Platform Engineering UNIVERSITY

GITOPS FOR PLATFORM ENGINEERING

GitOps in Enterprise – Scaling and Security

MODULE 05

Gitops at Scale – Reference Architecture(s)

Hub & Spoke

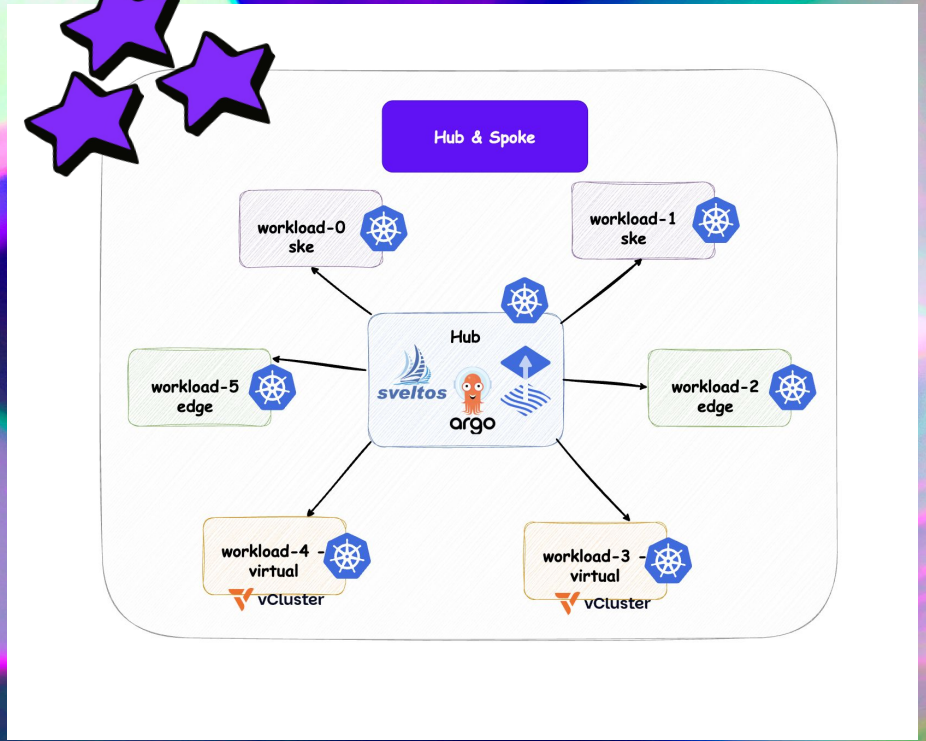
One central instance manages deployments for many remote clusters via their Kubernetes API.

Pros:

- **Centralized Visibility:** A single "Single Pane of Glass" for all clusters and applications.
- **Ease of Management:** SSO, RBAC, and repository credentials are configured only once.
- **Efficiency:** Perfect for using **ApplicationSets** with the `cluster generator` or **ClusterProfiles** to push add-ons across the fleet.

Cons:

- **Blast Radius:** A failure in the hub affects all connected clusters.
- **High Security Risk:** Central cluster stores admin credentials (kubeconfigs) for all target clusters.
- **Networking:** Requires the Hub to have direct network access to the target clusters' APIs.



Instance per Cluster

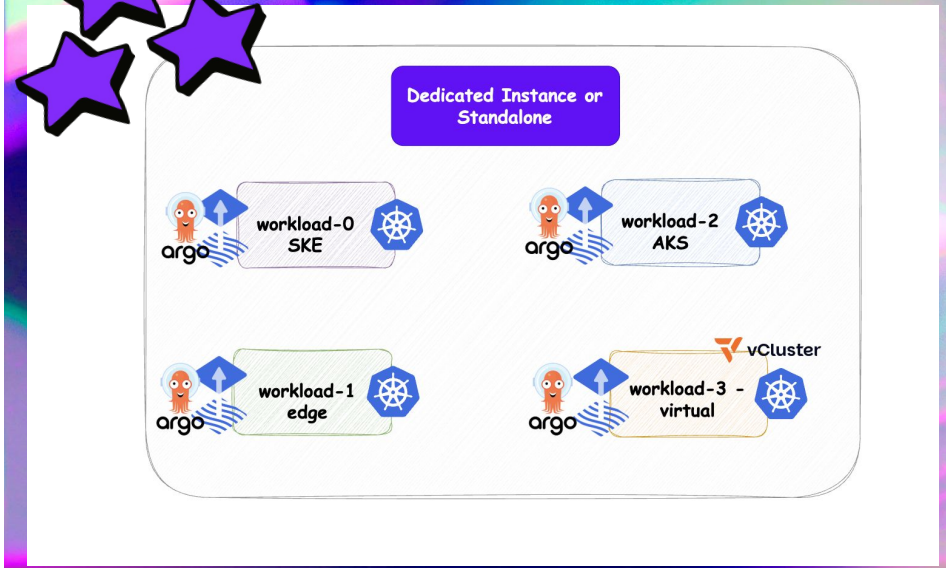
Instance is co-located with the workloads it manages.

Pros:

- **High Reliability:** Clusters are fully autonomous. An outage in Cluster A has no impact on Cluster B.
- **Strict Isolation:** No cluster-wide admin credentials stored centrally; security boundaries match cluster boundaries.
- **Edge/Air-Gapped Ready:** Best for Edge deployments or clusters behind strict firewalls.

Cons:

- **Management Overhead:** Every instance must be individually patched, updated, and configured.
- **Fragmented Visibility:** Developers must switch between multiple URLs/clusters to check application health.
- **Consistency Risk:** Harder to ensure all Argo/ instances stay synchronized in their configuration.



Hub & Spoke x Instance per Cluster

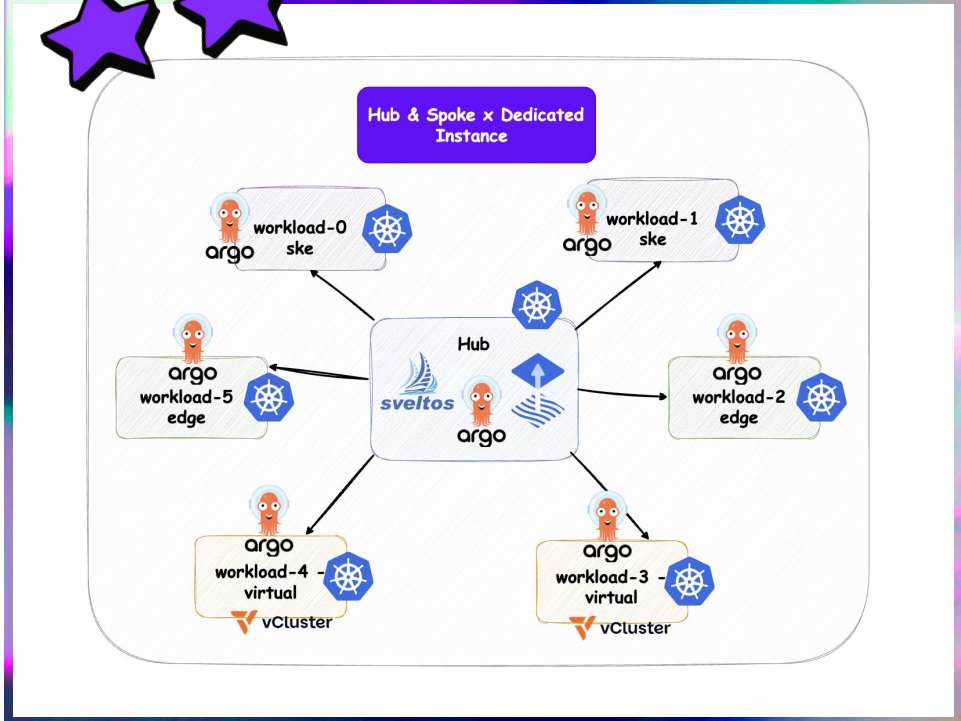
A central **Hub** orchestrates the fleet, while each target cluster runs its own **Dedicated Instance** of to execute reconciliations locally.

Pros

- **Decentralized Reliability:** If the central Hub goes down, local instances continue to sync and maintain the desired state independently.
- **Scalability:** Heavy lifting (manifest rendering and API watching) is distributed across clusters, preventing the central Hub from becoming a performance bottleneck.

Cons

- **Resource Overhead:** Running a full set of controllers in every single cluster significantly increases total CPU/RAM consumption.
- **Management Burden:** Operators must maintain, patch, and update two layers (the central Hub and all dedicated local instances) simultaneously.



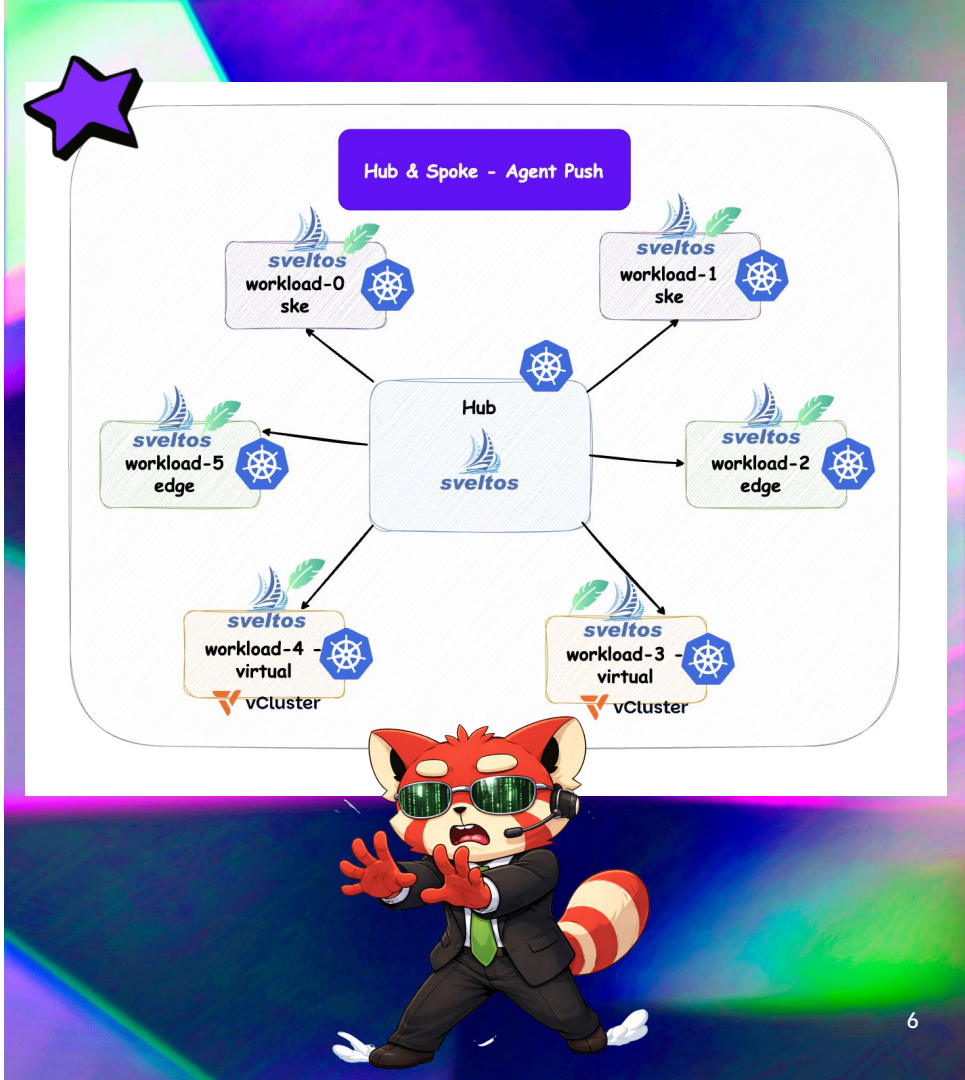
Hub & Spoke Agent based – Push

The Management Cluster acts as the active driver, directly "pushing" configurations to the target API servers.

Registration: A `SveltosCluster` resource is created and labeled in the management cluster.
Security: The managed cluster's **Kubeconfig** is stored as a **Secret** within the management cluster.
Deployment Flow:

- Sveltos fetches the Kubeconfig Secret.
- It creates a Kubernetes client to access the managed cluster's API server.
- Resources are deployed **directly** from the hub to the spoke.

Requirement: The management cluster must have **direct, immediate access** to every managed cluster's API endpoint (Inbound connectivity).



Hub & Spoke Agent based – Pull

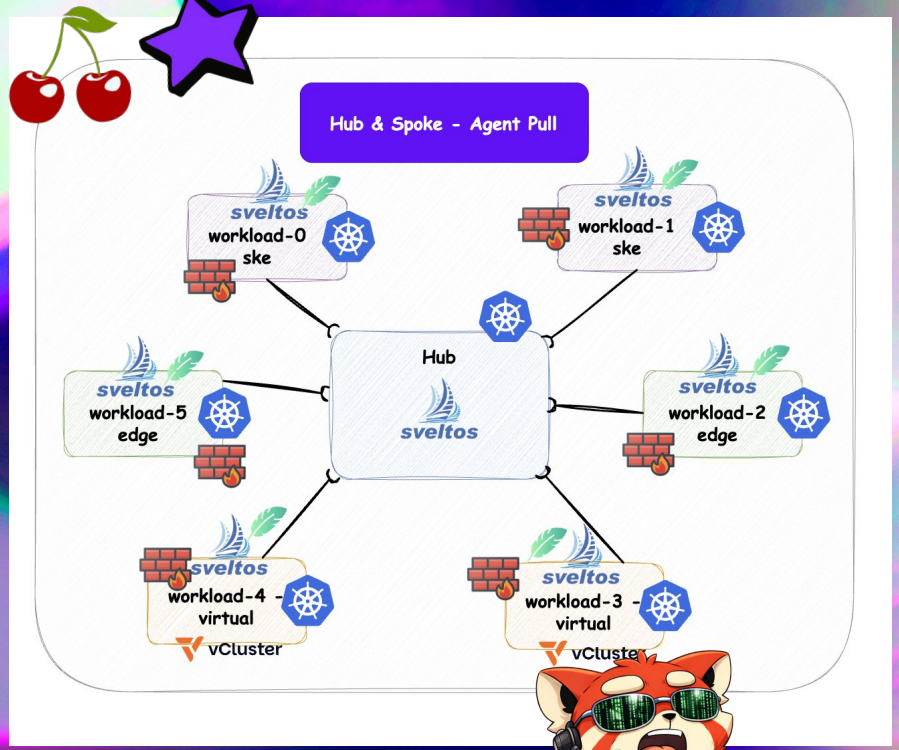
The Managed Cluster "pulls" its own desired state from the management hub, allowing for operation behind firewalls.

Registration: A `SveltosCluster` resource is created in the hub and explicitly marked for **Pull Mode**.
Security: A highly-permissioned **ServiceAccount** is created within the managed cluster for a local agent.

Deployment Flow:

- Sveltos prepares and stores resources in the management cluster.
- The **Agent** (running on the spoke) fetches the resources from the hub.
- The Agent applies the resources locally.

Advantage: Ideal for clusters in **private networks**; as long as the agent can reach the management hub (Outbound).




TOO MANY CHOICES
TOO MANY OPTIONS

Logical Grouping

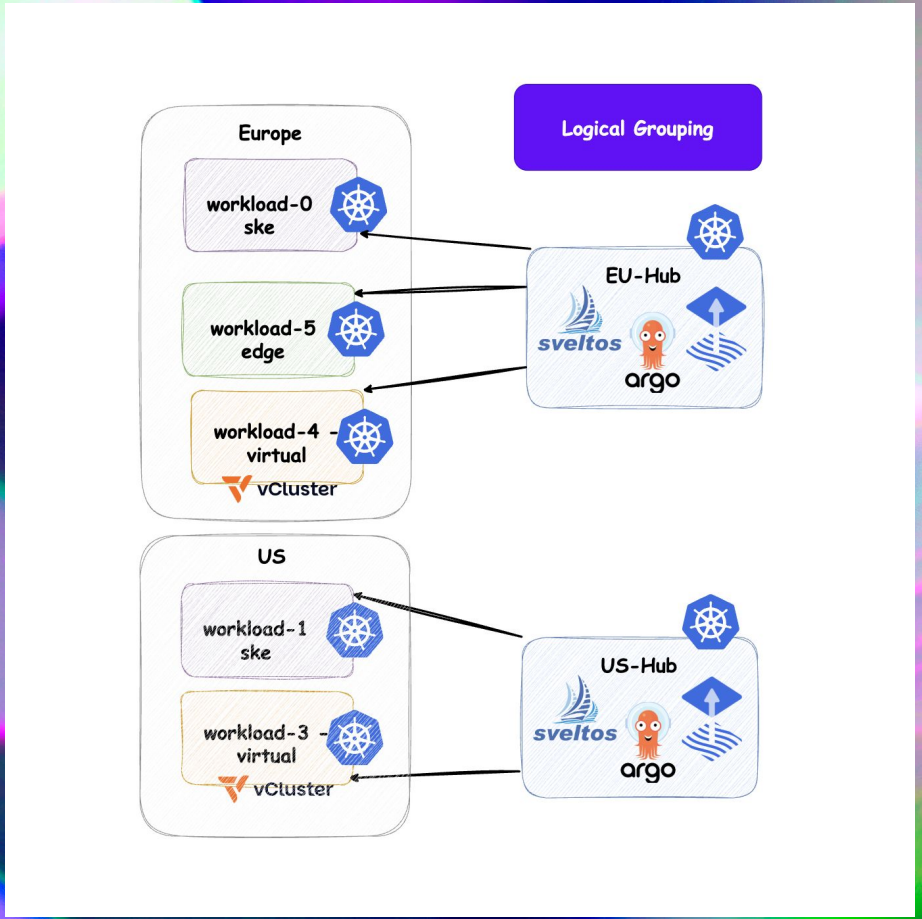
Separate instances for "Production" vs. "Non-Prod," or by geolocation units (e.g., "EU" vs. "US").

Pros:

- **Controlled Blast Radius:** Production issues are isolated from Dev/Staging.
- **Better Scaling:** Distributes the load (Repo Server/Controller) across multiple instances.
- **Reduced Friction:** Provides a single view for specific teams while maintaining critical boundaries.

Cons:

- **Middle Security Risk:** Central cluster(s) stores admin credentials (kubeconfigs) for all target clusters.
- **Fragmented Visibility:** No "Global Single Pane of Glass"; users must switch between different URLs/contexts depending on the group.
- **Resource Inefficiency:** Higher total CPU/RAM consumption as core controllers (API, Repo-server) are duplicated for every logical group.



Sharding

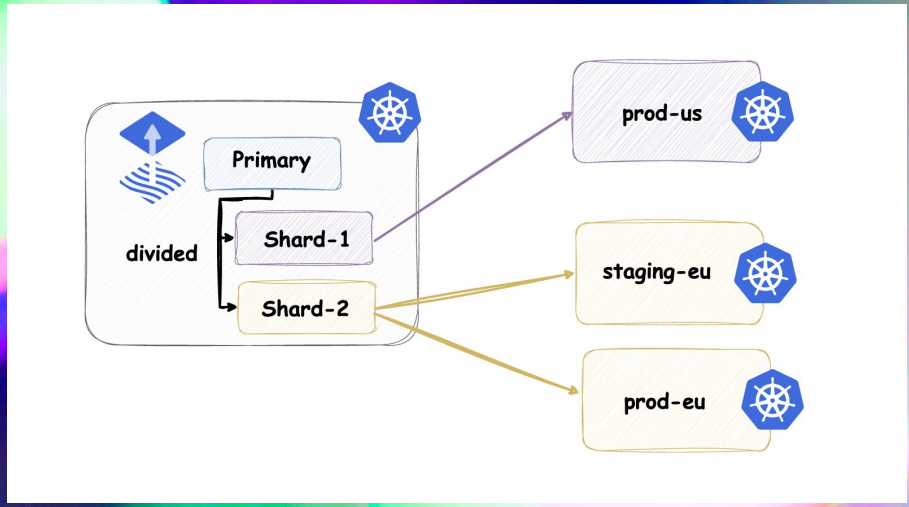
Sharding is a **horizontal scaling strategy** used to manage tens of thousands of applications by distributing the controller workload.

Pros:

- **Massive Scalability:** Distributes CPU/RAM load across multiple controller pods to handle huge fleets.
- **Fault Isolation:** Performance issues in one shard (e.g., a massive Git repo) do not affect other shards.
- **Tenant Dedication:** Allows assigning specific shards to high-traffic tenants or critical environments.
- s high activity doesn't interfere with others.

Cons:

- **Management Complexity:** Requires manual setup of directories and Kustomize patches for every shard.
- **Resource Overhead:** Higher total footprint as core controllers are duplicated for each shard.
- **Labeling Discipline:** All linked resources (Repo, Kustomization, Chart) must have consistent shard labels or they will be ignored.



Sharding

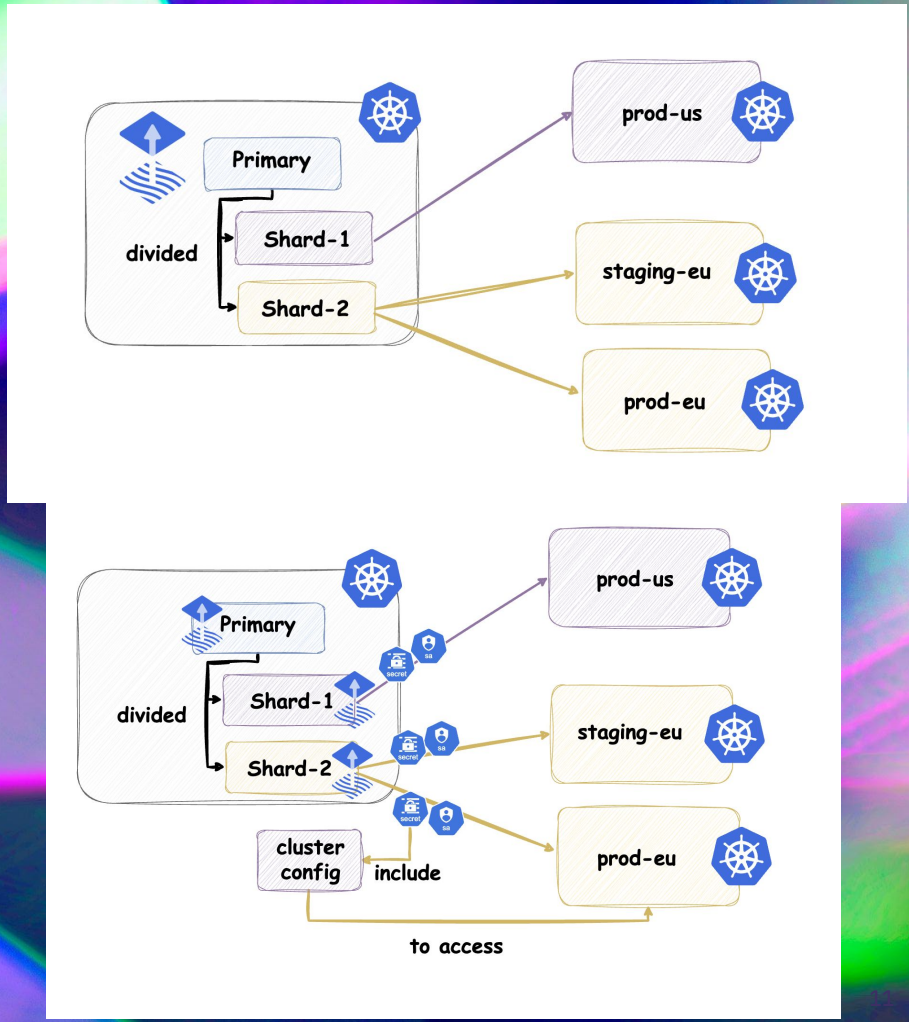
Sharding is a **horizontal scaling strategy** used to manage tens of thousands of applications by distributing the controller workload.

Pros:

- **Massive Scalability:** Distributes CPU/RAM load across multiple controller pods to handle huge fleets.
- **Fault Isolation:** Performance issues in one shard (e.g., a massive Git repo) do not affect other shards.
- **Tenant Dedication:** Allows assigning specific shards to high-traffic tenants or critical environments.
- s high activity doesn't interfere with others.

Cons:

- **Management Complexity:** Requires manual setup of directories and Kustomize patches for every shard.
- **Resource Overhead:** Higher total footprint as core controllers are duplicated for each shard.
- **Labeling Discipline:** All linked resources (Repo, Kustomization, Chart) must have consistent shard labels or they will be ignored.



GitOps and Security – Secrets Management and Compliance

Compliance – Kyverno

Kyverno is a Kubernetes-native policy engine that manages **Compliance** and **Best Practices** using declarative resources (CRDs). It eliminates the need for complex programming languages (like Rego) by using familiar YAML-based policies.

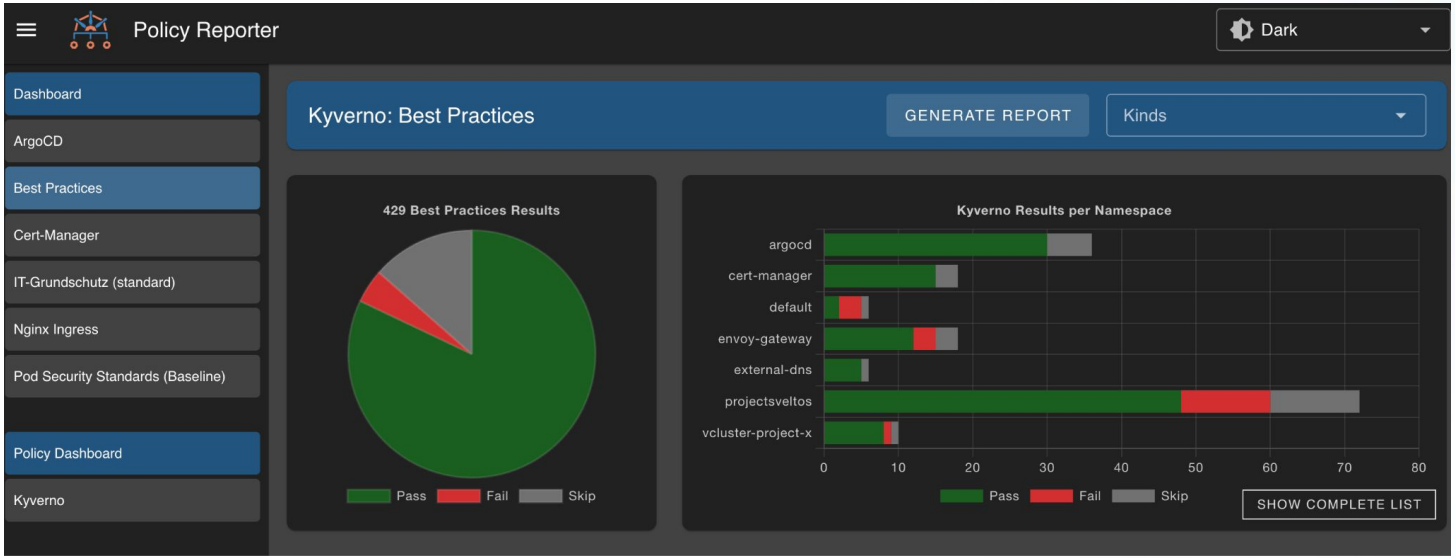
- **Declarative Compliance:** Policies are stored in Git, versioned, and synced to the cluster just like any other application or infrastructure component.
- **Shift-Left Security:** Validates manifests during the CI/CD pipeline or at the Pull Request stage before they ever reach the cluster.
- **Automated Remediation:** Beyond just "blocking" bad configs, it can **mutate** (fix) or **generate** (create) missing resources (e.g., auto-injecting NetworkPolicies or Sidecars).

Key Capabilities

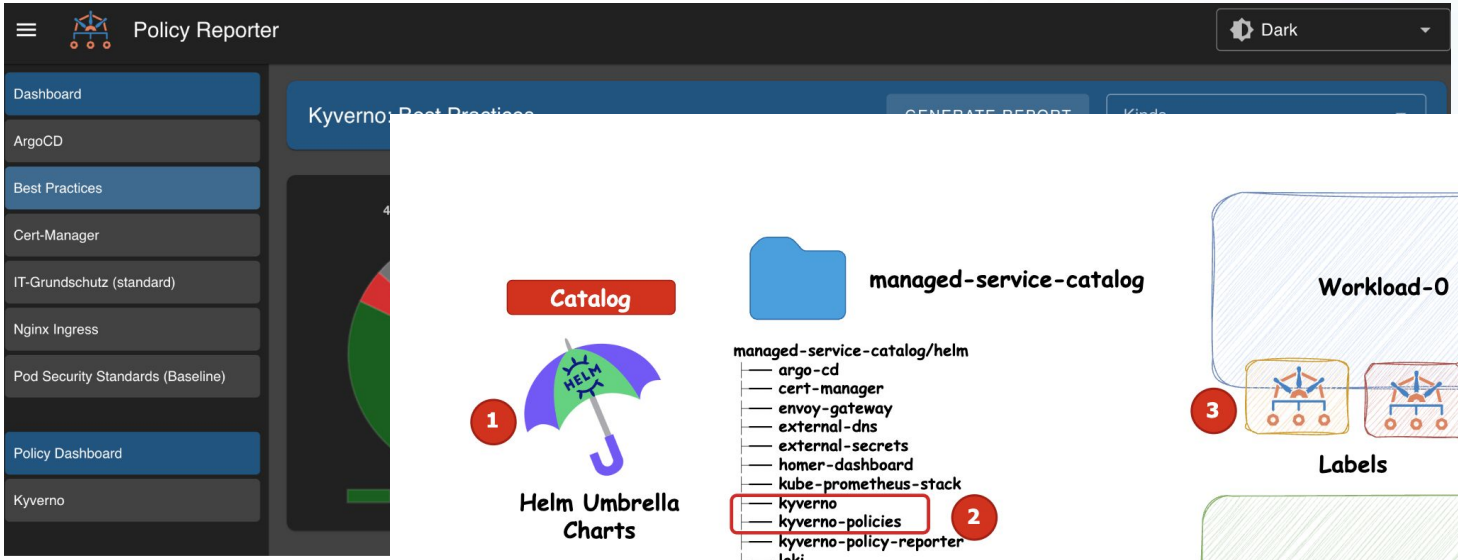
- **Validation:** Enforces best practices (e.g., "Disallow Root User," "Require Resource Limits") by blocking non-compliant deployments.
- **Mutation:** Automatically patches incoming resources to meet organizational standards (e.g., adding mandatory labels).
- **Generation:** Creates new resources based on triggers (e.g., when a Namespace is created, generate a default Deny-All NetworkPolicy).
- **Reporting:** Provides cluster-wide compliance reports to identify existing resources that violate new security policies.

Kyverno is the 'GitOps-native Guardrail': If Git defines what *should* run, Kyverno ensures only what *is allowed* can run

Guardrails that's Scales!



Guardrails that's Scales! (II)



Secrets Management with GitOps

Sealed Secrets Operator 

Uses **asymmetric encryption**. You encrypt a secret locally using a public key; only the controller in the cluster holds the private key to decrypt it

Pros:

- **Git-Centric:** The encrypted "SealedSecret" is safe to store in Git.
- **Simple Setup:** No external infrastructure (like a Vault) required.

Cons:

- **Key Management:** If you lose the cluster's private key, you cannot decrypt your Git-stored secrets anymore.
- **Rotation:** Manual re-encryption is usually required when secrets change.

External Secrets Operator (ESO) 

Acts as an **API bridge**. It fetches the actual secret values from an external provider (AWS Secret Manager, HashiCorp Vault, Azure Key Vault) at runtime.

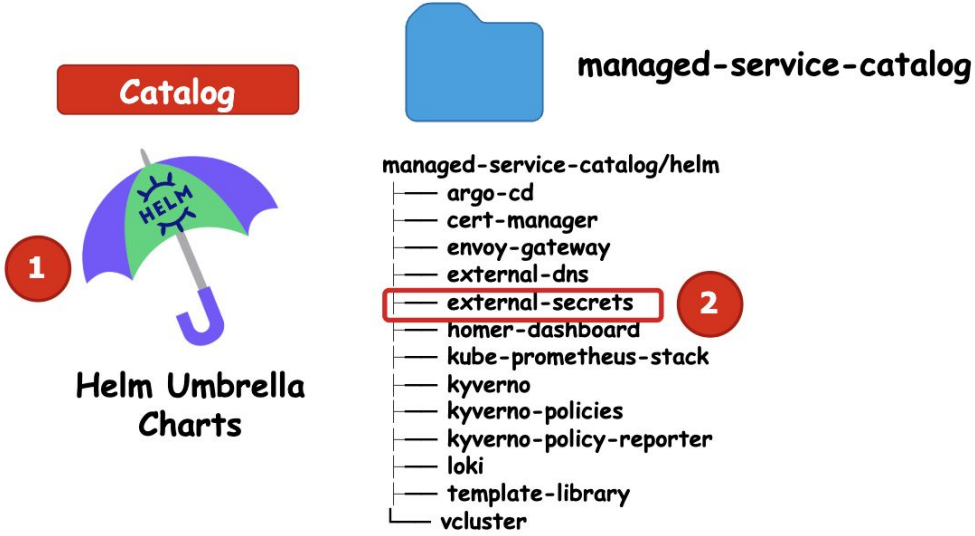
Pros:

- **Security Best Practices:** Sensitive data never even enters your GitOps repository—only a reference (ExternalSecret) is stored.
- **Auto-Sync:** Changes in the external provider are automatically synced to the cluster.

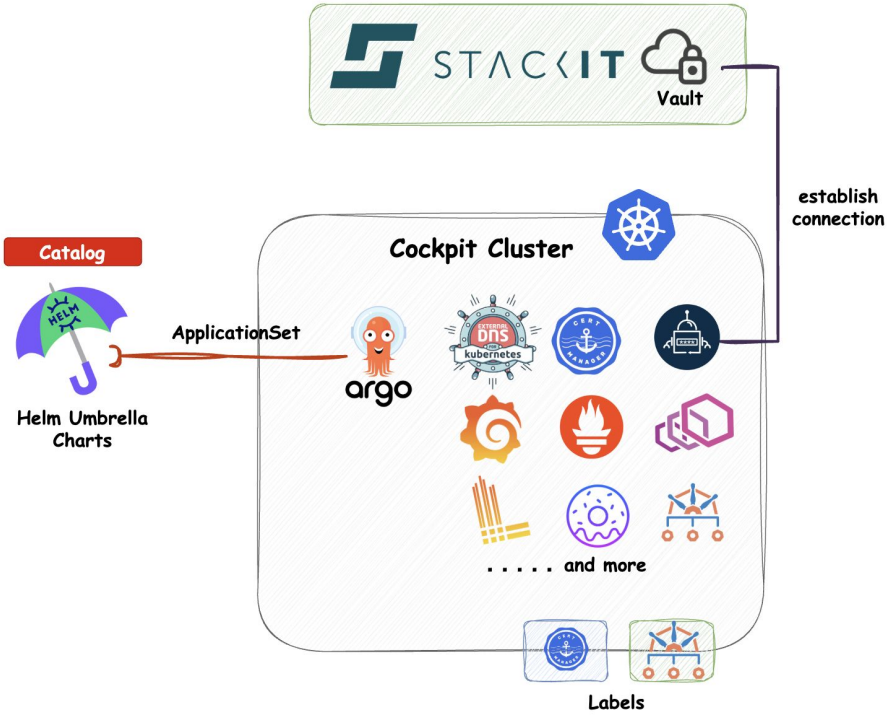
Cons:

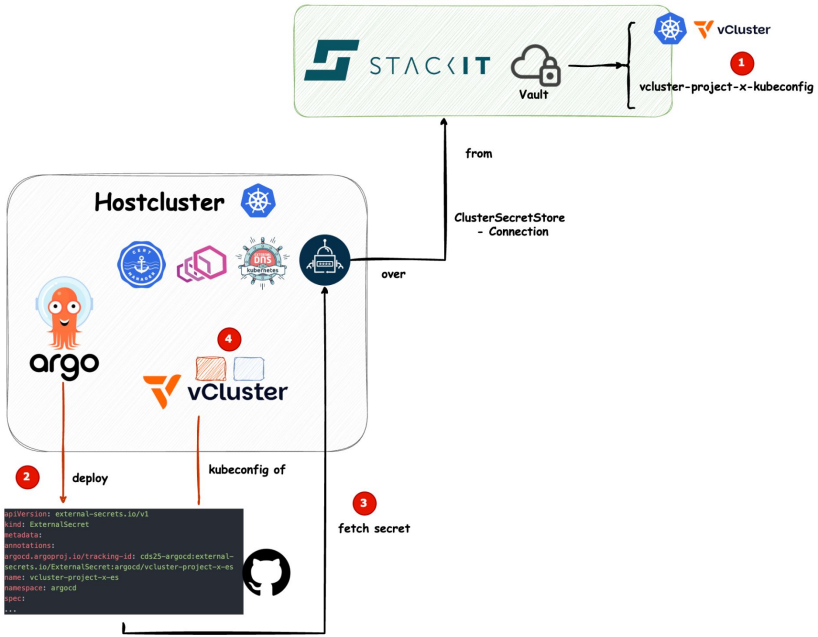
- **Dependency:** Requires an external Secret Store (Vault/Cloud Provider) to be available and managed.
- **Complexity:** More configuration needed (SecretStores, authentication, IAM roles).

External Secrets Operator (ESO) – Kubara



External Secrets Operator (ESO) – Kubara





External Secrets Operator (ESO) - Kubara


```yaml
apiVersion: external-secrets.io/v1
kind: ExternalSecret
metadata:
  labels:
    app.kubernetes.io/part-of: external-dns
  name: external-dns-webhook-es
  namespace: external-dns
spec:
  data:
  - remoteRef:
      conversionStrategy: Default
      decodingStrategy: None
      key: dns_zone_admin
      metadataPolicy: None
      property: project_id
    secretKey: PROJECT_ID
  - remoteRef:
      conversionStrategy: Default
      decodingStrategy: None
      key: dns_zone_admin
      metadataPolicy: None
      property: sa_key_json
    secretKey: SA_KEY_JSON
```

External Secrets Operator (ESO) - Kubara


```yaml
apiVersion: v1
data:
  PROJECT_ID: Mzg4...
  SA_KEY_JSON: eyJhY3...
kind: Secret
metadata:
  labels:
    app.kubernetes.io/part-of: external-dns
    reconcile.external-secrets.io/managed: "true"
  name: external-dns-webhook
  namespace: external-dns
type: Opaque
```

Culture Shift

GitOps in – GitOps out

Why GitOps Requires a Cultural Shift

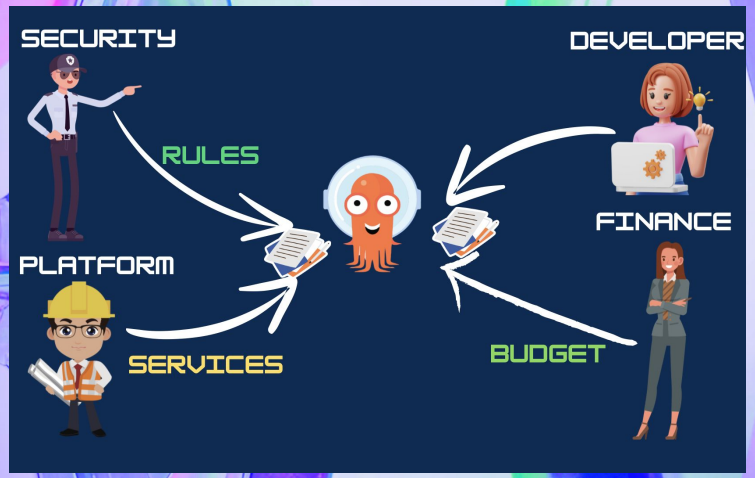
- **From Manual to Declarative:** Teams must stop "fixing" things directly in the cluster (`kubectl edit`) and trust the automated process via Git.
- **Shared Responsibility:** Operations and Development merge closer; the Git repository becomes the "Single Source of Truth" for everyone, requiring high levels of collaboration.
- **Transparency & Accountability:** Every change is visible through Pull Requests, which requires a culture that embraces open peer reviews and auditability.

Why is it hard to internalize at first?

- **Relinquishing Control:** The hardest part is "letting go" of direct cluster access; developers and admins often feel restricted.
- **Steep Learning Curve:** Teams must master trunk based workflows for progressive delivery.
- **The "Lag" Factor:** Changes are no longer instant (seconds via CLI) but take time to flow through Git and the reconciler, which requires a shift in mindset regarding feedback loops.
- **Strict Discipline:** GitOps punishes "quick and dirty" fixes

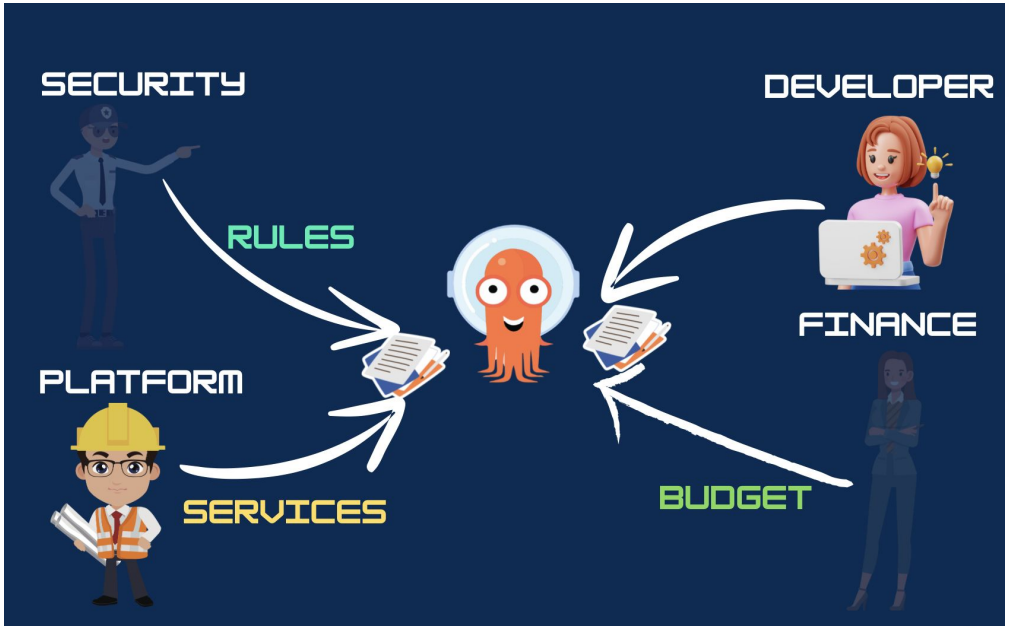
GitOps is 20% Tooling and 80% Discipline. The challenge isn't the technology, but the habit of never touching the cluster manually again

Everything as Code

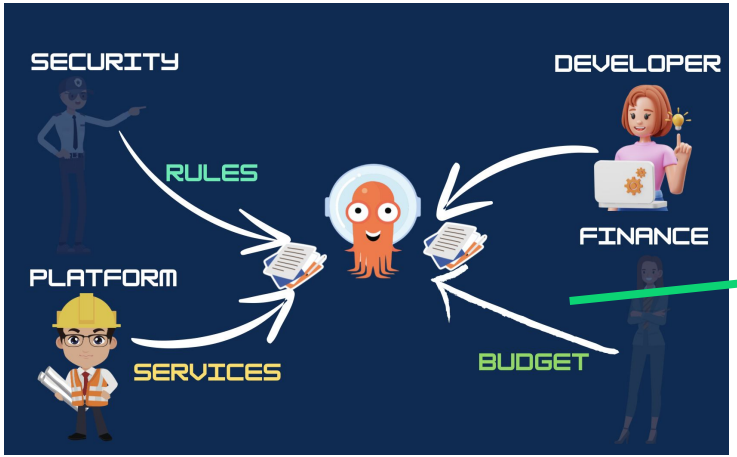


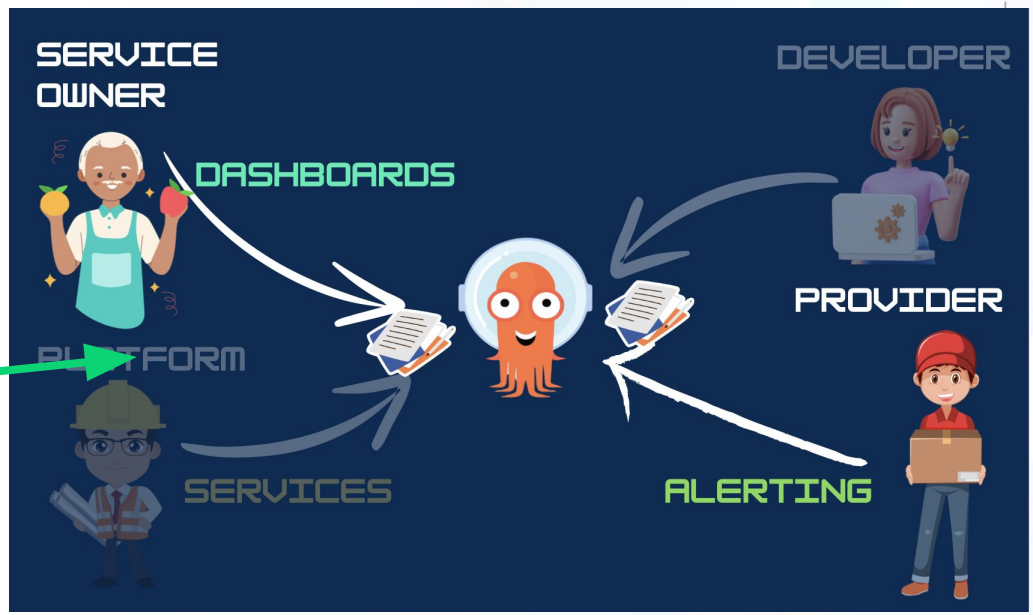
Our vision: A unified environment where **Security** and **FinOps** define policies or budgets as code. Whether via Git or an interface, every rule is declarative, versioned, and automatically enforced across clusters.

Everything as Code (II)

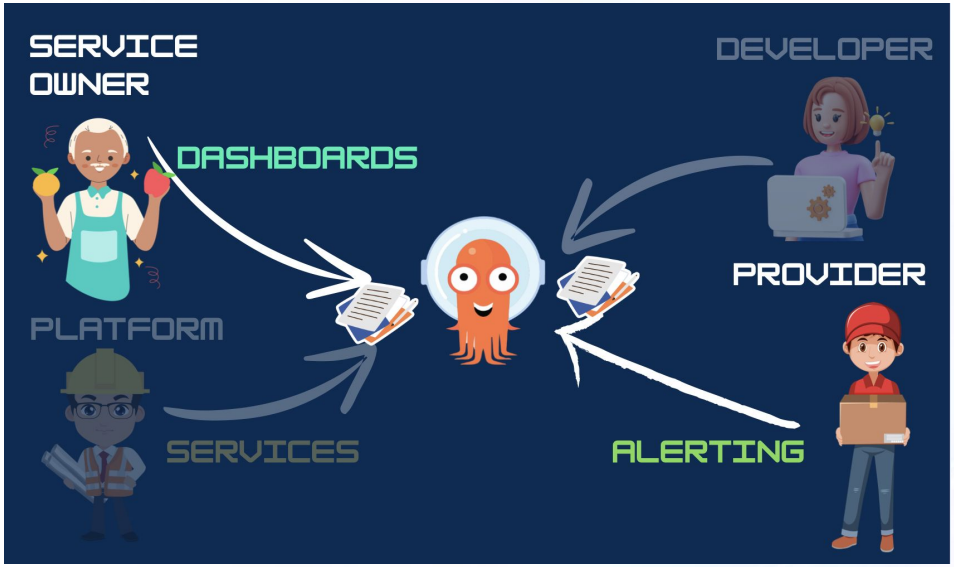


Everything as Code (III)





Everything as Code (IV)



Moving beyond '**ClickOps**' forced a mindset shift: **Service Owners** embraced **Dashboards-as-Code**. By taking ownership through **YAML** and **Helm**, they stopped being passive users and became active creators. While the start was difficult, it ultimately transformed infrastructure into a **shared, engaging responsibility**.

Recap: GitOps Architecture, Patterns...

- **You** can choose between **Hub & Spoke**, **Standalone or ...** topologies. Every environment is different, so feel free to combine these patterns to get the best of both worlds. The most popular are Hub & Spoke and Standalone.
- **Kubernetes is not secure by default.** As a platform provider, you must enforce critical policies like *"RunAsNonRoot"* through **guardrails** to keep the cluster safe.
- **Secrets management** should be as **easy as the rest of your code**. Use Sealed Secrets for a simple Git-native setup, or switch to **External Secrets Operator (ESO)** if you already have a **Vault**.
- The tools are nice, but never forget that **developers are the end-users**. No matter how you build it, the focus must stay on **providing a great self-service experience**. Culture matter.

Demo