

Argo CD A Visual Guide

Sample Book — Chapters 1 to 2

Essential GitOps and Kubernetes Delivery for New Systems Engineers

Reiner Braun

Introduction

This book is designed for readers who are taking their first steps with Argo CD, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

How to Use This Book

Read the chapters in order the first time through. Each chapter builds on concepts introduced earlier, so this sequence develops the necessary background step by step instead of asking you to infer missing foundations.

Before reading a chapter, review its **What You'll Be Able to Do** section. Those outcomes identify the practical understanding the chapter is intended to build and help you distinguish core material from supporting detail.

Use the diagrams to clarify **relationships, processes, boundaries, and structures** that are difficult to hold in memory from prose alone. The diagrams support the explanations; they do not replace the surrounding text.

Run the commands and hands-on examples yourself whenever a suitable learning environment is available. Reading establishes the model, but direct observation of synchronization, status, diffs, and Kubernetes resources makes that model more durable.

At the end of each chapter, use the **Chapter Summary** and **Review Questions** to confirm that you can explain the fundamentals in your own words. Return to the relevant section when an answer depends on memorized wording rather than a clear mental model.

Advanced-topic overviews do not need to be memorized at this stage. Treat them as signposts: understand why the topic exists, what problem it addresses, and where it fits so that you can recognize when deeper study is needed later.

Table of Contents

Introduction

How to Use This Book

Scope and Learning Goal

Chapter 1: What Argo CD Solves

- What You'll Be Able to Do
- Start with the operating model
- Why command-driven deployment becomes difficult
 - Inconsistent deployment steps
 - Configuration drift
 - Weak auditability
 - Environment differences
- Declarative configuration
- GitOps is more than storing YAML in Git
- What "Git is the source of truth" means
- Pull-based reconciliation
 - Direct deployment from CI
 - GitOps deployment with Argo CD
- Continuous integration and continuous delivery are different jobs
- A successful commit is not a successful application
- What Argo CD normally does not do
- Benefits and trade-offs
- A practical comparison
- Misconceptions that cause early mistakes
- Chapter Summary
- Review Questions

Chapter 2: The Kubernetes and Git Knowledge Argo CD Depends On

- What You'll Be Able to Do
- The minimum foundation
- Anatomy of a Kubernetes resource
- Desired state and live state
- The Kubernetes controller pattern
- Namespaces, labels, and annotations
- Owner references and finalizers
- Custom Resource Definitions
- The API server is the center of cluster interaction
- Essential kubectl commands
 - Confirm the current target
 - List resources
 - Inspect details and events
 - Preview declarative differences
 - Apply a manifest
 - Read controller or workload logs
- Git concepts needed for Argo CD
- Container registries, tags, and digests
- Helm and Kustomize in one sentence each
- Follow one Deployment from file to Pods
- Readiness checklist
- Mistakes that distort the mental model
- Chapter Summary
- Review Questions

Scope and Learning Goal

Argo CD continuously compares a desired Kubernetes state with the live state of a cluster and helps make the live state match the desired state. That sentence is the foundation for everything in this guide.

The chapters that follow build the knowledge needed to understand that process, install a safe learning environment, deploy a first application, interpret status and drift, organize configuration for teams and multiple environments, control access and deployment order, integrate continuous integration, troubleshoot failures, harden the platform, and plan recovery and upgrades.

The examples use Argo CD **v3.5.3** so that installation commands are reproducible. The core concepts are not tied to one release, but exact flags and fields should always be checked when a different version is used.

PART I: BUILD THE CORRECT MENTAL MODEL

Chapter 1: What Argo CD Solves

What You'll Be Able to Do

By the end of this chapter, you will be able to:

- Explain the operational problem Argo CD solves and why declarative desired state matters.
- Distinguish the responsibilities of Git, Kubernetes, Argo CD, and continuous integration.
- Recognize the benefits, trade-offs, and common misconceptions of a GitOps workflow.

Start with the operating model

Argo CD is a Kubernetes continuous delivery controller that turns version-controlled desired state into repeatable cluster changes. It does not replace Kubernetes, Git, or continuous integration. Its main job is to observe, compare, report, and reconcile.

A conventional deployment process often depends on a person or CI job running a sequence of commands:

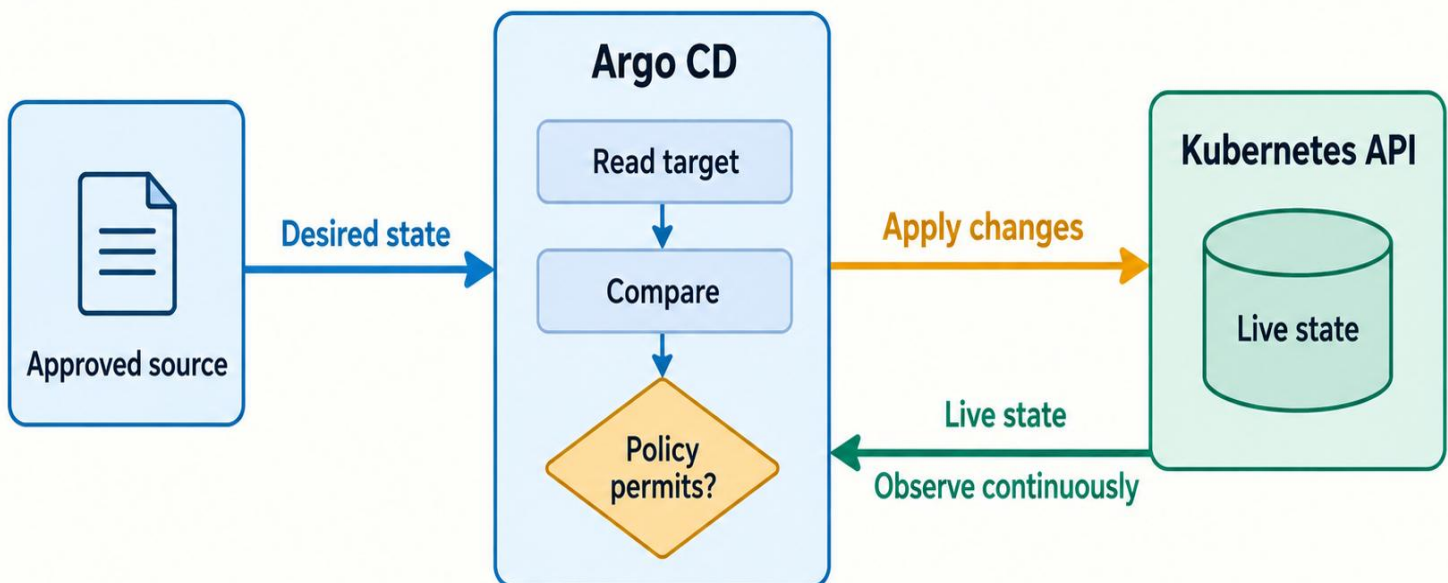
```
kubectl apply -f namespace.yaml
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

This can work, but the result depends on more than the files. It also depends on which cluster context was selected, which commands ran, which flags were used, whether someone changed the cluster afterward, and whether the operator remembered every step. Over time, environments that were intended to be identical can become different.

Argo CD changes the center of the workflow. Instead of treating deployment commands as the lasting record, it treats a version-controlled description of the intended system as the lasting record. The controller then compares that description with Kubernetes.

Chapter 1: What Argo CD Solves

1-1 Argo CD Operating Model



Continuous control loop—not a one-time push

Figure 1-1: Argo CD operating model

Why command-driven deployment becomes difficult

Systems engineers repeatedly encounter four related problems.

Inconsistent deployment steps

An **imperative operation** tells a system what action to perform, such as “set the replica count to three” or “apply this file now.” Imperative commands are useful for investigation and one-time administration, but a sequence of commands can become an undocumented deployment procedure.

Two engineers may deploy the same application with slightly different flags. A CI job may have credentials that allow it to change resources not mentioned in the repository. A failed step may leave only part of the intended release in place.

Configuration drift

Configuration drift is the difference that develops when a live environment no longer matches its intended configuration. Drift can be caused by emergency edits, debugging changes, another controller, a manual scaling operation, or a deployment tool that updated only part of the system.

Without a continuous comparison, drift is usually discovered after it causes a problem.

Weak auditability

A shell command in a terminal may never become part of a durable review history. Even when a CI system records its logs, it may be difficult to answer basic questions later:

- Which exact desired configuration was approved?
- Who reviewed the change?
- Which environments received it?
- What changed between the previous and current release?
- How should the change be reversed?

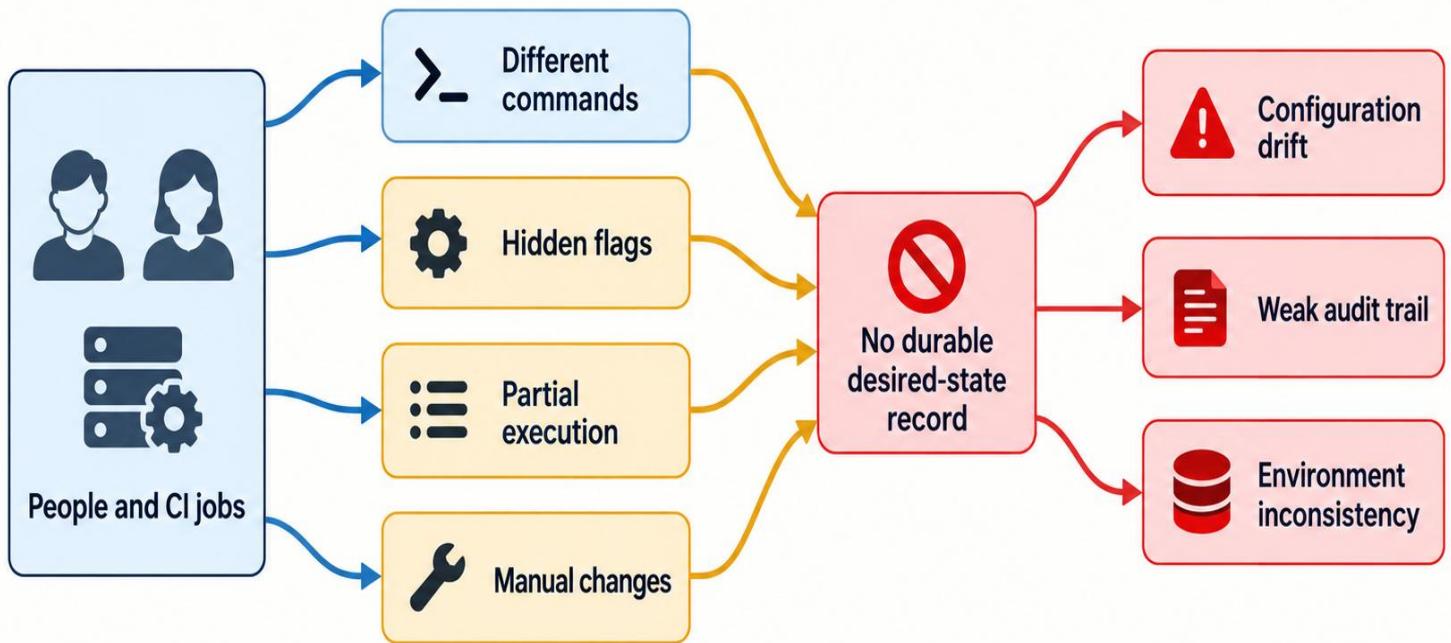
Git provides a commit history and review workflow, but only when the actual deployment state is represented there.

Environment differences

Development, staging, and production commonly need different replica counts, hostnames, resource limits, or feature settings. When those differences are handled through undocumented command-line flags, environments become difficult to compare. When they are represented as reviewable configuration, the differences are explicit.

Chapter 1: What Argo CD Solves

1-2 Why Command-Driven Deployment Degrades



002

Figure 1-2: Why command-driven deployment degrades over time

Declarative configuration

Declarative configuration describes the state that should exist rather than the commands required to create it.

For example, this fragment declares that a Deployment should have three replicas:

```
spec:
  replicas: 3
```

It does not say whether Kubernetes should create three new Pods, remove two old Pods, or do nothing. The Deployment controller examines the current state and chooses the actions needed to reach the declared state.

This is similar to setting a thermostat to 22°C. You declare the desired temperature. You do not continuously issue commands such as “run the heater for 30 seconds.” The thermostat measures the actual temperature and reacts to the difference.

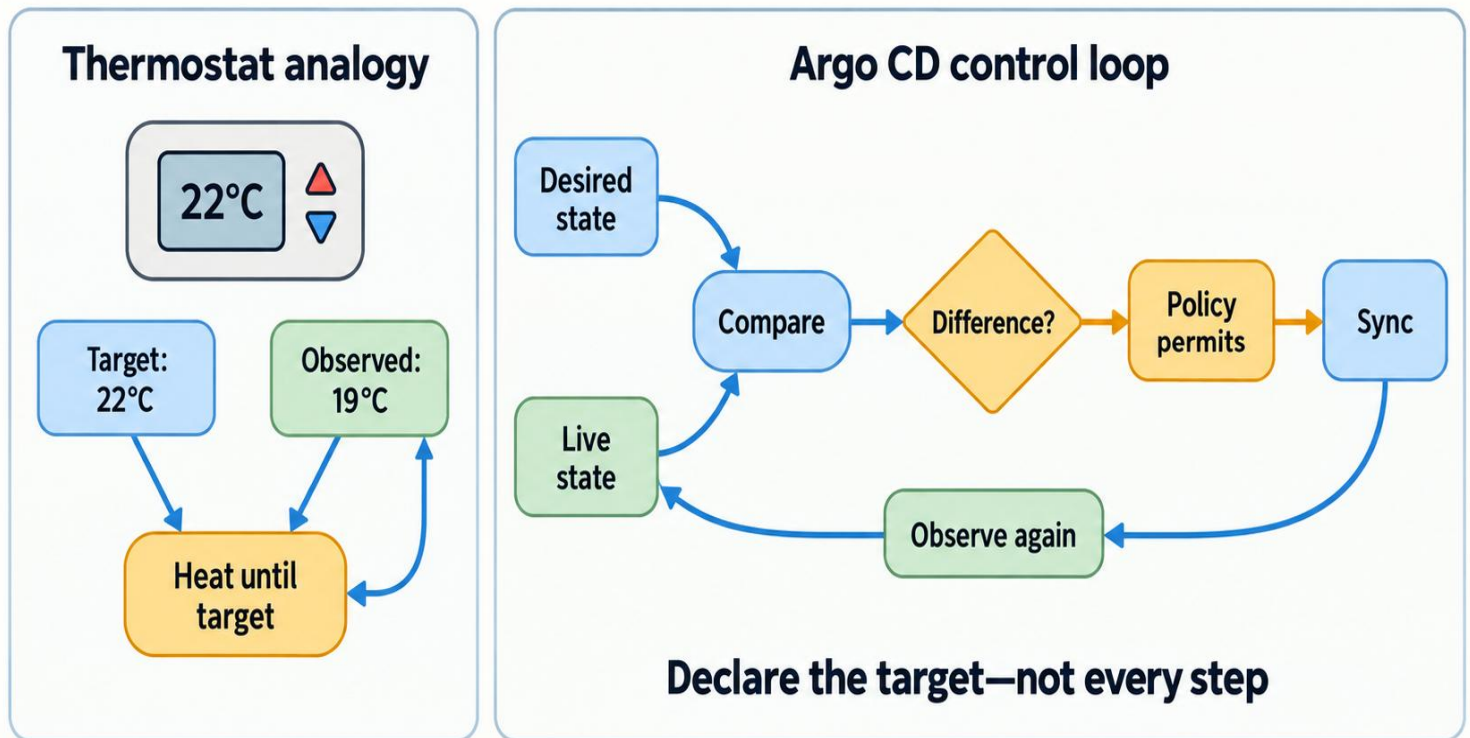
Argo CD applies the same style of control at the application level:

1. Read the desired manifests from a configured source.
2. Read the live resources from Kubernetes.
3. Compare the two states.
4. Report whether they match.
5. When permitted, synchronize the cluster toward the desired state.

6. Continue checking because the live state can change again.

Chapter 1: What Argo CD Solves

1-3 Declarative Configuration and Reconciliation



003

Figure 1-3: Declarative configuration and reconciliation loop

The repeated observe-and-correct process is called a **reconciliation loop**.

GitOps is more than storing YAML in Git

GitOps is an operating model in which version-controlled, reviewable desired state is reconciled into a running system by automation. Storing YAML files in Git is necessary in many GitOps workflows, but it is not sufficient by itself.

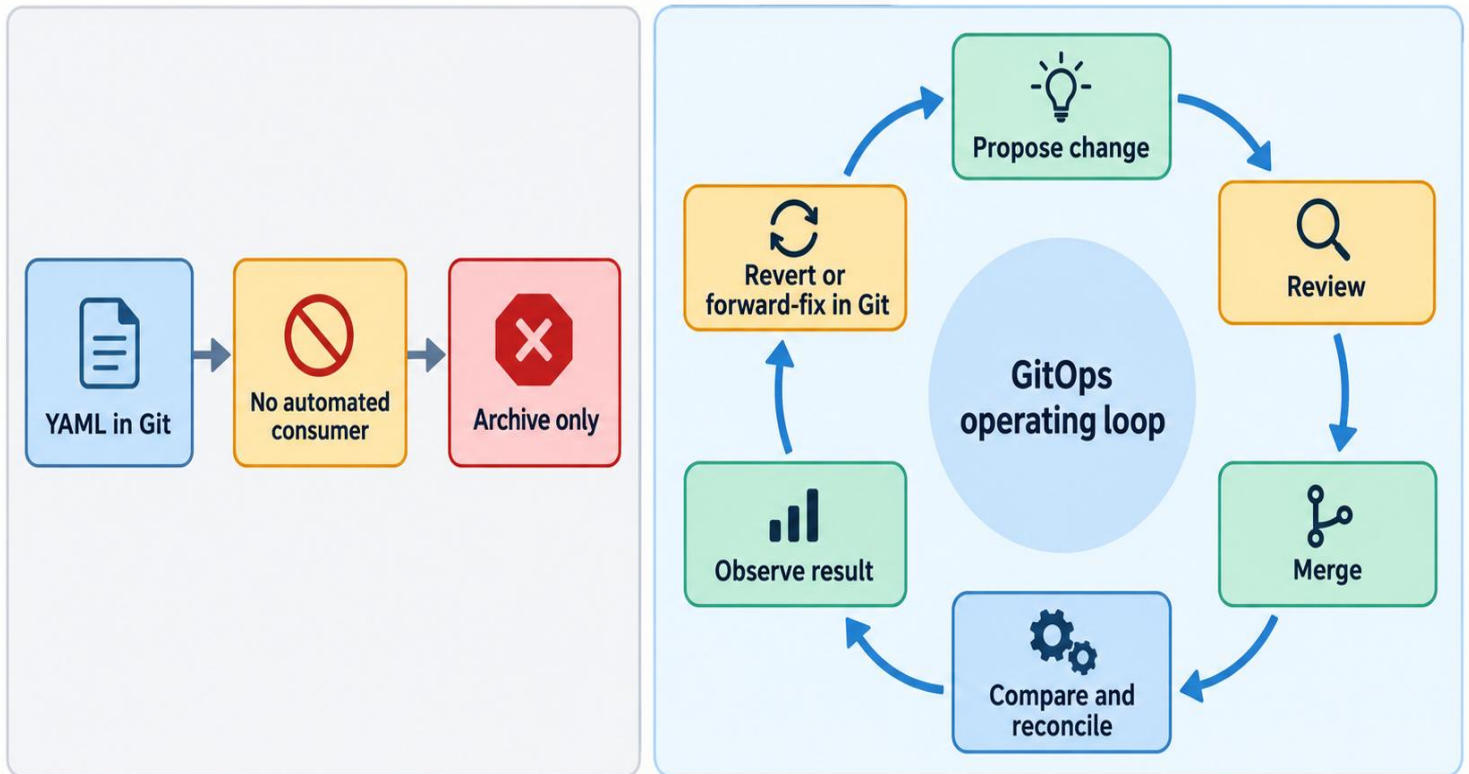
A useful GitOps workflow normally has these characteristics:

- The desired state is declarative.
- Changes are proposed through Git commits and usually reviewed through pull requests.
- The history shows how the desired state changed.
- An automated controller compares desired and live state.

- Recovery is based on changing or reverting the recorded desired state, not on remembering old command sequences.

Chapter 1: What Argo CD Solves

1-4 GitOps Is an Operating Loop



004

Figure 1-4: GitOps as an operating loop, not a YAML archive

A repository that contains old manifests but is not connected to the deployment process is only documentation. A GitOps repository must be part of the control loop.

What “Git is the source of truth” means

The source of truth is the authoritative record used to decide what the system should become. In an Argo CD workflow, that record is usually a Git revision, a Helm chart version, an OCI artifact, or a combination of approved sources.

The phrase does not mean that Git contains every fact about the running system. Kubernetes still owns runtime information such as Pod IP addresses, restart counts, assigned node names, and controller-generated status. Those facts belong in the live system.

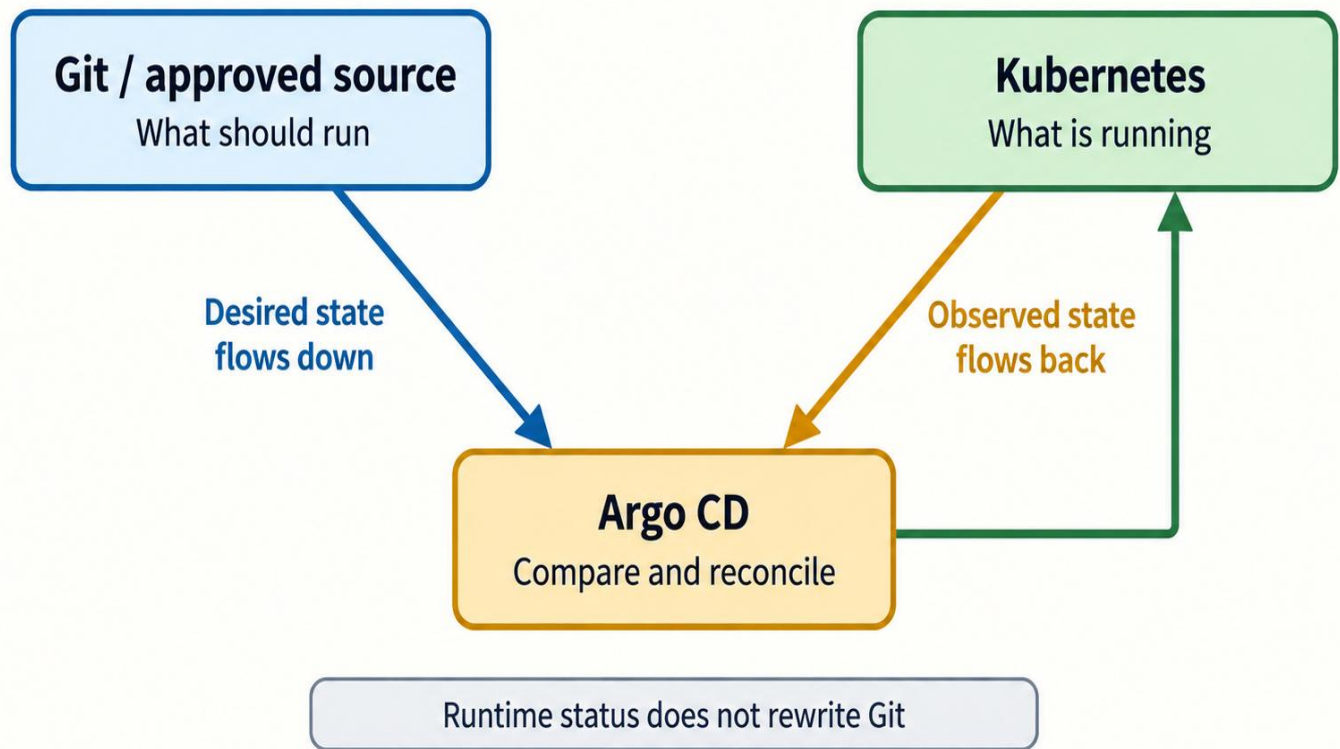
The distinction is:

- Git or another configured source answers, “What should be deployed?”
- Kubernetes answers, “What is running now?”

- Argo CD answers, “How do those states differ, and should I reconcile them?”

Chapter 1: What Argo CD Solves

1-5 Desired State, Observed State, and Responsibility



005

Figure 1-5: Desired state, observed state, and decision responsibility

The cluster is therefore an essential source of **observed state**, but it is not normally the authoritative source of desired application configuration.

Pull-based reconciliation

Argo CD is commonly described as **pull based** because a controller running with cluster access retrieves the desired configuration and applies it, rather than requiring a CI job to push every deployment directly into the Kubernetes API.

Consider two workflows.

Direct deployment from CI

1. A developer merges application code.
2. CI builds and tests a container image.
3. The same CI job runs `kubectl apply` against a cluster.
4. The cluster changes immediately.

The CI system needs cluster credentials. The deployment result is tied to the job execution. If someone later changes the cluster manually, the CI job may not notice.

GitOps deployment with Argo CD

1. A developer merges application code.
2. CI tests the code, builds an image, and publishes it to a registry.
3. A configuration change updates the image reference in Git.
4. The configuration change is reviewed and merged.
5. Argo CD detects that the desired state changed.

6. Argo CD compares the new desired state with the cluster.
 7. Argo CD synchronizes the approved change according to policy.
 8. Argo CD continues checking for later drift.
- The CI system does not need general-purpose credentials for every destination cluster. Its main output is an immutable artifact and a proposed desired-state change.

Chapter 1: What Argo CD Solves

1-6 Direct CI Push vs Pull-Based GitOps

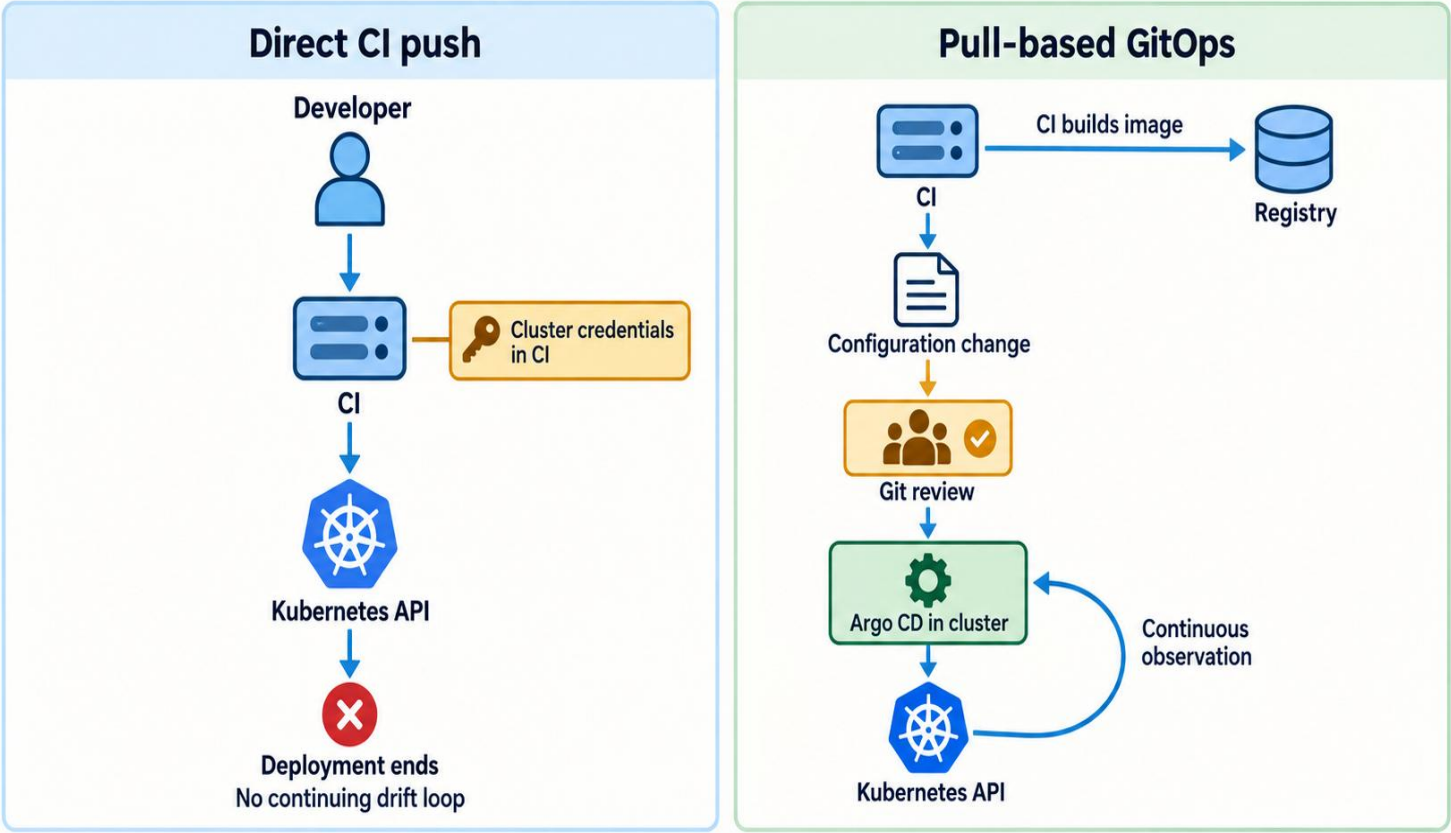


Figure 1-6: Direct CI push versus pull-based GitOps

Continuous integration and continuous delivery are different jobs

Continuous integration, or CI, verifies and packages a software change. Continuous delivery, or CD, makes a verified release available to an environment through a controlled process.

Responsibility	Typical CI role	Typical Argo CD role
Compile or package application code	Yes	No
Run unit and integration tests	Yes	No
Build a container image	Yes	No
Scan and publish the image	Yes	No
Generate Kubernetes manifests from an approved source	Sometimes for validation	Yes
Compare desired and live state	No	Yes
Apply approved Kubernetes resources	Not normally in a GitOps design	Yes
Report sync and resource health	Limited	Yes
Correct unauthorized live drift	No	Yes, when self-healing is enabled

Argo CD can be one component in a larger CI/CD system, but **it is not the entire pipeline.**

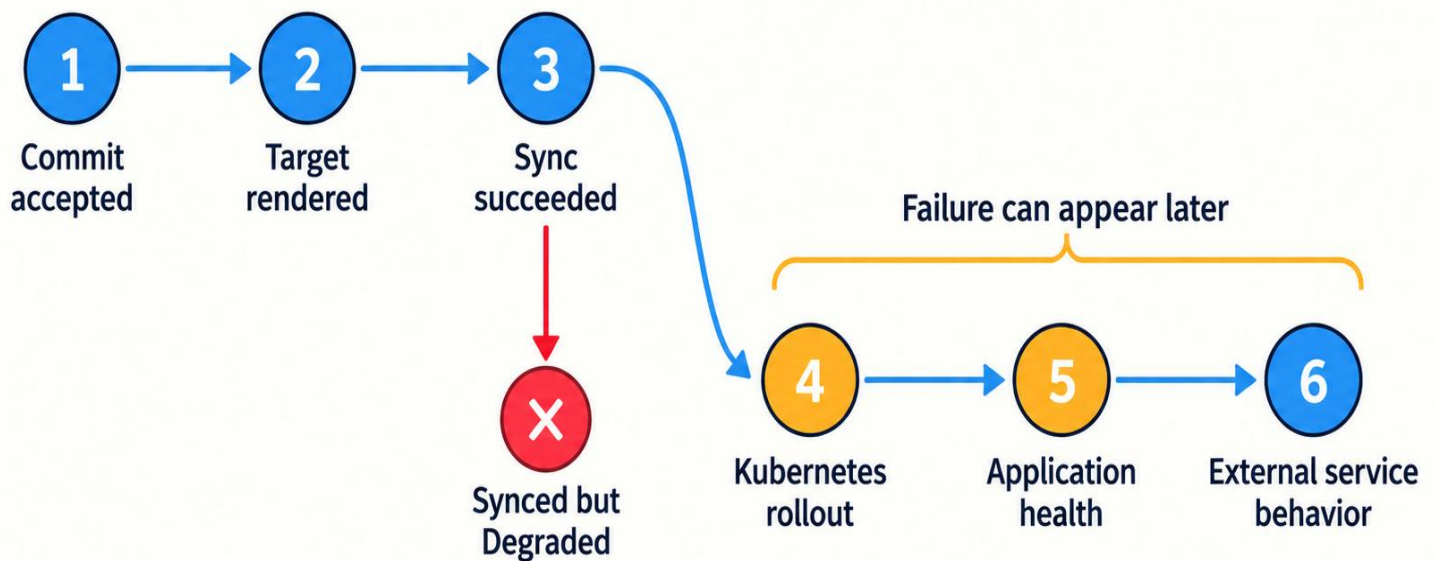
A successful commit is not a successful application

A merged commit proves only that Git accepted the change. A successful synchronization proves that Argo CD completed the requested Kubernetes operations. Neither event alone proves that the application can serve traffic correctly.

For example, all manifests may apply successfully while the application repeatedly crashes because an environment variable is wrong. In that case, the application may be **Synced** but **Degraded**. Sync and health are separate dimensions and must be read separately.

Chapter 1: What Argo CD Solves

1-8 Commit, Sync, and Health Are Separate Milestones



007

Figure 1-7: Commit, sync, and health are separate milestones

What Argo CD normally does not do

Argo CD is deliberately focused. It does not normally:

- Compile source code.
- Run the full application test suite.
- Build or scan container images.
- Replace a container registry.
- Replace Kubernetes controllers.
- Decide whether business behavior is correct.
- Automatically make every deployment safe.

A dangerous desired state can be faithfully deployed. Review, testing, policy, permissions, and monitoring remain necessary.

Benefits and trade-offs

The most important benefits are consistency, auditability, visibility, and recovery.

- **Consistency:** The same desired state can be rendered and compared repeatedly.
- **Auditability:** Git history and Argo CD operation history make changes easier to trace.
- **Visibility:** Engineers can see both drift and resource health without reconstructing a deployment from CI logs.
- **Recovery:** A bad desired-state change can often be reverted in Git and reconciled again.
- **Reduced cluster credentials in CI:** Build systems can update configuration rather than holding broad deployment access.

The model also introduces trade-offs.

- Reconciliation is not always instantaneous; polling and webhooks have finite latency.
- Deployment depends on repository and registry availability.
- A bad Git change can affect many resources quickly when automation is broad.
- Teams need disciplined ownership, review, and deletion rules.
- Live troubleshooting changes may be reverted by self-healing unless they are coordinated with the desired state.

The correct conclusion is not that GitOps removes operational risk. **It makes the intended state explicit and creates a controller-driven way to manage that risk.**

A practical comparison

Suppose a team needs to change a web service from two replicas to four.

In a command-driven process, an engineer might run:

```
kubectl scale deployment web --replicas=4 -n web
```

The command changes the cluster, but the repository may still say two replicas. A later deployment could restore two without anyone understanding why.

In a GitOps process, the engineer changes:

```
spec:
  replicas: 4
```

The pull request shows the intended change. After merge, Argo CD reports the difference and applies it according to the application's sync policy. If someone later scales the Deployment to five manually, Argo CD can show the drift and, when configured, return it to four.

Misconceptions that cause early mistakes

"GitOps means storing YAML in Git." Git storage is only one part. Automated comparison and reconciliation are essential.

"Argo CD is the entire CI/CD pipeline." Argo CD focuses on delivery and reconciliation of Kubernetes resources; CI still builds and verifies artifacts.

"A successful Git commit guarantees a healthy application." A commit can contain valid YAML and still produce an unhealthy workload.

"The cluster is the source of truth because it contains the running resources." The cluster contains live state. The approved source defines the intended state.

The mental model to retain is simple: **Argo CD is a controller that compares desired and live state. Git records intent; Kubernetes records reality; reconciliation brings them together.**

Chapter Summary

Argo CD continuously compares an approved desired state with the live state in Kubernetes and can reconcile differences when policy permits. Git records intended configuration, Kubernetes runs and reports resources, Argo CD observes and acts, and CI builds and tests artifacts. This separation improves repeatability and auditability, but it also requires disciplined repository ownership and change review.

Review Questions

1. What makes GitOps more than storing Kubernetes YAML in a repository?

2. Which system records desired state, which system reports live state, and what role does Argo CD play between them?
3. Why can a successful commit or synchronization still produce an unhealthy application?
4. Which build and test activities normally remain the responsibility of CI rather than Argo CD?

Chapter 2: The Kubernetes and Git Knowledge Argo CD Depends On

What You'll Be Able to Do

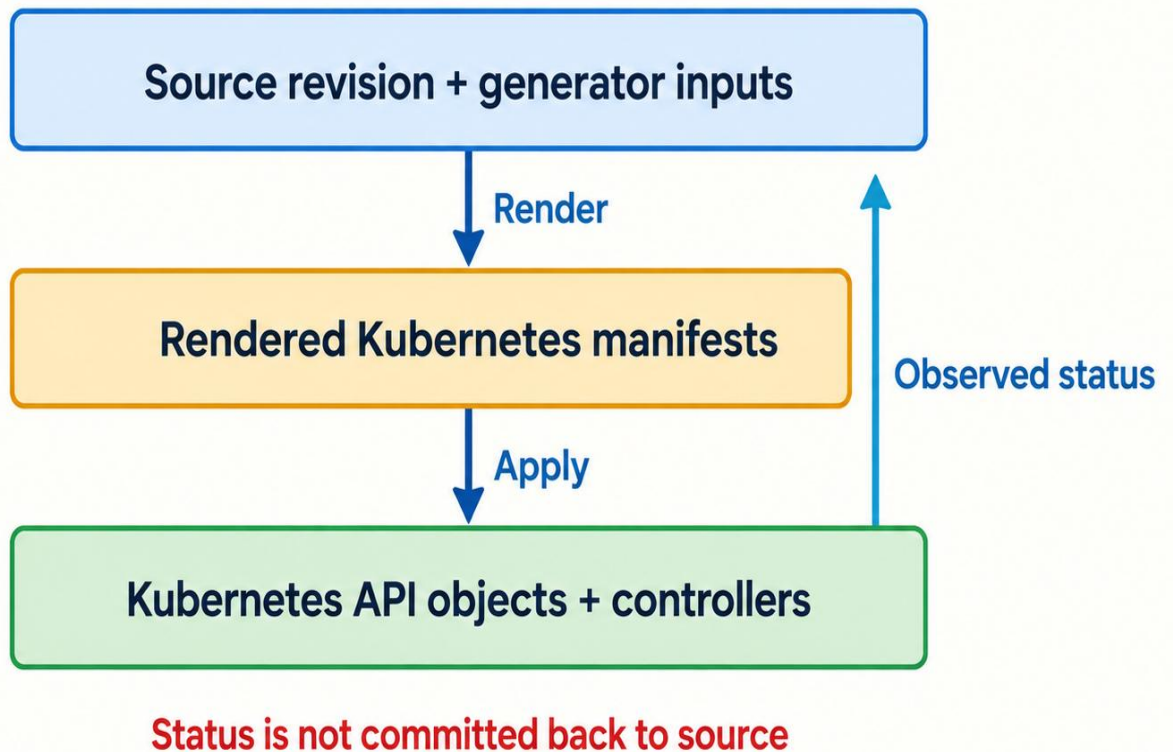
By the end of this chapter, you will be able to:

- Read the identity, desired-state, and observed-state fields of a Kubernetes resource.
- Explain how controllers, namespaces, ownership metadata, API access, and RBAC affect Argo CD behavior.
- Use the Git, image, Helm, and Kustomize concepts needed for later chapters.

The minimum foundation

Argo CD becomes predictable when you understand three layers: **Kubernetes resources, version-controlled revisions, and generated manifests**. You do not need to master every Kubernetes or Git feature first, but you must understand how desired state is represented and who is allowed to change it.

Chapter 2: The Kubernetes and Git Knowledge Argo CD Depends On 2-1 Three Prerequisite Layers



008

Figure 2-1: Three prerequisite layers for understanding Argo CD

Anatomy of a Kubernetes resource

A **Kubernetes resource** is an object stored through the Kubernetes API, such as a Namespace, Deployment, Service, ConfigMap, or Secret.

This is a small Deployment manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  namespace: web
```

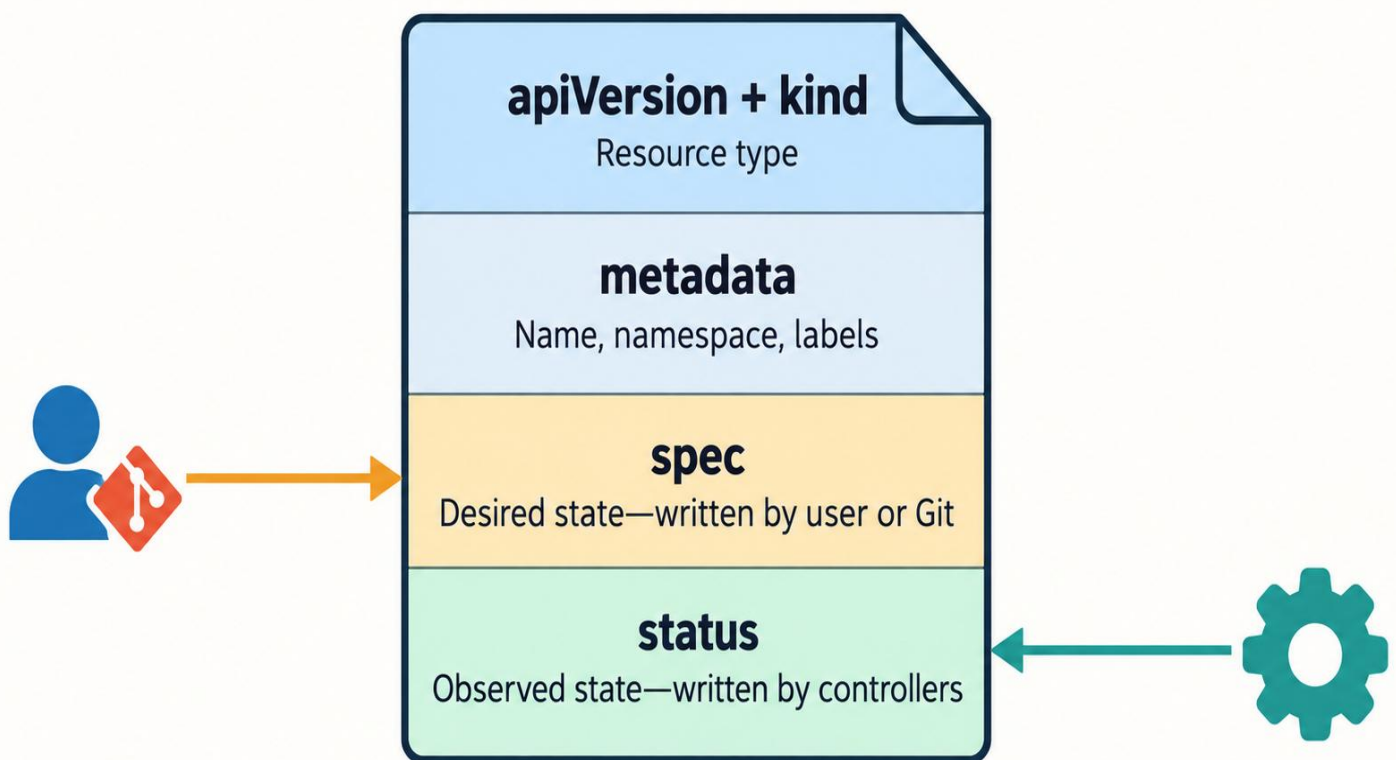
```
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.5-alpine
```

The main fields have distinct roles.

- `apiVersion` identifies the API group and version used to interpret the object.
- `kind` identifies the resource type.
- `metadata` identifies and organizes the object.
- `spec` describes the desired behavior or configuration.
- `status`, when present in a live object, reports controller-observed state.

Chapter 2: The Kubernetes and Git Knowledge Argo CD Depends On

2-2 Anatomy of a Kubernetes Resource



009

Figure 2-2: Anatomy of a Kubernetes resource: identity, desired state, and observed state

The manifest normally declares `spec`; Kubernetes controllers normally maintain `status`. Storing controller-generated status in Git is usually incorrect because it is observation, not intent.

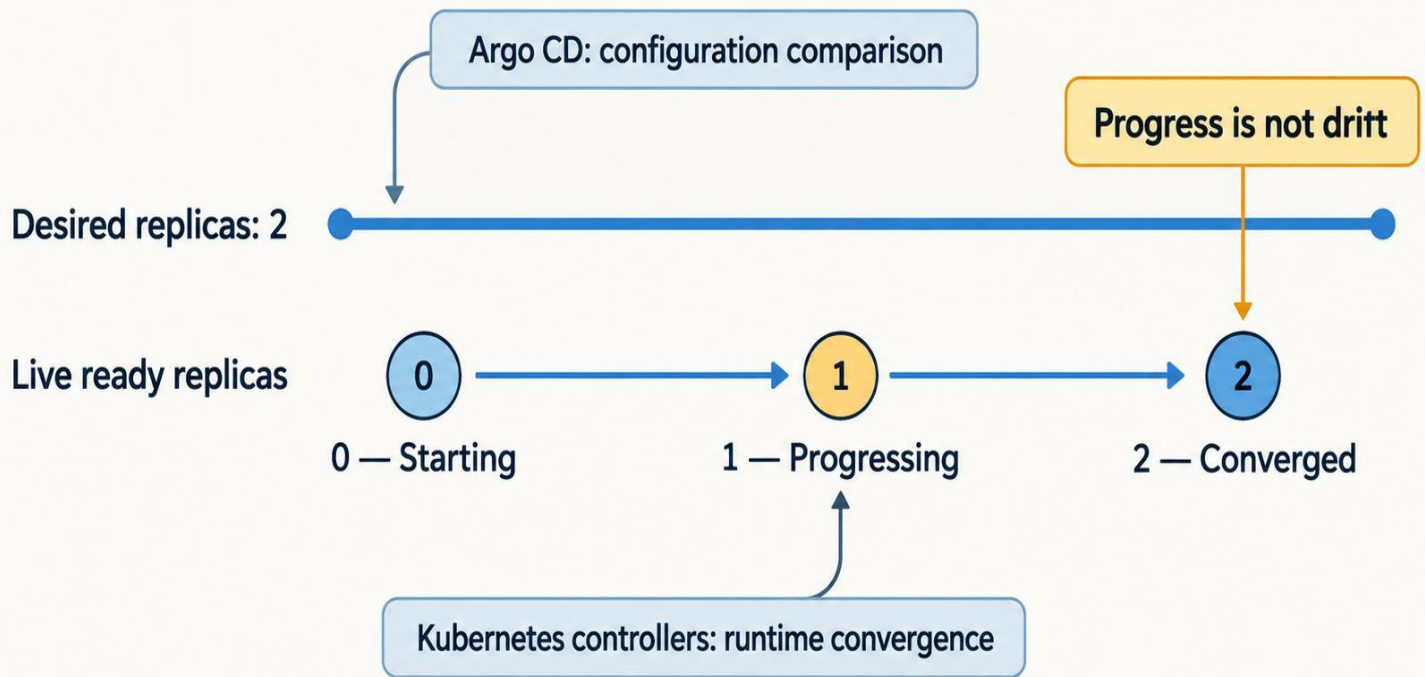
Desired state and live state

Desired state is the configuration that should exist. **Live state** is the object currently stored in the Kubernetes API and the runtime condition produced from it.

For the Deployment above, the desired state includes two replicas. The live state may temporarily contain one ready replica while Kubernetes is starting the second Pod. That does not necessarily mean the desired manifest is wrong. It means the controller has not yet completed reconciliation.

Chapter 2: The Kubernetes and Git Knowledge Argo CD Depends On

2-3 Desired and Live State During Convergence



010

Figure 2-3: Desired state and live state during convergence

Argo CD compares desired resource fields with live resource fields. Kubernetes controllers then perform the lower-level work needed to realize those fields.

The Kubernetes controller pattern

A **controller** is a process that watches resources and repeatedly works to make actual state match desired state.

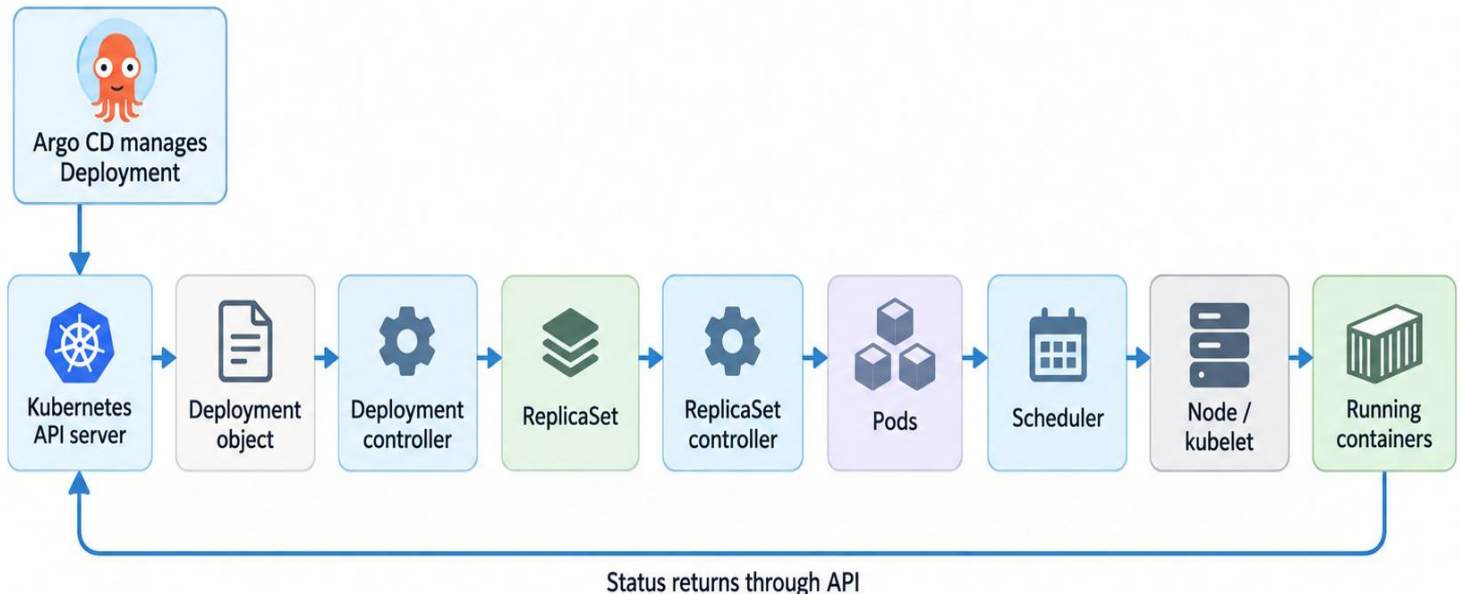
A Deployment illustrates the pattern:

1. The Deployment manifest is submitted to the API server.
2. The Deployment controller observes the new Deployment.
3. It creates or updates a ReplicaSet.
4. The ReplicaSet controller creates the required Pods.
5. The scheduler assigns Pods to nodes.
6. Kubelets start containers on those nodes.

7. Controllers update status as the rollout progresses.

Chapter 2: The Kubernetes and Git Knowledge Argo CD Depends On

2-4 Kubernetes Controller Chain to Running Containers



011

Figure 2-4: Kubernetes controller chain from Deployment to running containers

Argo CD operates one level above this sequence. It ensures the Deployment object itself matches the approved desired manifest. Kubernetes ensures the declared Deployment produces the required Pods.

Namespaces, labels, and annotations

A **namespace** is a Kubernetes scope used to group many namespaced resources and to support access and policy boundaries. A Deployment named `web` can exist in several namespaces because its full identity includes the namespace.

A Kubernetes namespace is not the same as an Argo CD project. A namespace organizes Kubernetes resources. An Argo CD project defines application policy and allowed sources or destinations.

A **label** is a key-value pair intended for identifying and selecting resources. Services and controllers commonly use labels to find related objects.

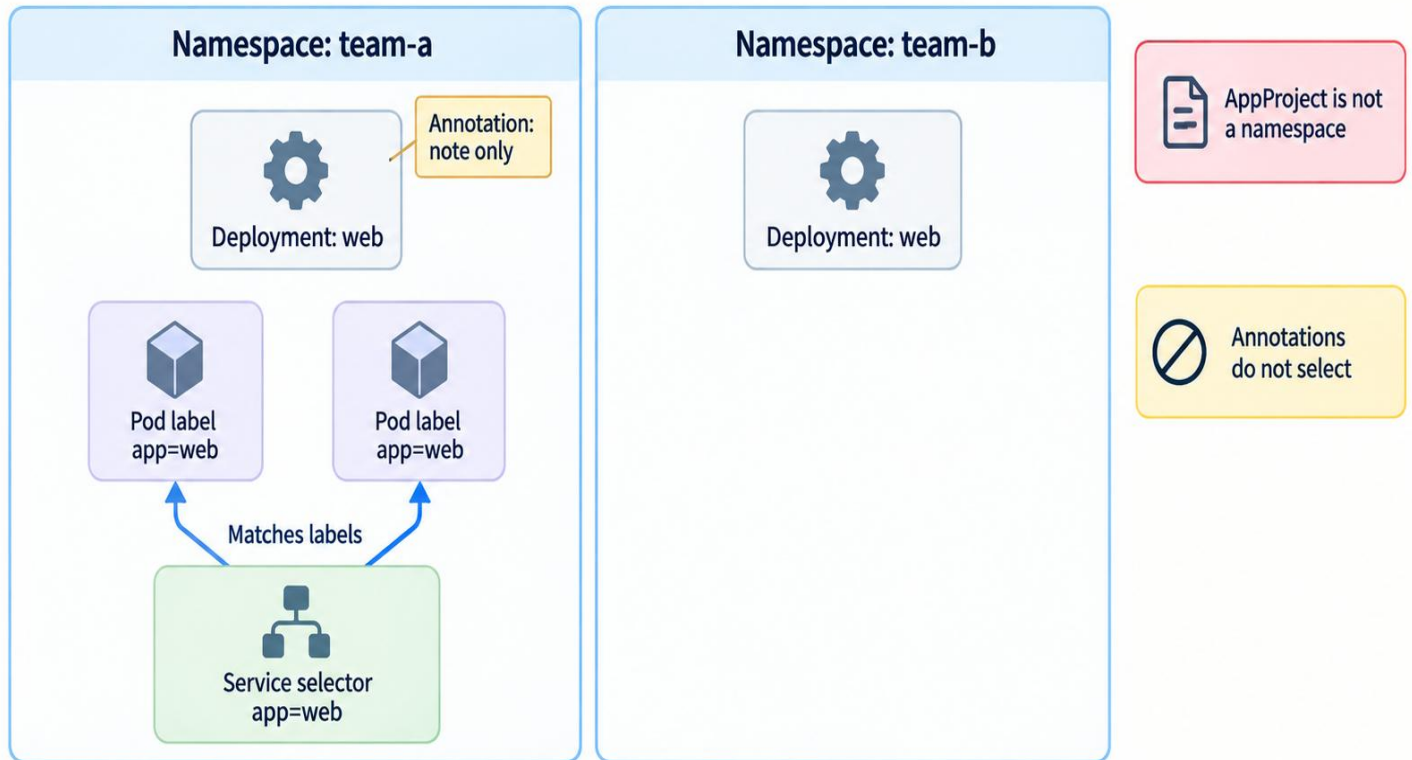
```
metadata:
  labels:
    app.kubernetes.io/name: web
    app.kubernetes.io/part-of: learning-demo
```

An **annotation** is a key-value pair used for non-identifying metadata, configuration hints, or integration-specific settings.

```
metadata:
  annotations:
    example.com/owner: platform-team
```

Chapter 2: The Kubernetes and Git Knowledge Argo CD Depends On

2-5 Namespaces, Labels, Annotations, and Selection



012

Figure 2-5: Namespaces, labels, annotations, and selection

Labels should contain values used for grouping or selection. Annotations can contain descriptive or tool-specific data that should not be used as a selector.

Owner references and finalizers

An **owner reference** records that one Kubernetes resource is owned by another, allowing Kubernetes garbage collection to remove dependents when the owner is deleted under the selected propagation policy.

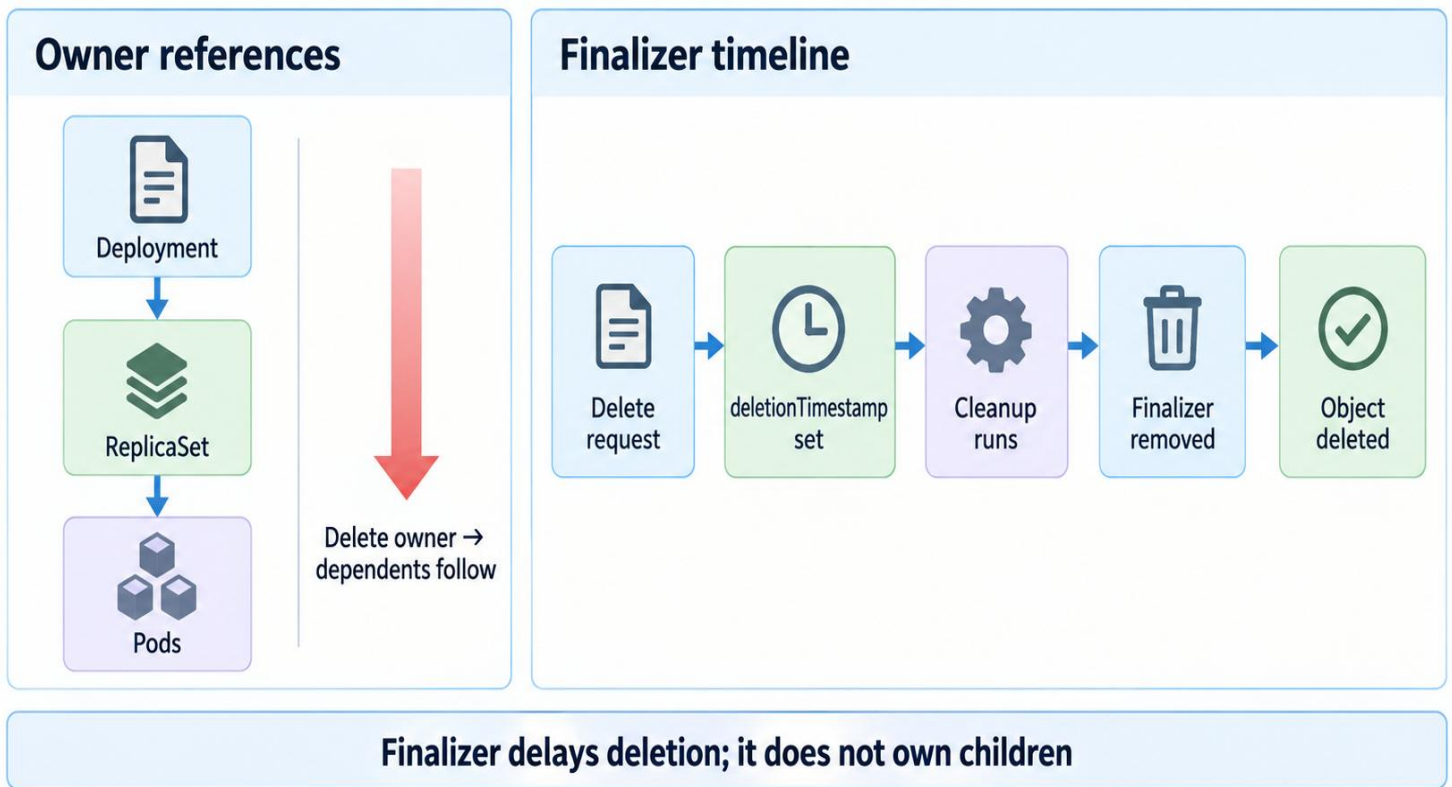
For example, a Deployment owns a ReplicaSet, and the ReplicaSet owns Pods. You normally do not create those owner references yourself; controllers do.

A **finalizer** is a metadata entry that delays deletion until a controller completes required cleanup. When an object with a finalizer is deleted, Kubernetes marks it for deletion but keeps it until the responsible controller removes the finalizer.

Argo CD can use a finalizer on an `Application` so that managed resources are deleted before the Application disappears. This behavior is powerful and must be understood before deleting applications.

Chapter 2: The Kubernetes and Git Knowledge Argo CD Depends On

2-6 Owner References and Finalizers



013

Figure 2-6: Owner references and finalizers

Custom Resource Definitions

A **Custom Resource Definition, or CRD**, adds a new resource type to the Kubernetes API.

Argo CD installs CRDs for resources such as:

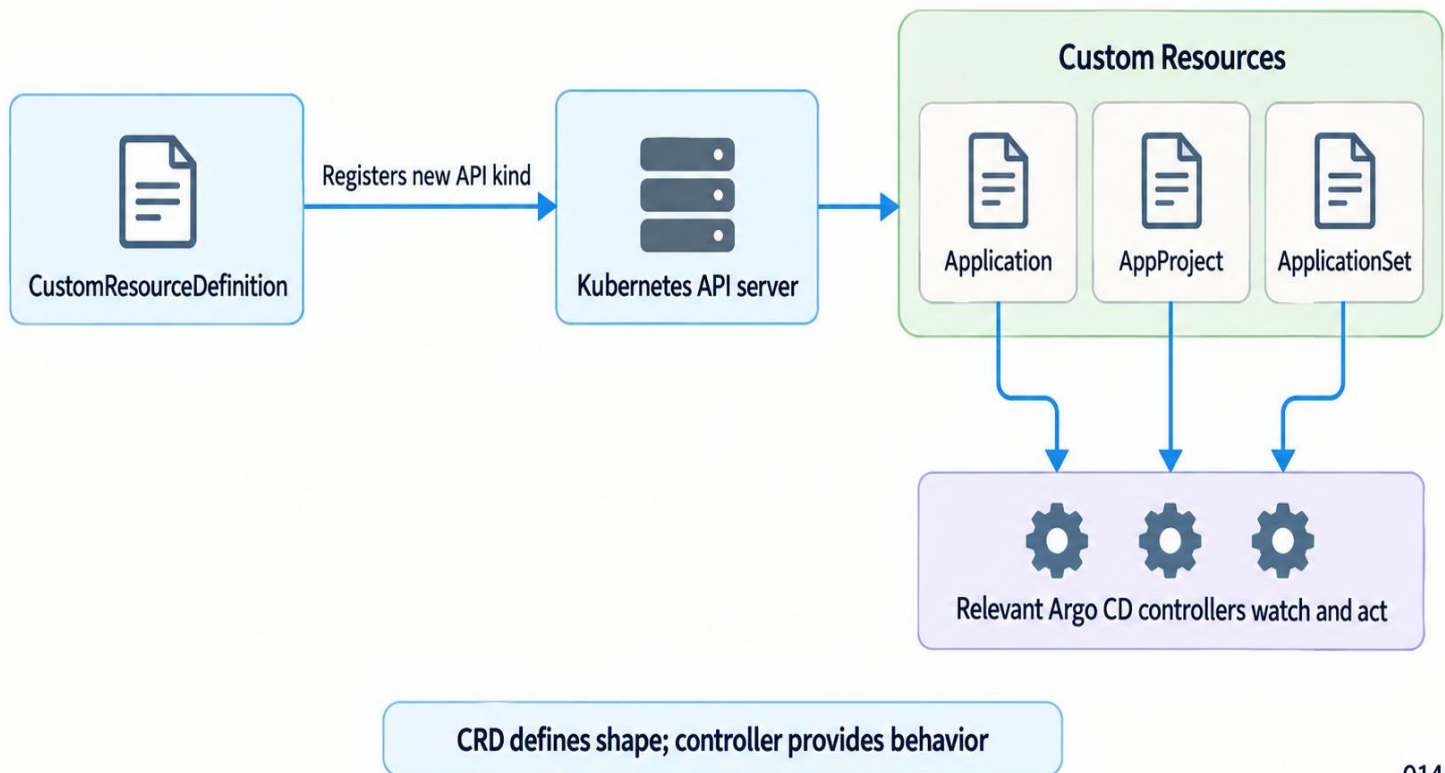
- `Application`
- `AppProject`
- `ApplicationSet`

After the CRDs exist, those objects can be created and managed through normal Kubernetes commands. This is why an Argo CD Application can be stored as YAML and applied with `kubectl`.

A custom resource is only data until a controller acts on it. The Argo CD application controller watches `Application` resources and performs the corresponding comparisons and sync operations.

Chapter 2: The Kubernetes and Git Knowledge Argo CD Depends On

2-7 How a CRD Extends the Kubernetes API



014

Figure 2-7: How a CRD extends the Kubernetes API

The API server is the center of cluster interaction

The **Kubernetes API server** validates, stores, and serves Kubernetes resources. `kubectl`, Argo CD, controllers, and operators communicate with it.

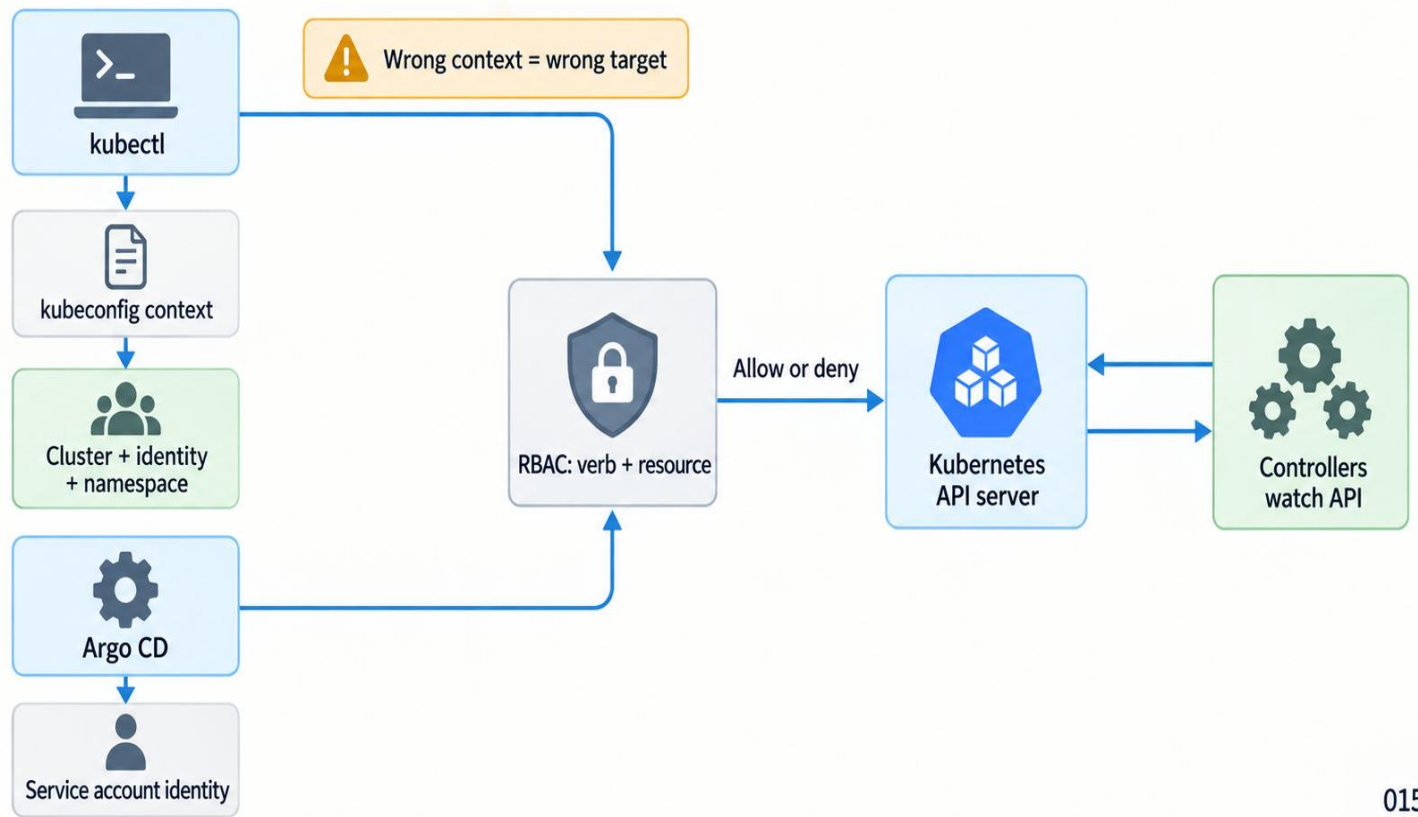
A tool's ability to work in a cluster depends on its identity and authorization.

- A **service account** is a Kubernetes identity normally used by in-cluster workloads.
- **Role-based access control, or RBAC**, determines which identities can perform which actions on which resources.
- A **kubeconfig** file stores cluster endpoints, credentials, certificate data, and named contexts for command-line access.

- A **context** selects a cluster, identity, and optional default namespace.

Chapter 2: The Kubernetes and Git Knowledge Argo CD Depends On

2-8 API Server, Context, Identity, and RBAC



015

Figure 2-8: Kubernetes API server, identities, RBAC, kubeconfig, and context

Selecting the wrong context is one of the most consequential beginner mistakes because a correct command can then target the wrong cluster.

Essential `kubectl` commands

The following commands are sufficient for the early Argo CD workflow.

Confirm the current target

Before running any modifying command, use a configured kubeconfig and verify the selected context:

```
kubectl config current-context
kubectl cluster-info
```

The first command prints the context name. The second confirms that the API server is reachable. The expected result is the intended learning or non-production cluster, not merely any reachable cluster.

List resources

With access to the target namespace, run:

```
kubectl get deployment,pod,service -n web
```

This reads Deployments, Pods, and Services from the `web` namespace. The expected result is a concise status table. It does not explain why a resource is unhealthy.

Inspect details and events

When a resource does not behave as expected, run:

```
kubectl describe deployment web -n web
```

This shows configuration, current conditions, and recent related events. The expected result is enough context to identify common problems such as unavailable replicas, image pull errors, or failed scheduling.

Preview declarative differences

If local manifests are intended for direct `kubectl` management, run:

```
kubectl diff -f manifests/
```

This asks the API server to calculate the difference between the local files and live objects. The expected result is diff output when changes exist and no resource modification. In an Argo CD-managed application, prefer Argo CD's diff because it uses the Application's configured source and rendering settings.

Apply a manifest

With permission to create the declared resources, run:

```
kubectl apply -f application.yaml
```

This creates or updates the resource in the current cluster. When the file contains an Argo CD `Application`, the expected result is creation or configuration of the Application object; Argo CD then handles the application resources according to its policy.

Read controller or workload logs

To inspect one container's logs, run:

```
kubectl logs deployment/web -n web
```

The expected result is application log output from a selected Pod behind the Deployment. For multi-container Pods, specify the container with `-c`.

`kubectl logs` explains runtime behavior. It does not tell you whether the live manifest matches Git; Argo CD provides that comparison.

Git concepts needed for Argo CD

A **commit** is an immutable snapshot of tracked content plus metadata and parent history. Argo CD can target a specific commit, branch, or tag.

A **branch** is a movable name that normally advances as commits are added. Tracking a branch such as `main` means the desired state can change whenever the branch advances.

A **tag** is a named reference commonly used for releases. Tags can be protected by repository policy, but Git itself does not make every tag immutable; a poorly governed tag can be moved.

A **pull request** is a review workflow for proposing changes before they are merged. The exact name varies by hosting platform, but the important idea is reviewable change.

A **revert** creates a new commit that reverses the effect of an earlier commit while preserving history. In GitOps operations, a revert is often clearer than manually editing the cluster back to an older state.

Merge history matters because it links an approved configuration to a specific revision. When Argo CD reports a revision, engineers should be able to find the corresponding commit and review.

Container registries, tags, and digests

A **container registry** stores versioned container images. Kubernetes pulls images from a registry based on an image reference.

An image tag is a readable name:

```
registry.example.com/team/web:1.4.2
```

A tag may be moved to a different image. The tag `latest` is especially ambiguous because its meaning can change without a manifest change.

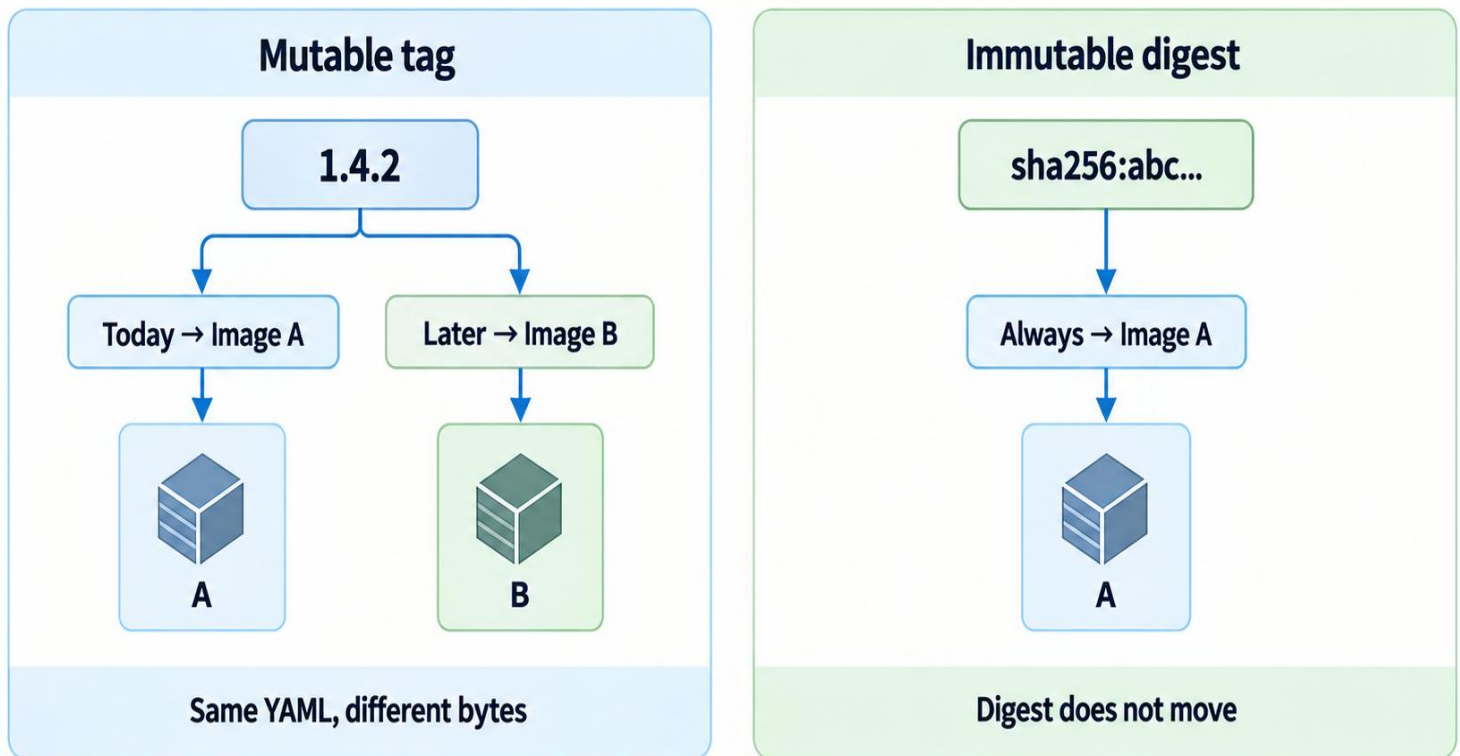
An image digest is a content-addressed identifier:

```
registry.example.com/team/web@sha256:0123456789abcdef...
```

A digest identifies exact image content, while a tag identifies a name that may be reassigned. Controlled promotion processes prefer digests because the deployed artifact is unambiguous.

Chapter 2: The Kubernetes and Git Knowledge Argo CD Depends On

2-11 Mutable Image Tags vs Immutable Digests



016

Figure 2-9: Mutable image tags versus immutable digests

Helm and Kustomize in one sentence each

Helm is a package and templating tool that renders Kubernetes manifests from a chart and values. Argo CD can render Helm charts before comparison and synchronization.

Kustomize transforms and combines Kubernetes manifests without using a general-purpose template language. Argo CD can build a Kustomize base or overlay before comparison.

Both tools generate manifests. Argo CD ultimately compares the generated Kubernetes resources with the cluster. A beginner does not need full Helm or Kustomize mastery to understand that rendering step.

Follow one Deployment from file to Pods

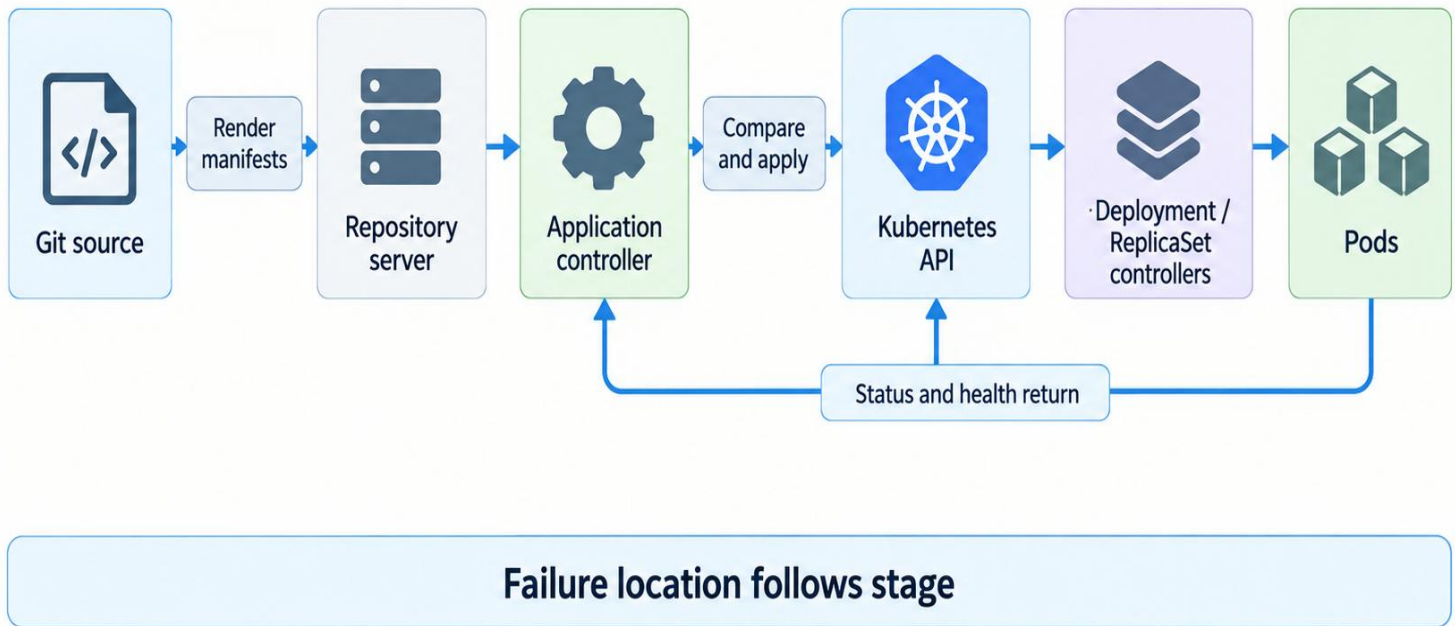
Use the earlier Deployment as a concrete trace.

1. A YAML file declares a Deployment named `web` with two replicas.
2. The file is committed to a source repository.
3. Argo CD reads the selected revision and generates the Deployment manifest.
4. Argo CD compares the generated Deployment with the live Deployment.

5. During sync, Argo CD submits the desired Deployment to the API server.
6. The Deployment controller creates or updates a ReplicaSet.
7. The ReplicaSet controller ensures that two Pods exist.
8. Kubernetes reports rollout status.
9. Argo CD reads the live Deployment and evaluates both sync and health.

Chapter 2: The Kubernetes and Git Knowledge Argo CD Depends On

2-13 End-to-End Path from Repository File to Pods



017

Figure 2-10: End-to-end path from repository file to Pods

Each layer has a separate responsibility. When troubleshooting, identifying the failing layer is more effective than changing random fields.

Readiness checklist

Continue when you can explain the following without relying on memorized wording:

- The difference between `spec` and `status`.
- The difference between desired state and live state.
- Why a controller continues watching after a resource is created.
- How a namespace differs from an Argo CD project.
- Why a kubeconfig context must be checked before modifying a cluster.
- How a branch differs from a commit.
- Why an image digest is more precise than a mutable tag.
- That Helm and Kustomize generate Kubernetes manifests for Argo CD to compare.

Mistakes that distort the mental model

Confusing a namespace with an Argo CD project: They operate at different layers and enforce different kinds of boundaries.

Using `latest` as a precise release identifier: A mutable tag can point to different bytes without a Git change.

Assuming repository file order controls creation order: Kubernetes and Argo CD use resource relationships, phases, waves, and health—not alphabetical filenames—as their operating model.

Treating `status` as desired configuration: Status is normally written by controllers and changes as the system runs.

The essential foundation is that **Git identifies an approved revision, Argo CD renders its desired resources, and Kubernetes controllers realize those resources.**

Chapter Summary

Argo CD depends on the same API-driven, declarative model used by Kubernetes. Accurate reasoning therefore requires a working understanding of resource fields, controller ownership, namespaces, finalizers, CRDs, API identities, and basic Git history. Immutable image digests and predictable manifest generation make the desired state precise enough to review and reproduce.

Review Questions

1. What is the difference between a Kubernetes resource's `spec` and `status`?
2. How do owner references and finalizers affect resource deletion?
3. Why must a systems engineer confirm the active kubeconfig context before running a command?
4. Why is an image digest a more precise release identifier than a mutable tag?