:** Teams with high automation, automated testing, and a "fail forward" mentality.

**Branch-Based Development (Env per Branch):**
In this model, different Git branches represent different environments (e.g., a staging branch and a production branch).

- **How it works in GitOps:** Argo CD has multiple "Applications." The Staging App tracks the staging branch, and the Production App tracks the production branch. To deploy to production, you perform a Pull Request (PR) from staging to production.
- **Focus:** Control, manual gates, and clear separation between environment states.
- **Best for:** Regulated industries or teams that require manual sign-offs before a production release.

Trunk-based is about **Continuous Deployment**, while Branch-based is about **Release Control**

# Promotions between Stages: Trunk-Based




- **No Branch Merges:** We don't merge `staging` into `production`. Both environments track `main`.
- **Environment-Specific Values**: We use separate `values.yaml`
- **Promotion = Tag Update:** Promoting a version means updating the `image.tag` in the production configuration files.
- **Decoupled:** Code is integrated into `main` immediately (CI), but deployed to production only when the configuration is updated (CD).

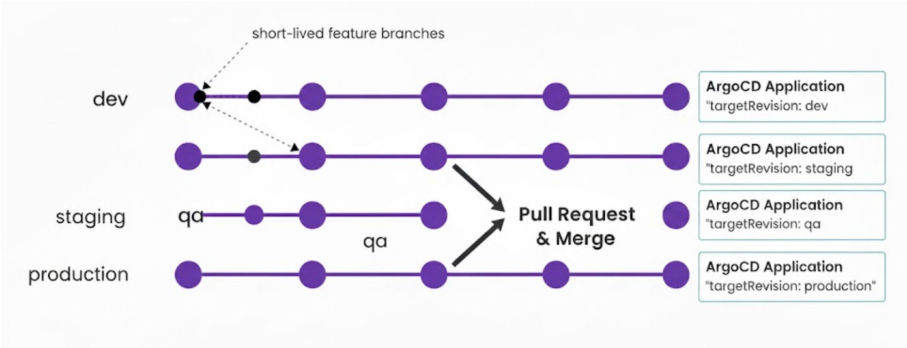
```yaml
# staging-app.yaml (ArgoCD)
targetRevision: main
helm:
  values: |
    image:
      tag: "v1.2.0-rc1" # Promoted first for testing

---

# production-app.yaml (ArgoCD)
targetRevision: main
helm:
  values: |
    image:
      tag: "v1.1.0" # Remains stable until staging is verified
```

> In Trunk-Based GitOps, we promote 'Artifact Versions,' not 'Git Branches.**

# Promotions between Stages: Branch-Based




```yaml
# Application: dev-cluster
spec:
  source:
    repoURL: https://github.com/org/app
    targetRevision: dev # Watches the dev branch

---
# Application: prod-cluster
spec:
  source:
    repoURL: https://github.com/org/app
    targetRevision: main # Watches the main/prod branch
```

- **Environment = Branch:** The state of the `staging` branch is exactly what is running in the Staging cluster.
- **Promotion via Merge:** Moving code from Dev to Staging requires a Git Merge/Pull Request.
- **Visibility:** You can use `git diff dev..staging` to see exactly what is about to be deployed.
- **Permissions:** Git branch protection rules (e.g., "Only Artem can merge to `main`") act as deployment gates.

> This architecture is categorized as a GitOps anti-pattern and is not recommended for production-scale environments.

# Promotions between Stages: Kargo as Bridge

- **Continuous Promotion:** Instead of manually editing YAML tags for every environment, Kargo orchestrates the "Promotion" of Freight (a bundle of Git commits, Images, and Helm charts) across stages.
- **Trunk-Based but Controlled:** It allows you to keep a single branch (Trunk-based) while providing the visual Promotion Gates (Dev → Staging → Prod) that teams usually try to solve with messy branch-based GitFlow.
- **Decoupled from CI:** Kargo stops the "CI-as-CD" anti-pattern. You don't need GitHub Actions or Jenkins to "push" changes; Kargo manages the lifecycle natively within Kubernetes.


✖ **Before Kargo:** You write a custom bash script in GitHub Actions to `sed` an image tag in a YAML file and commit it to another branch. (Fragile & complex).

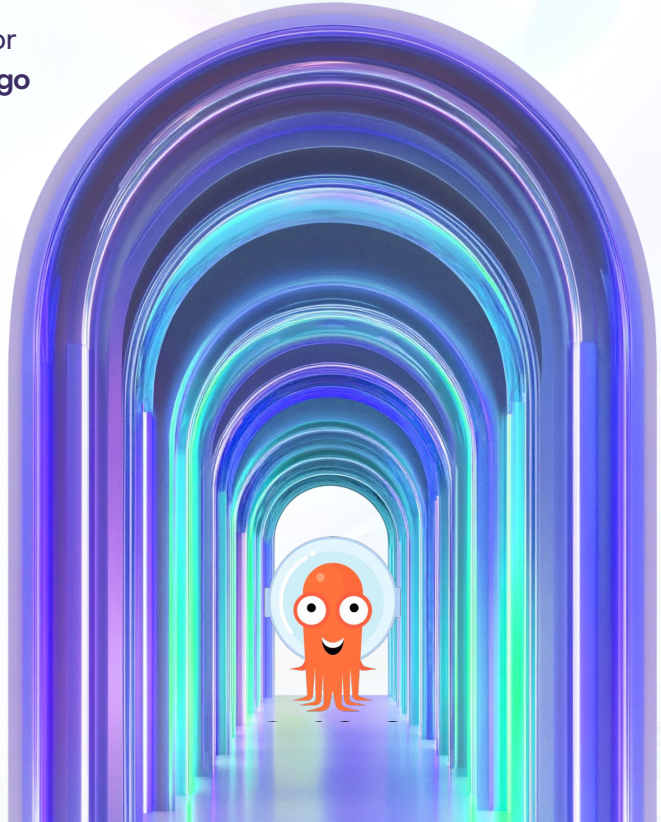
✓ **With Kargo:** You define a Warehouse and Stages. You click "Promote" in a UI (or via CLI). Kargo handles the GitOps-compliant state change automatically.

# Argo CD: Pull-Request Generator (I)

**The PR Generator** is a feature of the **Argo CD ApplicationSet** controller.

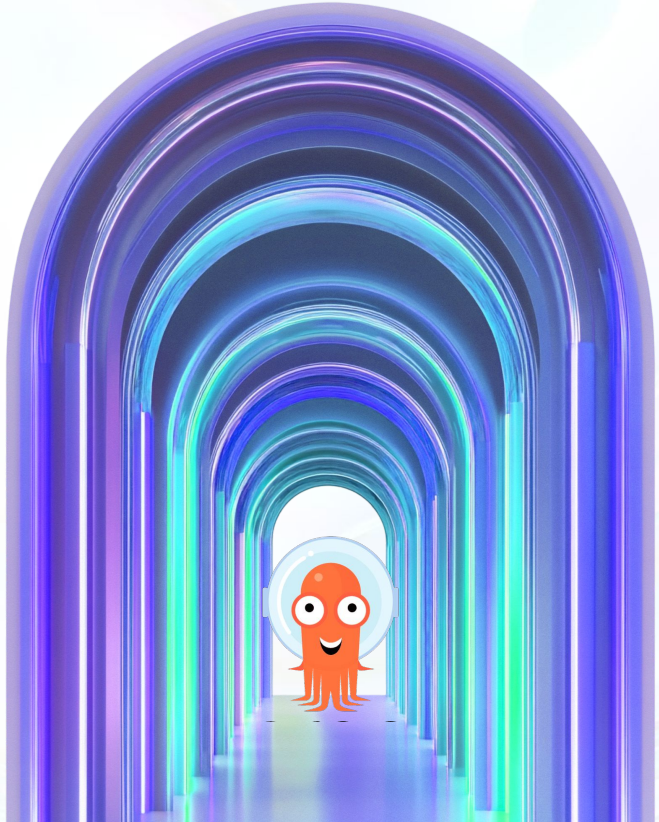
Normally, Argo CD needs a folder in Git to create an application. The PR Generator changes this: **It treats an open Pull Request as a trigger to create a temporary Argo CD Application automatically.**

1. **The Trigger:** A developer creates a new branch (e.g., `feat-login`) and opens a **Pull Request** against the `main` branch.
2. **The Discovery:** The PR Generator constantly polls your Git provider (GitHub/GitLab). It "sees" the new PR.
3. **The Templating:** The Generator uses a template to create a new Argo CD Application on the fly. It can use variables from the PR, such as:
    a. `{{.branch}}`: The branch name (e.g., `feat-login`).
4. **The Deployment:** Argo CD creates a **temporary namespace** (e.g., `preview-pr-123`) and deploys the app there.



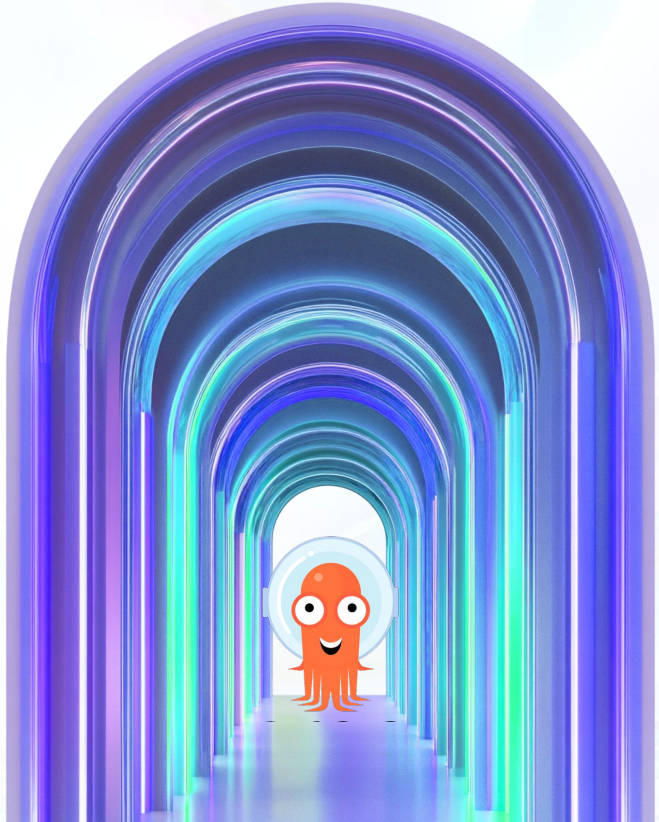
# Argo CD: Pull-Request Generator (II)

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: web-app-preview-environments
  namespace: argocd
spec:
  goTemplate: true # Enables advanced templating like {{.branch_slug}}
  generators:
    - pullRequest:
        github:
          owner: my-organization      # Your GitHub Org/User
          repo: my-app-source-code    # The Repo Kargo/Argo watches for PRs
          # labels: ["preview-ready"] # Optional: Only deploy if this label is set
        requeueAfterSeconds: 60       # How often to check for new PRs
  template:
    metadata:
      # Creates a unique name like: web-app-pr-42
      name: 'web-app-pr-{{.number}}'
    spec:
      project: default
      source:
        repoURL: 'https://github.com/my-organization/k8s-manifests.git'
        targetRevision: HEAD
        path: charts/my-app
        helm:
          # This is where the MAGIC happens:
          # We override the image tag with the Git Hash from the PR
          parameters:
            - name: "image.tag"
              value: '{{.head_sha}}'
            - name: "ingress.host"
              value: 'pr-{{.number}}.preview.example.com'
      destination:
        server: https://kubernetes.default.svc
        # Deploys each PR into its own isolated namespace
        namespace: 'preview-pr-{{.number}}'
```



# Argo CD: Pull-Request Generator (III)

```yaml
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: web-app-preview-environments
  namespace: argocd
spec:
  goTemplate: true # Enables advanced templating like {{.branch_slug}}
  generators:
    - pullRequest:
        github:
          owner: my-organization      # Your GitHub Org/User
          repo: my-app-source-code    # The Repo Kargo/Argo watches for PRs
          # labels: ["preview-ready"] # Optional: Only deploy if this label is set
        requeueAfterSeconds: 60       # How often to check for new PRs
  template:
    metadata:
      # Creates a unique name like: web-app-pr-42
      name: 'web-app-pr-{{.number}}'
    spec:
      project: default
      source:
        repoURL: 'https://github.com/my-organization/k8s-manifests.git'
        targetRevision: HEAD
        path: charts/my-app
        helm:
          # This is where the MAGIC happens:
          # We override the image tag with the Git Hash from the PR
          parameters:
            - name: "image.tag"
              value: '{{.head_sha}}'
            - name: "ingress.host"
              value: 'pr-{{.number}}.preview.example.com'
      destination:
        server: https://kubernetes.default.svc
        # Deploys each PR into its own isolated namespace
        namespace: 'preview-pr-{{.number}}'
```



# Repository Strategies

# Repository Strategies: Mono vs Multi Repo(s)

**Mono Repo (The "One for All")**
All code, Helm charts, and Kubernetes manifests for all services/teams are **kept in a single, large Git repository.**

- **How it works in GitOps:** Argo CD points to different folders within the same repo
- **Focus:** Visibility, cross-service consistency, and simplified dependency management.
- **Best for:** Smaller teams or organizations that want to share code/templates easily and maintain a global overview.

**Multi Repo (The "One per Service")**
Each service or team has its **own dedicated Git repository** for its code and manifests.

- **How it works in GitOps:** Argo CD has many "Application" objects, each pointing to a different repository.
- **Focus:** Autonomy, isolation, and granular access control (RBAC).
- **Best for:** Large organizations with many independent teams where Team A should not be able to touch Team B's code or configuration.

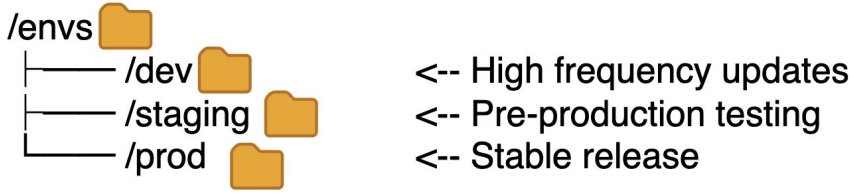
Mono Repo prioritizes shared consistency and visibility; Multi Repo prioritizes team autonomy and security boundaries.

# Folder per Environment (I)

**The Core Philosophy**
Instead of using Git branches (which lead to "Merge Hell"), we use folders within a single branch (usually `main`).

- ➔ **Single Source of Truth:** One branch to rule them all.
- ➔ **Promotion via Copy:** Moving a release is a simple `cp` (copy) command between folders.
- ➔ **Clarity:** A quick look at the `/envs` folder shows exactly what is deployed where.

**Simple (Stage-Based)**
Best for small teams or single-region applications. It follows the software lifecycle.

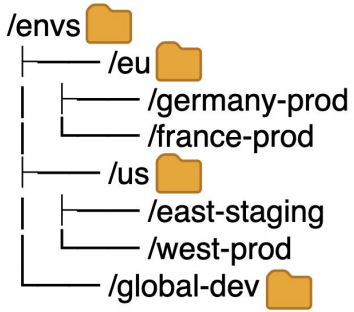



**Best for:** Projects where "Prod" is a single cluster.

**Note:** `staging` and `prod` should inherit from the same `variants/stable` folder to ensure they remain identical in configuration.

# Folder per Environment (II)

**The Geographic Matrix (Region/Country-Based)**
Required for global applications with data residency (GDPR) or latency requirements.




**Best for:** Large scale apps where "Prod" is not one place, but many.

**Note:** This structure allows you to use *Kustomize* Components to apply "EU-only" settings (like specific database endpoints) to all folders under the `/eu` tree.

**The Specialized Hardware/Feature Pattern**
Used when environments differ by technical capabilities (e.g., AI/ML workloads).


**Best for:** Performance testing, GPU-heavy workloads, or cost-optimization (using smaller clusters for QA).

# Folder per Environment (III)

| Pattern | Complexity | Best Use Case |
| --- | --- | --- |
| **Stage-only** | Low | Start-ups, single apps. |
| **Stage + Region** | Medium | SaaS companies with global users. |
| **Stage + Variant** | High | Specialized tech (Edge computing, AI, Finance). |

# Folder per Environment (IV)

| Selection Criteria | Recommended Grouping | Example Structure |
| --- | --- | --- |
| **Low Complexity** | Stages | envs/dev, envs/prod |
| **High Compliance** | Countries | envs/germany, envs/usa |
| **Multi-Cloud Setup** | Cloud Providers | envs/aws-eu, envs/azure-us |
| **Specific Hardware** | Variants | envs/gpu-cluster, envs/cpu-cluster |
| **Global Scaling** | Regional Matrix | envs/prod-eu, envs/prod-us |

# State Store

# State Store: Git, OCI or ConfigHub (I)

**The State Store** is the ***"Source of Truth"*** for your desired Kubernetes state. While Git is the default, modern GitOps architectures often use OCI or ConfigHub.

**Git** (The Standard) 

**OCI** – Open Container Initiative (The Modern Way) 

**ConfigHub** (The New Solution on the Market) 


> Don't confuse the **Development Source (Git)** with the **Distribution Source** (State Store). You can write code in Git, but use a CI pipeline to push it to **OCI** for the actual deployment.

> Using **OCI** as a State Store is becoming the gold standard for enterprises because it treats YAML exactly like code—versioned, packaged, and signed

# State Store: Git, OCI or ConfigHub (II)

## Git

**Mechanism:** Direct sync from a Git repository (GitHub, GitLab, Bitbucket).

**Best For:** Most teams; provides full history, Pull Request workflows, etc.

**Challenge:** Large repositories with thousands of files can cause performance bottlenecks (slow polling) for the GitOps controller.

## OCI

**Mechanism:** Packaging Kubernetes manifests into an OCI Artifact (the same format as Docker images) and pushing them to a Registry (GHCR, Harbor, ECR).

**Best For:** Scaling GitOps. Decouples the "Source" (Git) from the "Release" (Registry).

**Advantages: * Faster Sync:** Argo CD/Flux pulls a single compressed artifact instead of cloning a whole Git history.

- Immutable Releases: Every version is a signed, immutable "Logical Container" of YAMLs

## ConfigHub


**Mechanism:** Moves away from "File-based" Git to "Data-based" storage. It stores fully rendered, literal YAML manifests in a structured database (Source of Record) instead of managing complex templates in Git.

**Best For:** Platform Engineering teams managing hundreds of clusters or apps who want to eliminate "Config Sprawl" and template errors.

- **WYSIWYG** (What You See Is What You Get): No more hidden logic. Every manifest is stored in its final, "WET" (Write Every Time) form, making it instantly readable and validatable.

# Recap: GitOps Architecture, Patterns...


- **GitOps Architecture**: 4 principles + internal vs. external reconcilers (single cluster vs. fleet).
- **Progressive Delivery:** strategies, risks, and benefits
- **Trunk vs. Branch:** trunk reduces complexity; combine approaches when needed
- **Repository Strategies:** choose a promotion model that scales
- **State Store:** Git is common, but not the only option—consider **OCI** and **ConfigHub**


**GitOps is based on the four principles.**
But if you want to get the **most out of** it, you also need to understand the other **building block!**

# Demo