Infrastructure Automation Tools

TEKsystems Global Services

WORKFORCE DEVELOPMENT

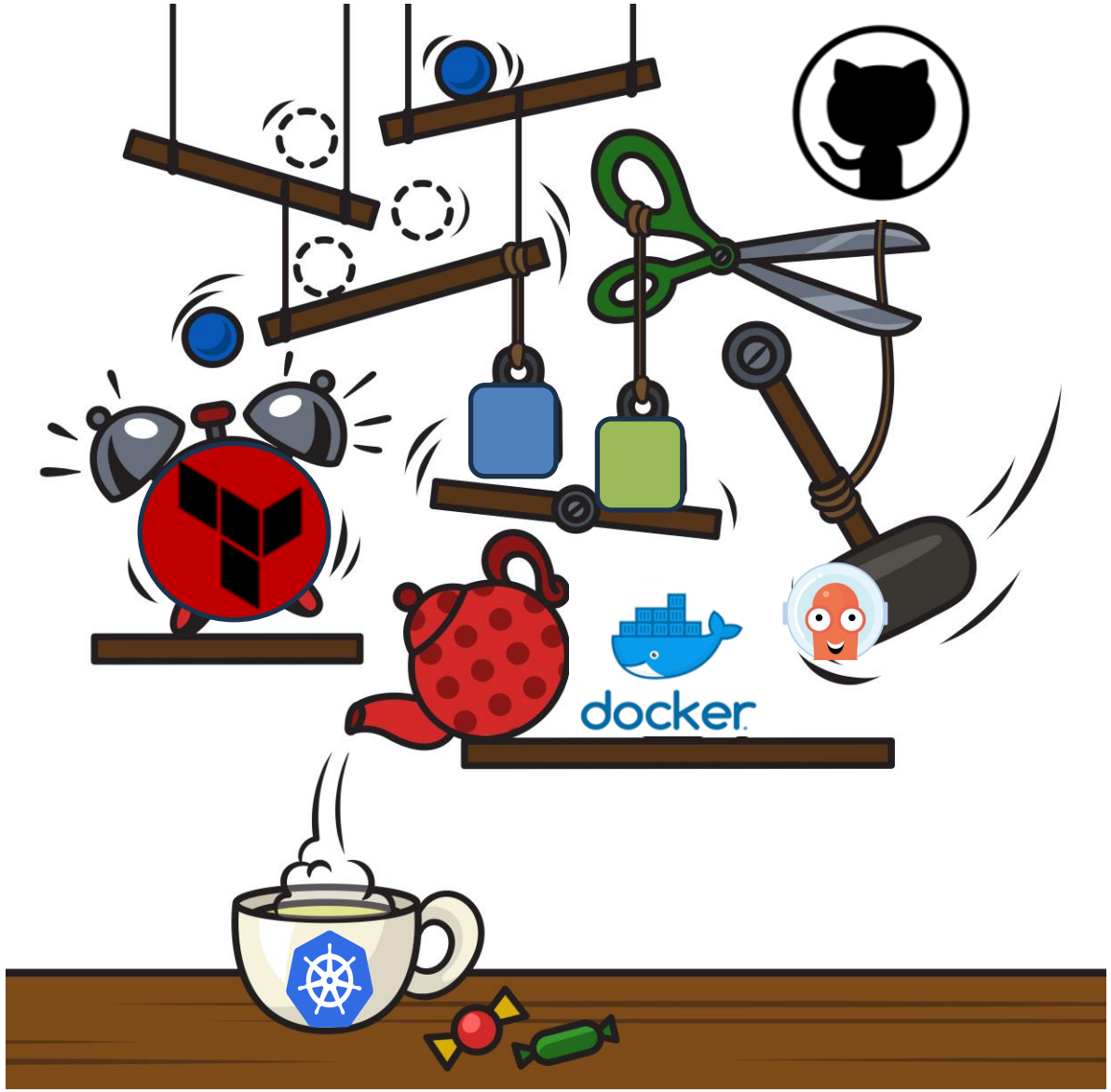
CI/CD and Infra Automation

Learning Objectives

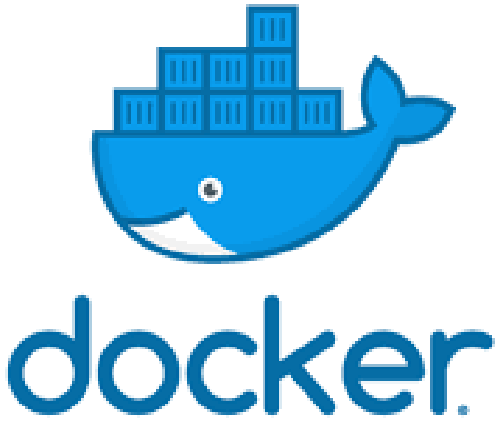
Day 4 — Staging, Releases, GitOps & ArgoCD

- Learn how applications move through **Dev → Staging → Prod**
- Deploy using **environment-specific tags**
- Test staged releases and perform **safe rollbacks**
- Understand **GitOps fundamentals**
- Use **ArgoCD** to deploy applications from Git automatically
- Manage **container environment variables** and configurations
- Build and promote **new releases** through CI/CD

CI CD



Automation tools



kubernetes

Staging

Staging

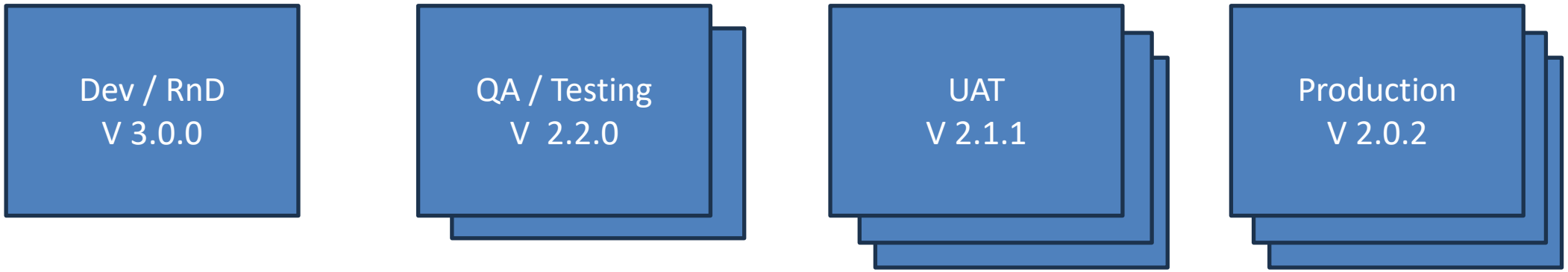
Staging is a **pre-production environment** that closely mirrors real production so teams can safely validate new releases before users see them.

Key points:

- Used to test deployments, configs, and infrastructure changes
- Catches environment-specific bugs that don’t appear in Dev
- Allows realistic testing without impacting real customers
- Ensures only stable, verified builds are promoted to Prod

Staging reduces deployment risk by acting as the final checkpoint before production.

Staging



Production is the most powerful and heavily monitored environment because it serves real users and carries the highest risk. UAT (User Acceptance Testing) is typically a close mirror of production, used by stakeholders or testers to validate releases in a realistic setting.

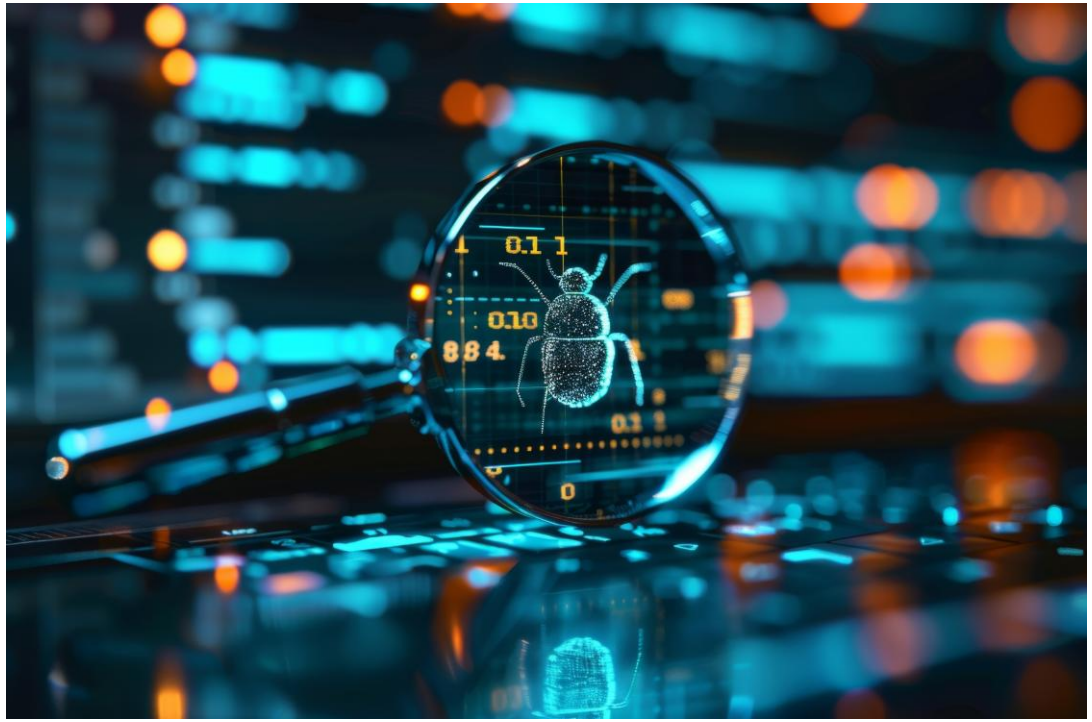
QA environments are similar in architecture but run with fewer resources and are primarily used by engineers for functional and integration testing. Development environments are the most flexible, giving developers space to build, experiment, and iterate quickly without affecting other stages.

Dev / RnD env

The development environment is where engineers build, experiment, and test changes quickly without affecting downstream environments.

- Often integrates with a **CI/CD pipeline** that auto-deploys the latest commits
- Enables rapid iteration and immediate feedback for developers
- Used heavily for **R&D, prototyping, and early-stage testing**
- May contain mocked services, temporary data, or partial configs to speed up work

QA Testing ENV



The QA environment is where engineers and testers validate application behavior in a controlled, repeatable setting before it moves closer to production.

- Mirrors production architecture but typically with fewer resources
- Used for **functional testing, integration testing, synthetic testing, and regression testing**
- Helps identify issues that dev environments may hide due to mocks or shortcuts
- Often populated with test data rather than real customer data

UAT Env

UAT (User Acceptance Testing) is where stakeholders, product owners, or end-users validate that the application behaves correctly for real-world workflows.

- Closely mirrors **production** in configuration and behavior
- Used to confirm that features meet business requirements and expectations
- Testers follow real user journeys rather than technical test cases
- Typically uses sanitized or realistic test data
- Primarily for **business validation**

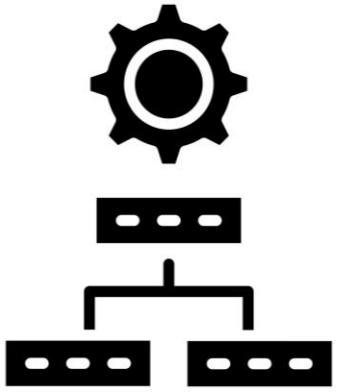
UAT serves as the final approval checkpoint before promoting a release into production.

Production Env

Production is the live environment where real users interact with the application, making it the most stable, monitored, and resource-intensive environment in the entire pipeline.

- Hosts **real customer traffic** and must remain highly available
- Backed by the **strongest monitoring, alerting, and scaling strategies**
- Uses the most optimized infrastructure and security configurations
- Changes are introduced carefully through staged promotions and rollout strategies
- Failures here have direct business impact, so reliability is the top priority

Promotion

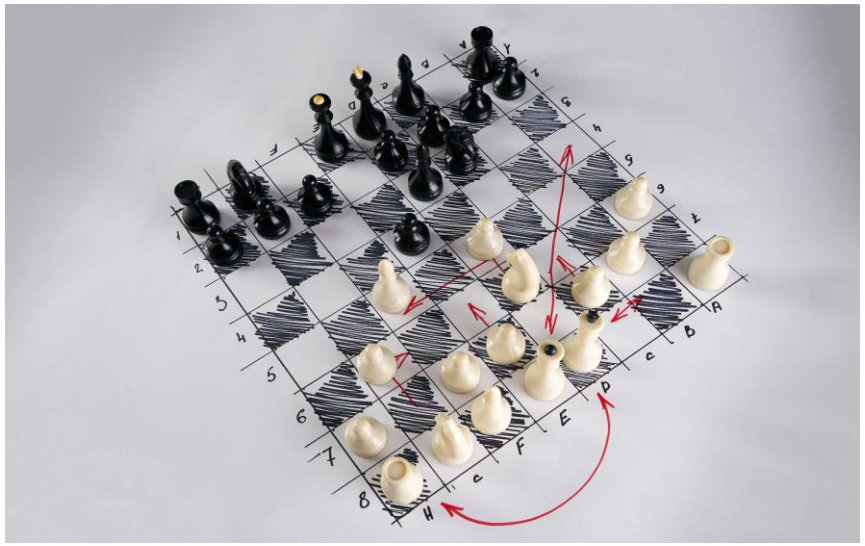


Promotion means moving a release from one environment to the next, but each stage should include testing to ensure the build is ready for the next level.

- **Dev:** Unit tests validate small, isolated pieces of code
- **QA:** Regression, functional, stress, and load tests; verify against **SLOs/SLAs**
- **UAT:** End-users or product owners validate real workflows and business requirements
- **Prod:** Synthetic tests simulate user behavior to confirm the system is healthy before real traffic hits

Promotion is not just moving code forward, it's a sequence of quality gates that protect production stability.

Release Strategies



Release strategies define *how* new versions of an application are rolled out to users while minimizing risk.

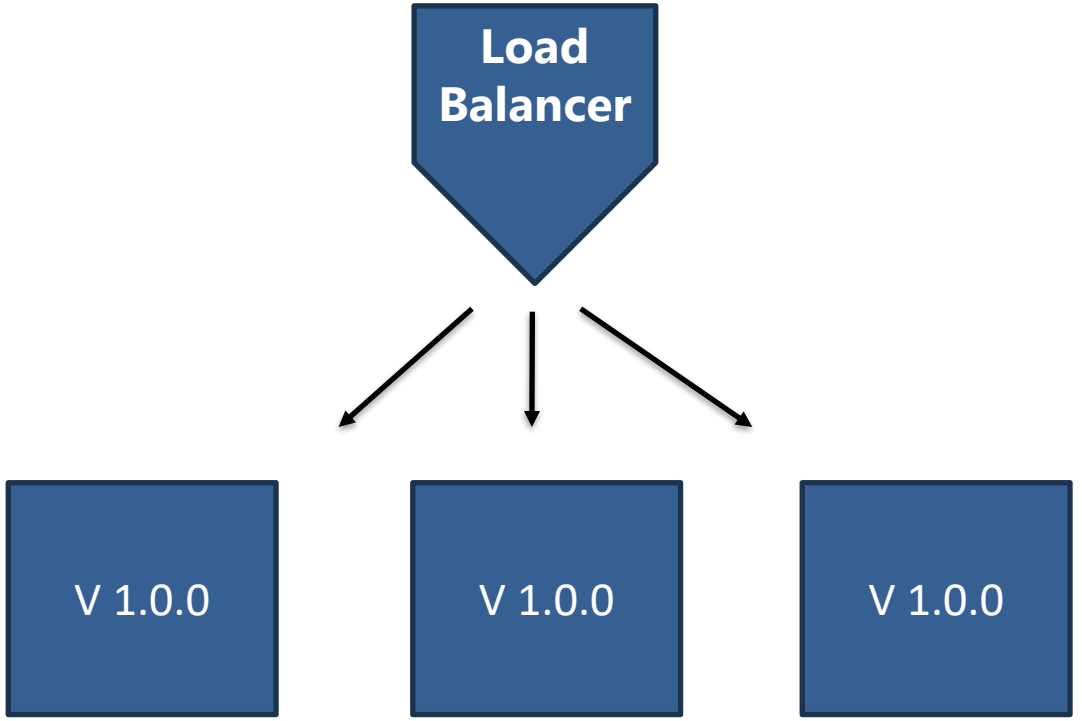
- **Rolling Deployment:** Gradually replace old instances with new ones; users slowly shift to the new version
- **Blue/Green Deployment:** Two identical environments; new version goes to "Green" and traffic switches only when verified
- **Canary Release:** Release to a small percentage of users first; expand gradually as confidence grows
- **Recreate Deployment:** Shut down old version, then bring up the new one — simple but causes downtime
- **A/B Testing:** Deploy two versions and split traffic intentionally to compare user behavior or performance

Rolling Update

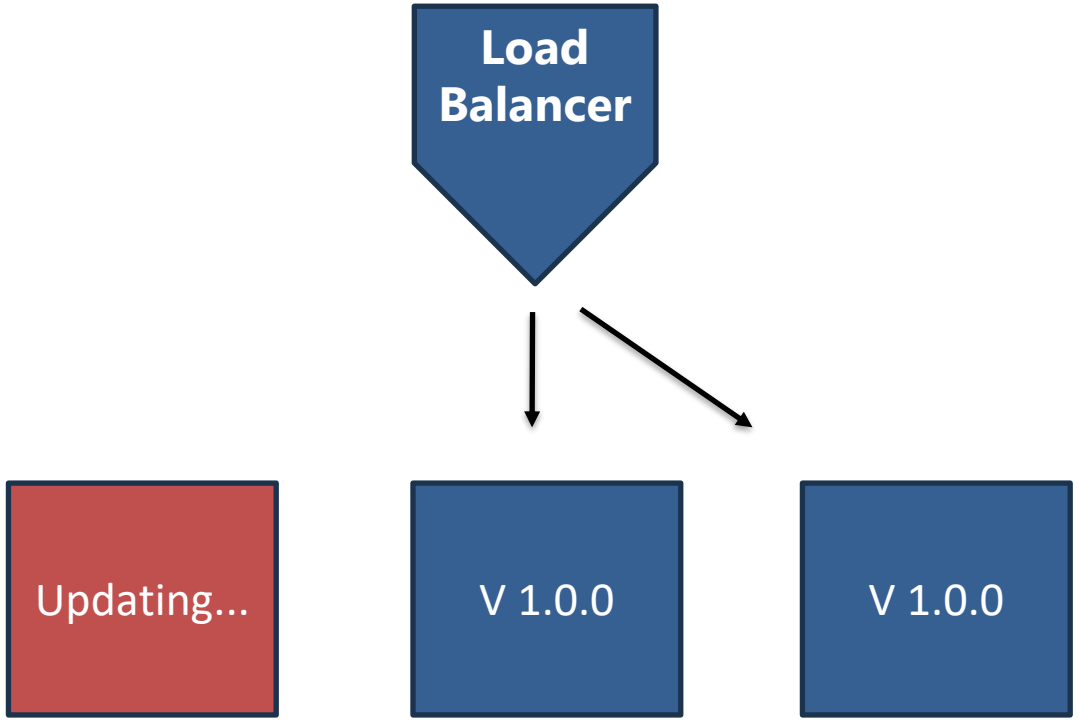
A rolling update gradually replaces the old version of an application with a new one, updating a subset of instances at a time.

- New instances are brought online before old ones are removed
- Traffic is shifted incrementally to reduce risk and downtime
- Works well with **Auto Scaling Groups** and container platforms
- Rollback usually means redeploying the previous version

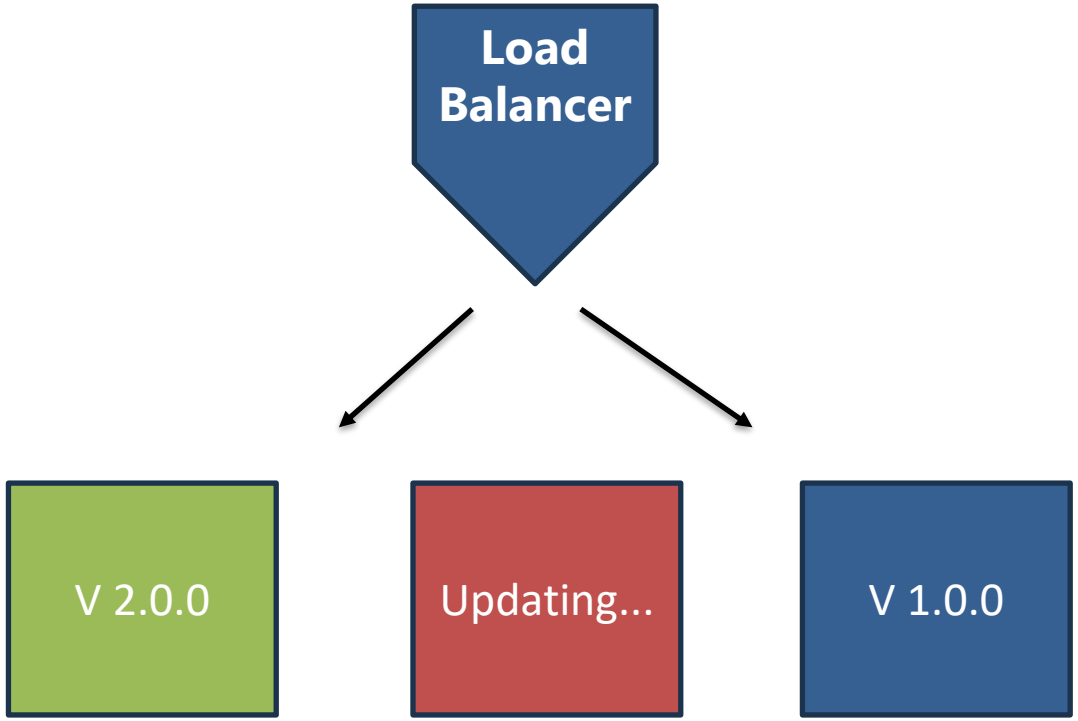
Rolling updates balance availability and simplicity, making them one of the most common deployment strategies in production systems.



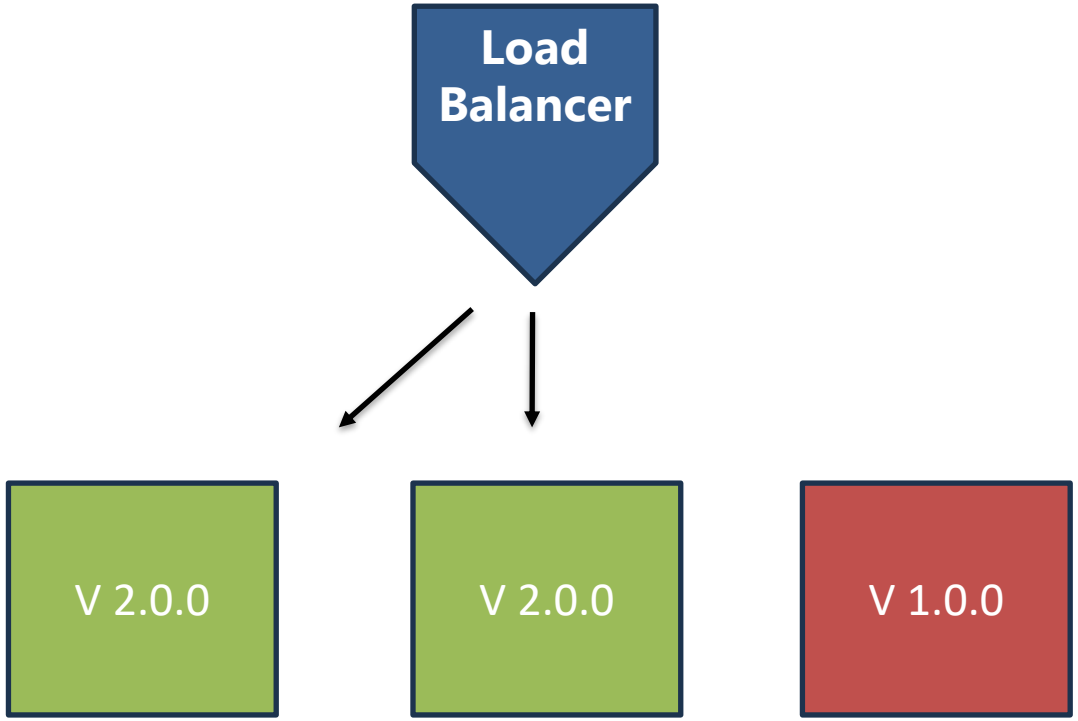
Rolling Update



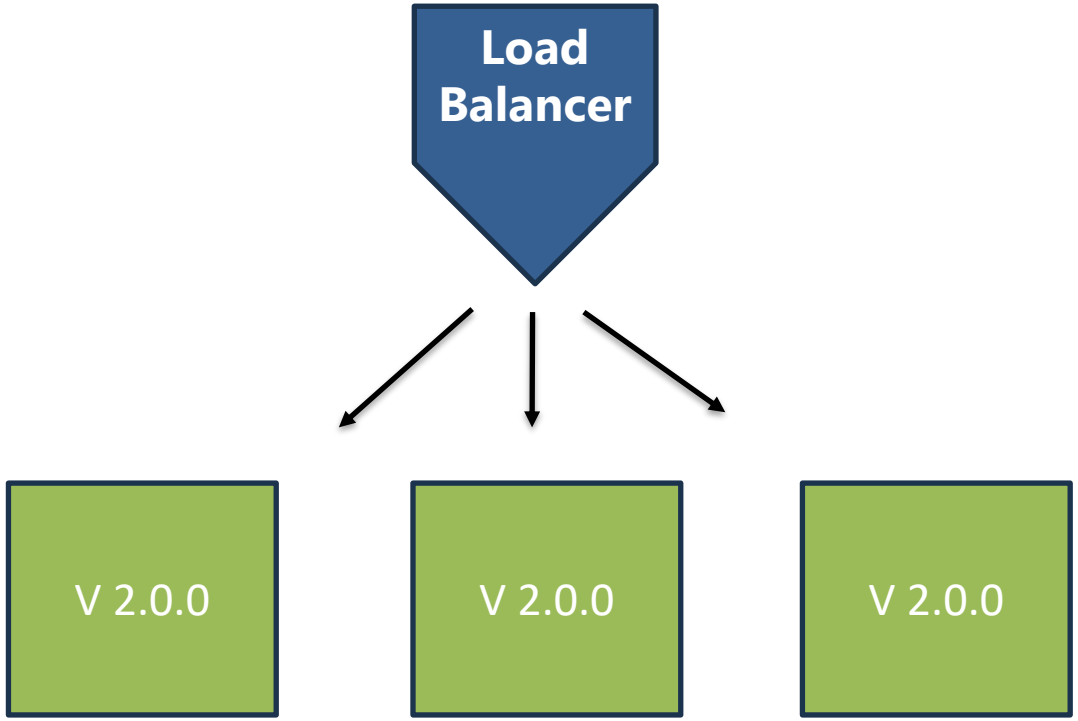
Rolling Update



Rolling Update



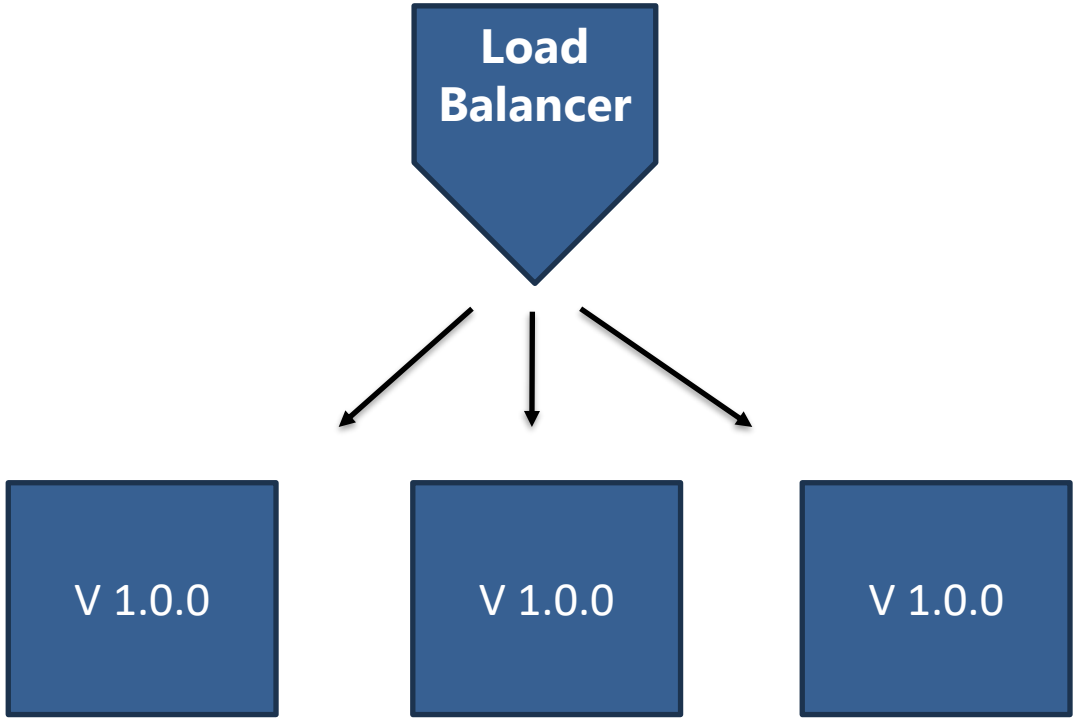
Rolling Update



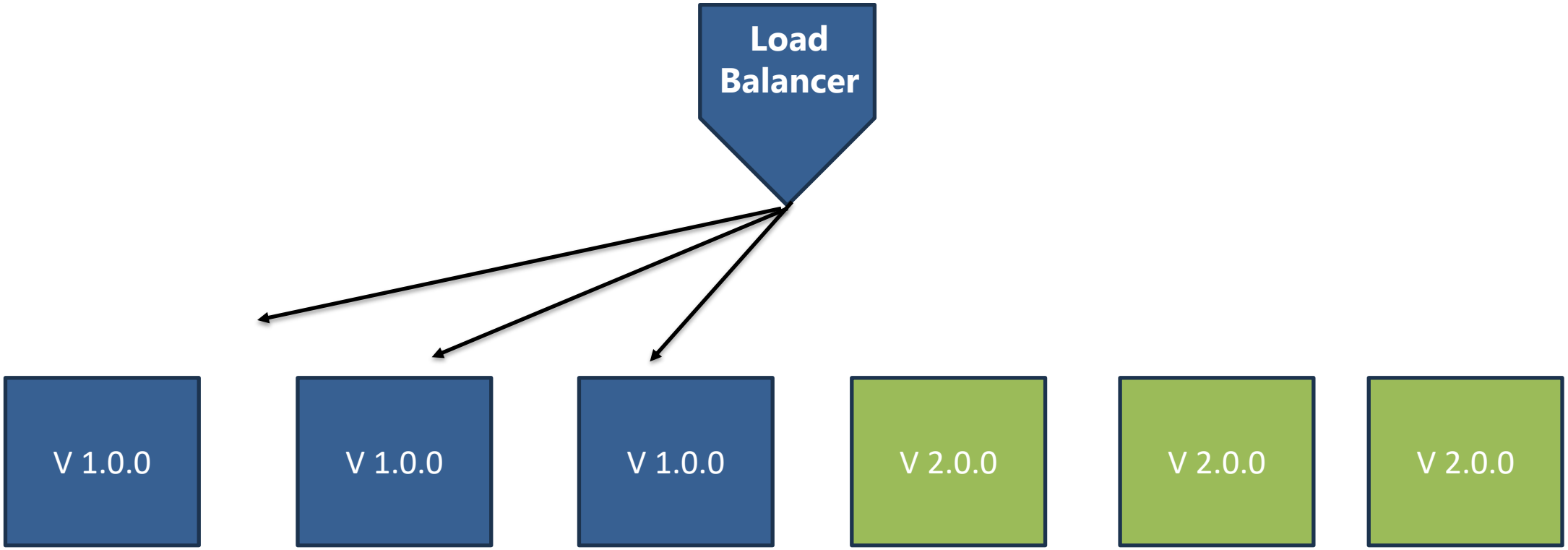
Blue/Green Deployment

Blue/Green is a widely used deployment strategy where two identical environments run side by side. The new version is deployed to the Green environment while the Blue environment continues serving users. Once the new version is tested and verified, traffic is switched from Blue to Green.

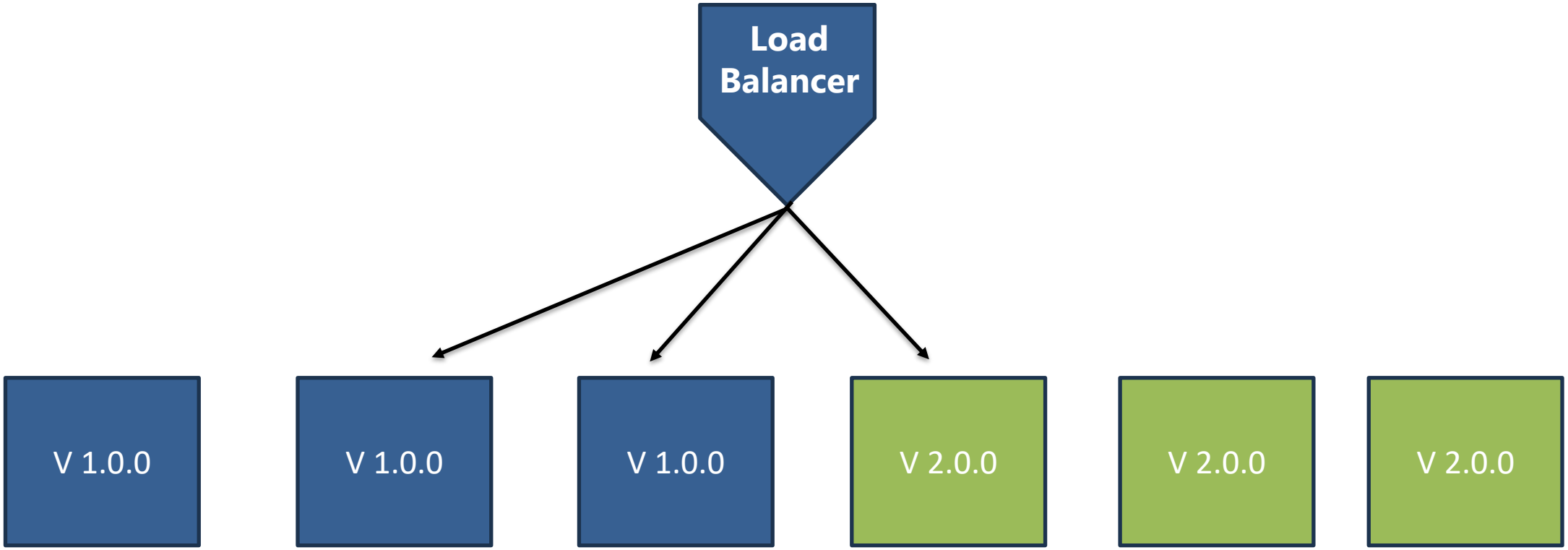
If something goes wrong, rollback is immediate by switching traffic back. The primary downside is cost — maintaining two full environments means higher infrastructure usage.



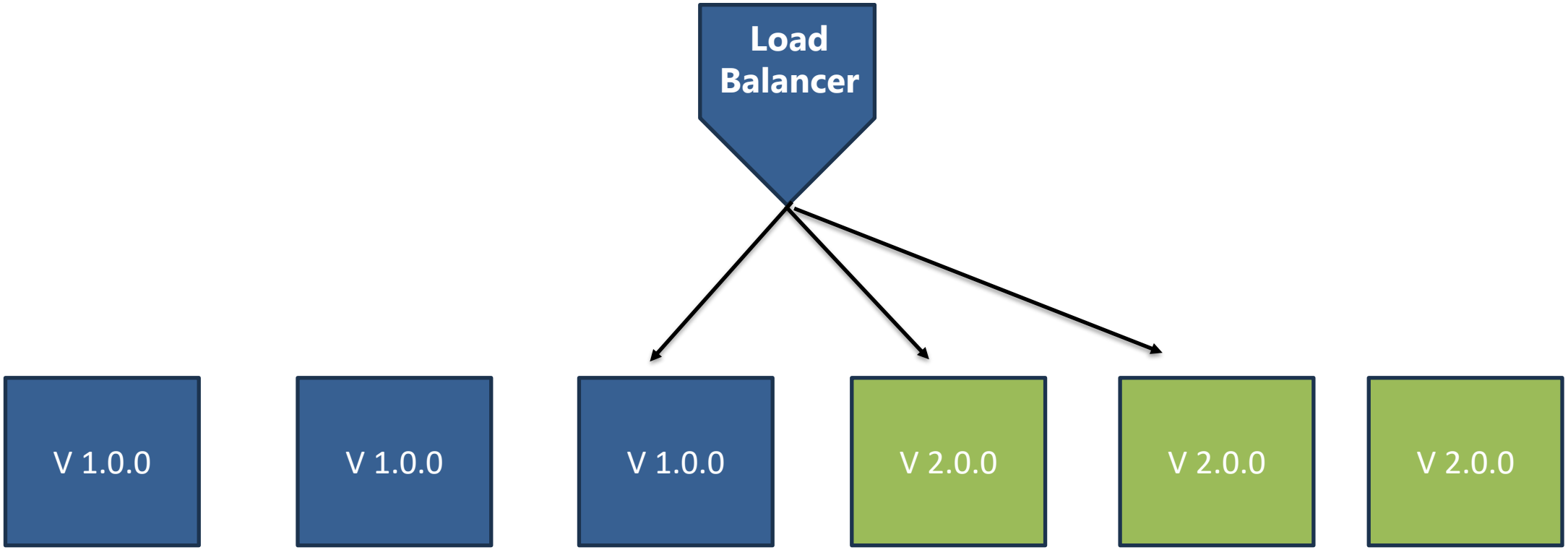
Blue/Green Deployment



Blue/Green Deployment



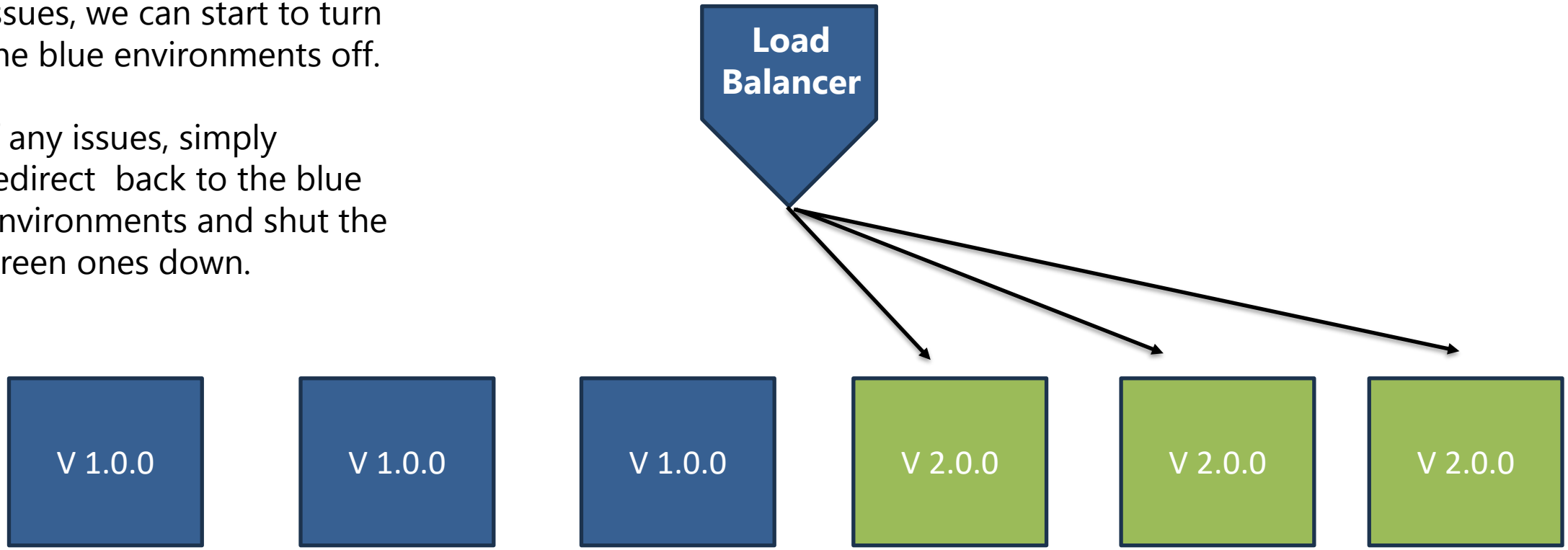
Blue/Green Deployment



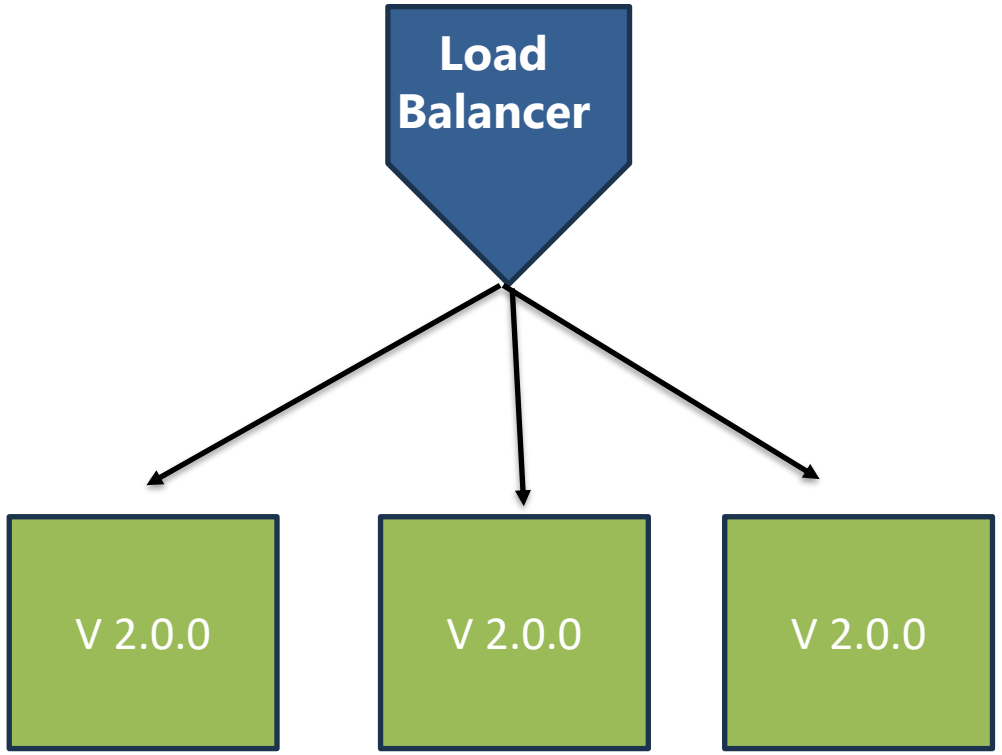
Blue/Green Deployment

As long as there are no issues, we can start to turn the blue environments off.

If any issues, simply redirect back to the blue environments and shut the green ones down.



Blue/Green Deployment



Rollback

Rollback is the ability to quickly return to a previous, stable version when a release causes issues.

- Used when a deployment introduces errors or instability
- Faster and safer than trying to patch issues in production
- Often implemented by switching traffic, reverting a tag, or redeploying a known-good artifact
- Most effective when releases are **immutable and versioned**

Rollback is a critical safety mechanism that allows teams to recover quickly and protect users when things go wrong.

Change Management

Change management is the process of planning and controlling changes to systems to minimize risk and avoid unexpected outages.

- Changes to production are **planned and reviewed** in advance
- **Blackout periods** prevent risky deployments during critical business times
- **CABs (Change Advisory Boards)** may review and approve high-impact changes
- Stakeholders are informed before changes occur
- A **rollback plan** is defined before any change is executed

Strong change management balances speed with stability, ensuring progress without unnecessary disruption.

POP QUIZ:

What is the primary purpose of a staging environment?

A. To allow developers to experiment freely
B. To replace production during outages
C. To validate releases in a production-like environment before users see them
D. To perform unit testing

POP QUIZ:

What is the primary purpose of a staging environment?

A. To allow developers to experiment freely
B. To replace production during outages
C. **To validate releases in a production-like environmen**t
D. To perform unit testing

POP QUIZ:

Which environment typically has the most monitoring and allocated resources?

A. Development
B. QA
C. UAT
D. Production

POP QUIZ:

Which environment typically has the most monitoring and allocated resources?

A. Development
B. QA
C. UAT
D. **Production**

POP QUIZ:

Which environment is mainly used for business and user workflow validation?

A. QA
B. Development
C. UAT
D. Production

POP QUIZ:

Which environment is mainly used for business and user workflow validation?

A. QA
B. Development
C. **UAT**
D. Production

POP QUIZ:

What is a key goal of testing in QA environments?

A. Validate user experience workflows
B. Test against SLOs and SLAs
C. Serve real customer traffic
D. Perform feature experimentation

POP QUIZ:

What is a key goal of testing in QA environments?

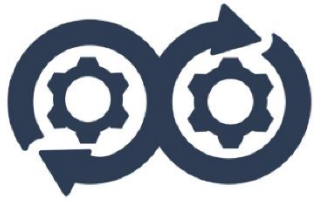
A. Validate user experience workflows
B. **Test against SLOs and SLA**s
C. Serve real customer traffic
D. Perform feature experimentation

POP QUIZ:

What is the primary purpose of rollback?

A. Speed up deployments
B. Improve performance
C. Revert to a known good version after a failure
D. Reduce infrastructure costs

Lab: Zero Downtime Deployment



GitOps with ArgoCD

Git Ops

GitOps is a deployment model where **Git is the single source of truth** for application and infrastructure state.

- Desired state is defined and versioned in Git
- Changes are made through **pull requests**, not manual actions
- Automated systems continuously reconcile Git with the running environment
- Every change is **auditable, reviewable, and reversible**

GitOps improves reliability and confidence by turning deployments into controlled, repeatable Git operations.

Kubernetes

Kubernetes is a platform for **running and managing containerized applications at scale**.

- Schedules and runs containers across multiple machines
- Handles **scaling, restarts, and availability** automatically
- Abstracts away individual servers so apps run as a system
- Commonly used with CI/CD and GitOps workflows

Kubernetes focuses on keeping applications running reliably, not on how they are built.

ArgoCD

ArgoCD is a **GitOps continuous delivery tool** for Kubernetes that deploys applications directly from Git repositories.

Key points:

- Continuously watches Git for desired application state
- Automatically syncs changes from Git to Kubernetes
- Provides visibility into **what's deployed vs what should be deployed**
- Enables easy rollbacks by reverting Git changes

ArgoCD turns Git into the control plane for deployments, making releases predictable and auditable.

Kubernetes

Kubernetes manages how containerized applications run, scale, and stay available across multiple machines.

- Applications run inside **Pods**, which contain one or more containers
- **Deployments** define how many replicas of a Pod should run
- **Services** provide stable networking to access Pods
- Kubernetes continuously works to keep the **desired state** running
- Infrastructure details (servers, restarts, failures) are abstracted away

Kubernetes lets teams focus on **what should run**, not **where or how it runs**, which pairs naturally with GitOps and ArgoCD.

Kubernetes Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-application
spec:
  replicas: 1
  selector:
    matchLabels:
      app: example-application
  template:
    metadata:
      labels:
        app: example-application
    spec:
      containers:
      - name: example-application
        image: docker.io/YOUR_USERNAME/
        example-application:v1.0.0
        ports:
        - containerPort: 8080
```

Kubernetes manifests describe the **desired state** of an object, and Kubernetes works continuously to make reality match that definition.

- **kind** is the most important field — it defines *what type of object* this is (Deployment, Service, Pod, etc.)
- **apiVersion** tells Kubernetes which API group manages the object
- Built-in objects (like Deployments) use standard APIs, not CRDs
- Together, apiVersion and kind tell Kubernetes **how to interpret the file**

Before Kubernetes can act, it must first understand *what* you are asking it to create — and kind answers that question.

Kubernetes Service

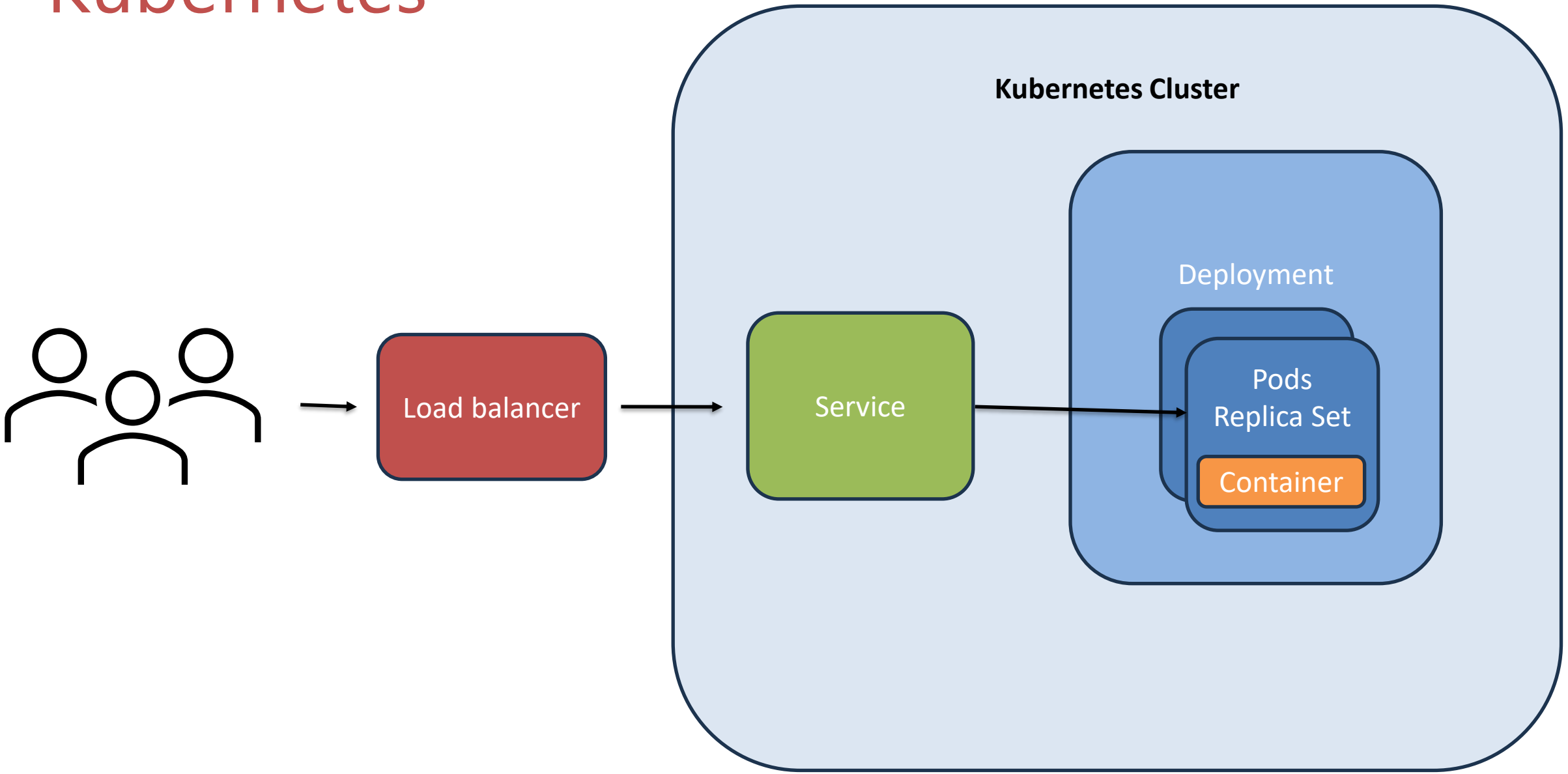
```yaml
apiVersion: v1
kind: Service
metadata:
  name: example-application
spec:
  type: LoadBalancer
  selector:
    app: example-application
  ports:
  - port: 8080
```

A Kubernetes Service provides a **stable network endpoint** for accessing Pods, even as Pods are created, destroyed, or replaced.

- Services expose applications running in Pods
- They provide a **consistent IP and DNS name**
- Traffic is load-balanced across matching Pods
- Defined using a manifest with kind: Service

Services decouple networking from Pods, allowing applications to scale and update without breaking connectivity.

Kubernetes



Kubernetes and ArgoCD

Normally, Kubernetes objects are created and managed through the command line.

- Manifests are applied using commands like kubectl apply
- You can inspect resources with commands such as kubectl get deployments
- Application logs are viewed with commands like kubectl logs deploy/example-deployment

With **ArgoCD**, you simply commit the manifest to a Git repository. ArgoCD detects the change, deploys the object automatically, and provides visibility into application status and logs directly through its UI.

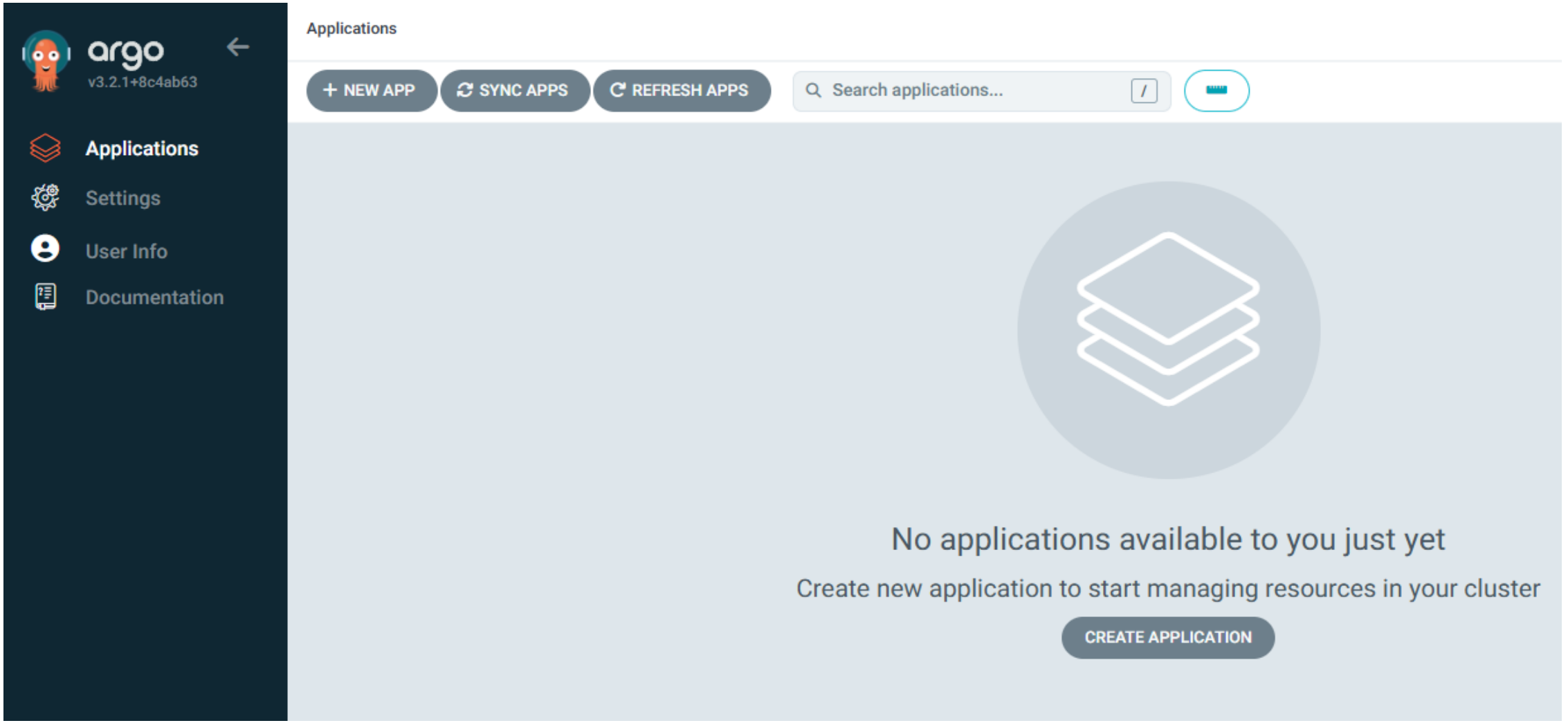
ArgoCD Web Console

ArgoCD provides a web-based UI for managing applications and visualizing GitOps deployments.

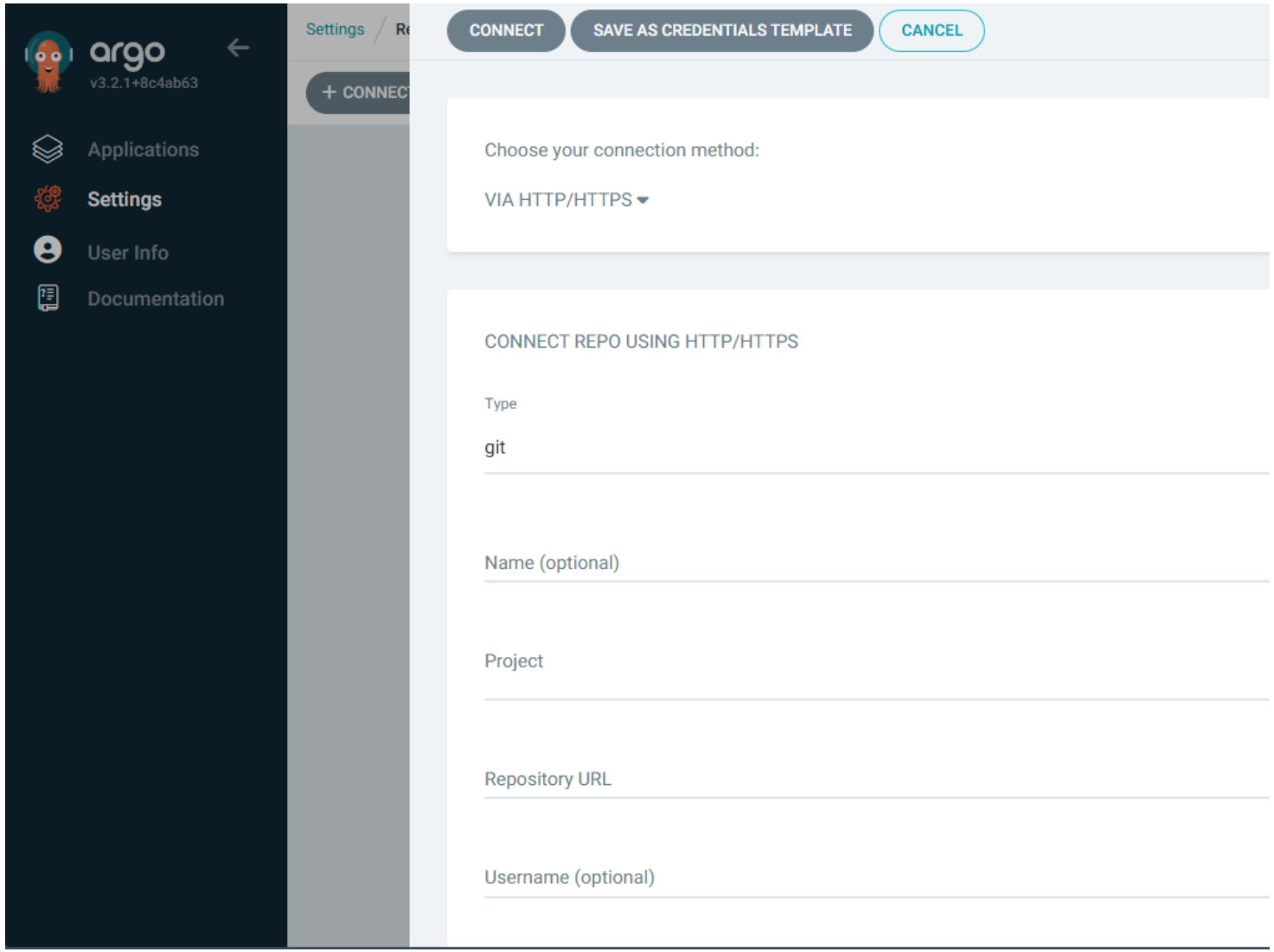
- View application **sync status** (In Sync / Out of Sync)
- Compare **desired state (Git)** vs **live state (cluster)**
- Manually trigger syncs and rollbacks when needed
- Inspect application health and deployment history
- View logs and resource relationships visually

The web console gives teams visibility and control while Git remains the source of truth.

Argo



Argo

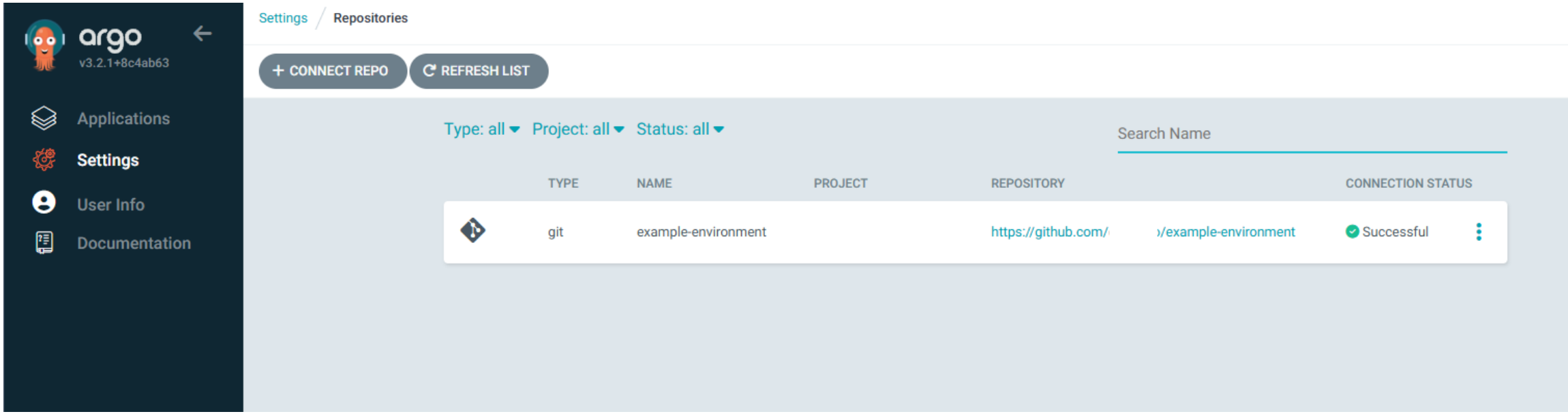


ArgoCD is configured to **read Kubernetes manifests directly from a Git repository**.

- A Git repository is registered with ArgoCD as a source
- ArgoCD is given read access to the repository
- The repository contains Kubernetes manifests, Helm charts, or Kustomize files
- ArgoCD continuously watches the repo for changes

Once connected, Git becomes the source of truth for what ArgoCD deploys to the cluster.

Argo



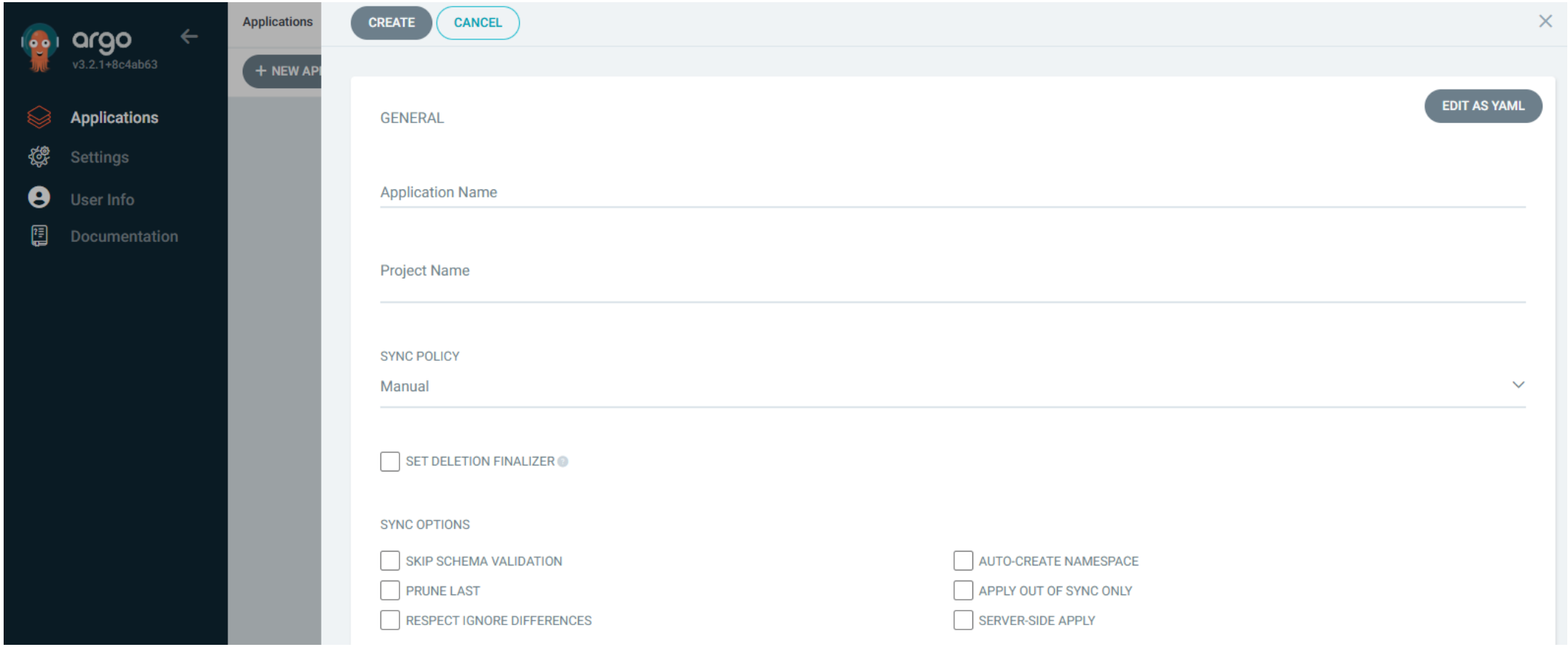
ArgoCD Applications

An ArgoCD **Application** defines *what* to deploy and *where* it should be deployed from Git.

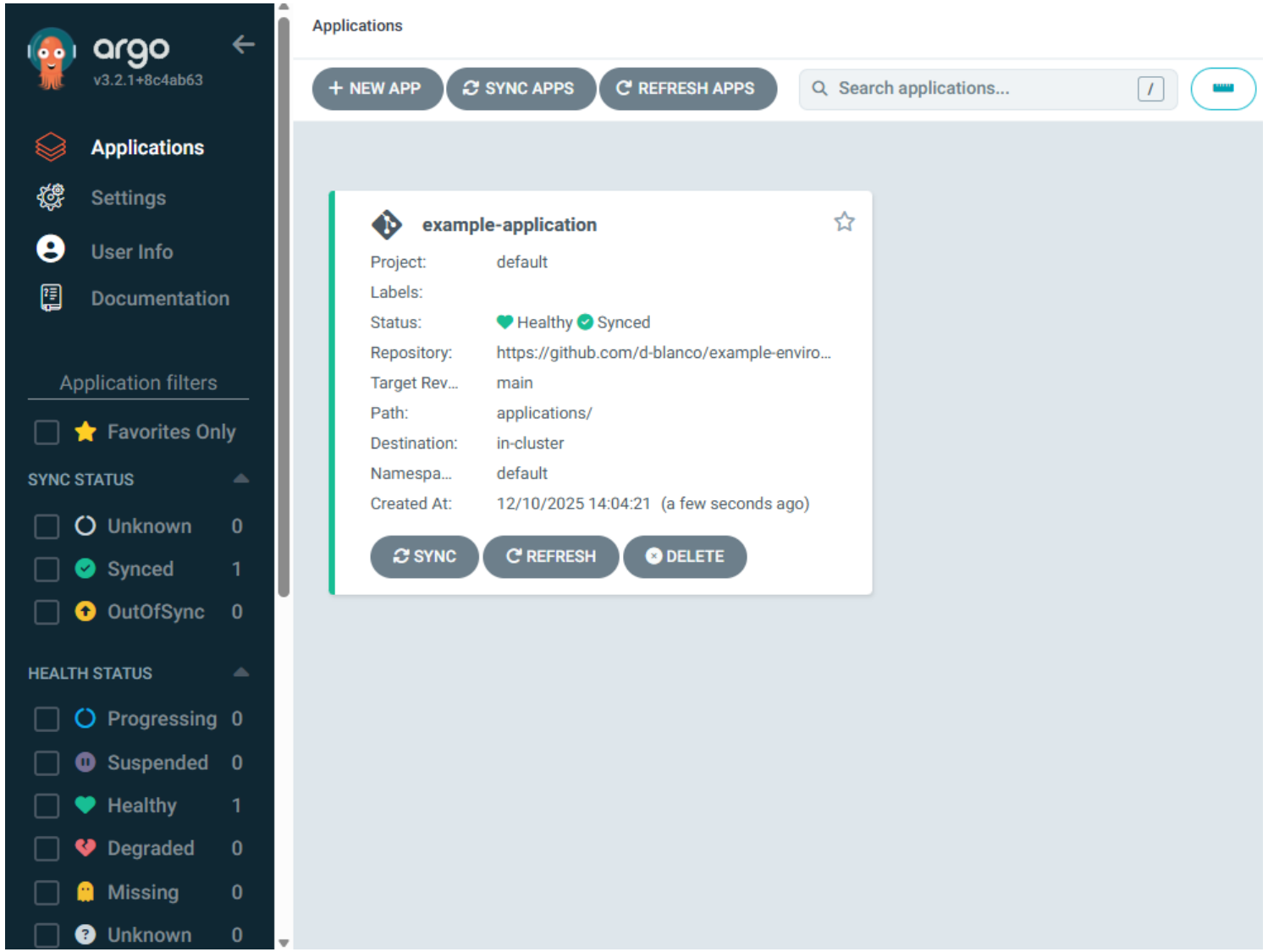
- Specifies the **Git repository** and **folder path** to monitor
- Defines the target cluster and namespace
- Can enable **automatic synchronization**
- Supports self-healing and resource pruning
- Any manifests added to the configured path are deployed automatically

Once an Application is created, ArgoCD continuously ensures the cluster matches what's defined in Gi

Argo



Argo



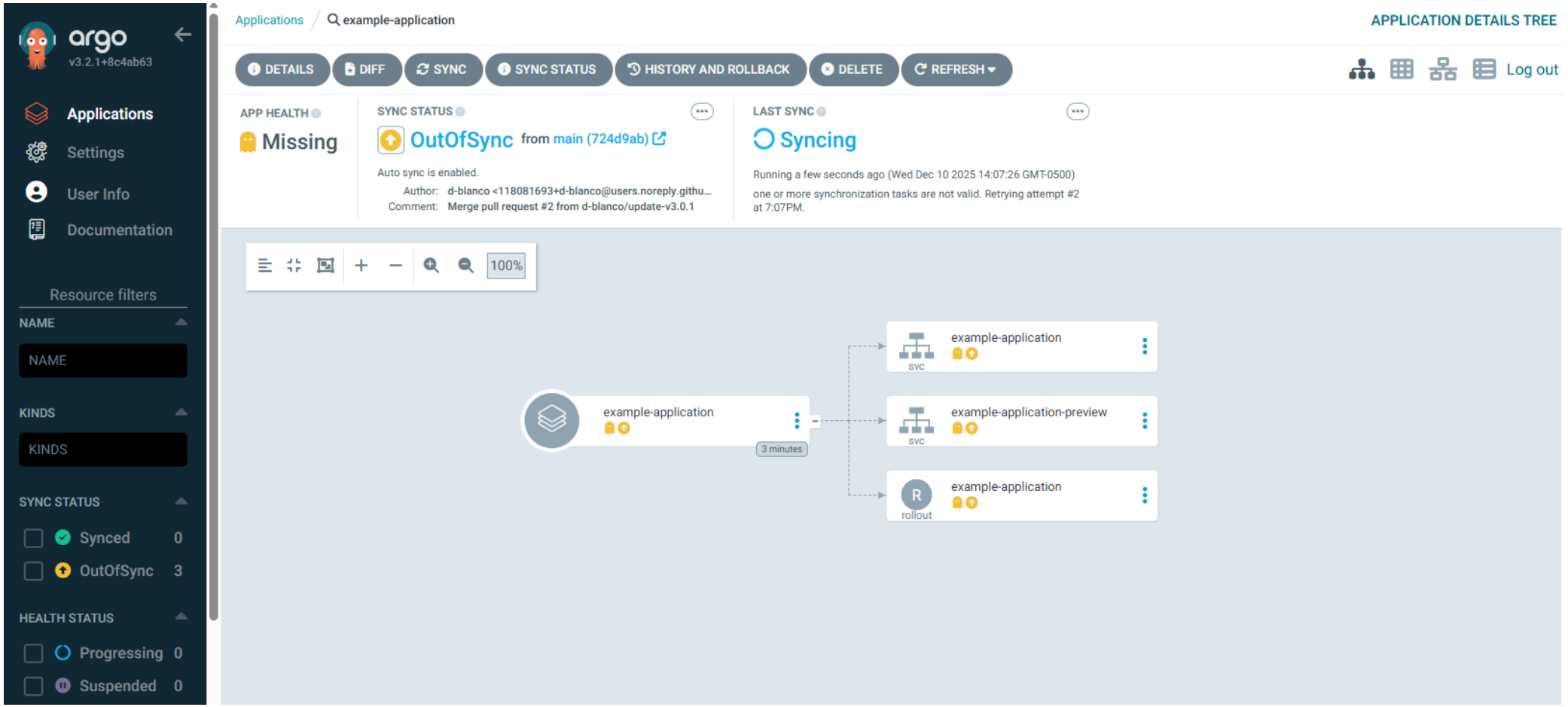
ArgoCD Viewing Resources

ArgoCD provides visibility into all Kubernetes objects managed by an Application.

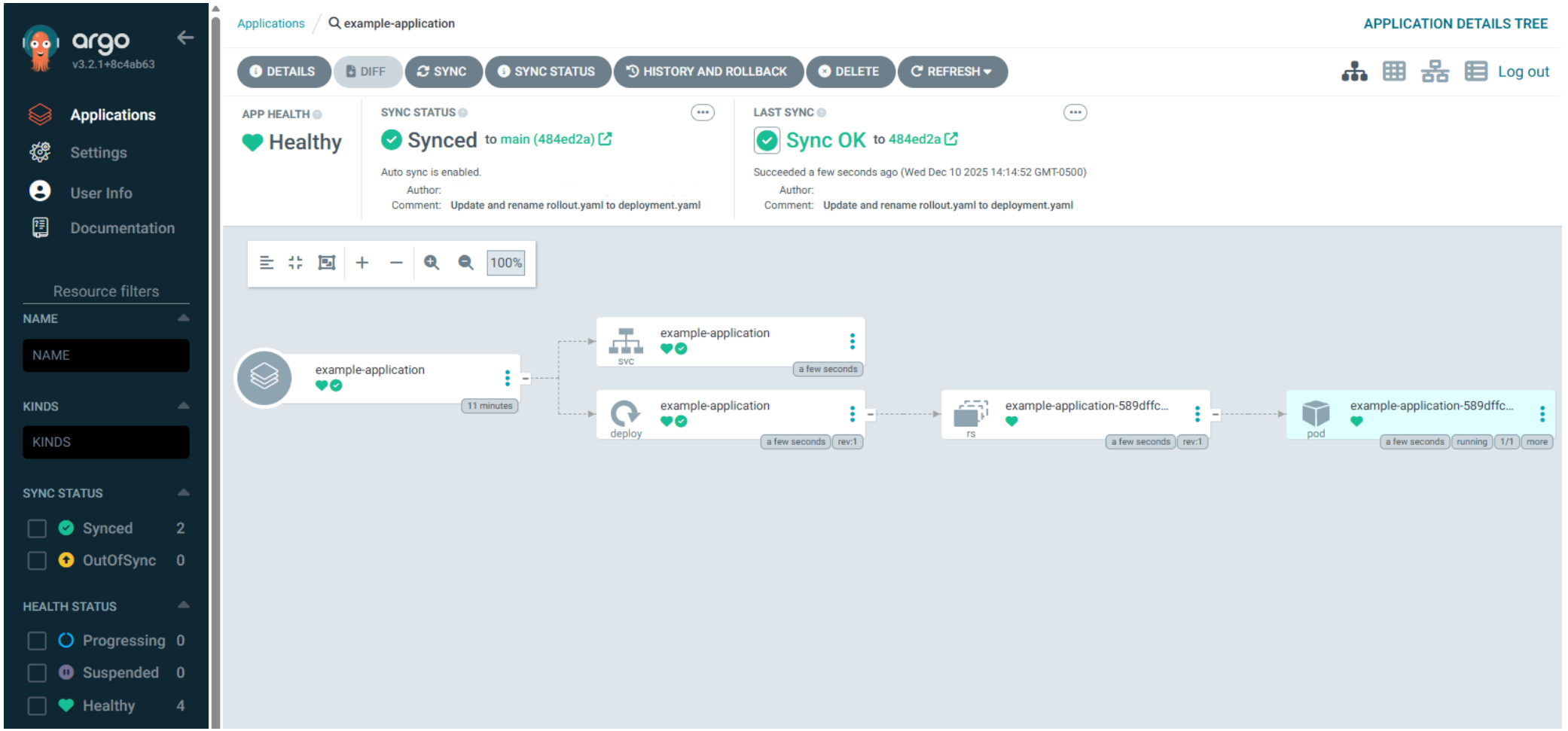
- View all resources created by the Application (Deployments, Services, etc.)
- See real-time **health and sync status** of each object
- Inspect **events** and error messages when something fails
- Access **logs and details** for running workloads
- Easily identify drift between Git and the live cluster

This visibility makes troubleshooting and validating deployments much easier while keeping Git as the source of truth.

Argo Synchronizing



Argo Synchronized App View



Argo Application Pod Logs

POP QUIZ:

What is the core principle of GitOps?

A. Kubernetes replaces CI/CD tools
B. Git is the single source of truth for desired state
C. All deployments must be manual
D. Containers manage infrastructure directly

POP QUIZ:

What is the core principle of GitOps?

A. Kubernetes replaces CI/CD tools
B. **Git is the single source of truth for desired state**
C. All deployments must be manual
D. Containers manage infrastructure directly

POP QUIZ:

In Kubernetes, which field is the most important for determining what type of object a manifest creates?

A. metadata
B. spec
C. apiVersion
D. kind

POP QUIZ:

In Kubernetes, which field is the most important for determining what type of object a manifest creates?

A. metadata
B. spec
C. apiVersion
D. **kind**

POP QUIZ:

Traditionally, how are Kubernetes manifests applied to a cluster?

A. Automatically by Docker
B. Through ArgoCD only
C. Using `kubectl` commands
D. By restarting the cluster

POP QUIZ:

Traditionally, how are Kubernetes manifests applied to a cluster?

A. Automatically by Docker
B. Through ArgoCD only
C. **Using `kubectl` commands**
D. By restarting the cluster

POP QUIZ:

How does ArgoCD deploy Kubernetes manifests differently than traditional methods?

A. It replaces Kubernetes
B. It runs shell commands on nodes
C. It deploys manifests directly from Git repositories
D. It embeds manifests into container images

POP QUIZ:

How does ArgoCD deploy Kubernetes manifests differently than traditional methods?

A. It replaces Kubernetes
B. It runs shell commands on nodes
C. **It deploys manifests directly from Git repositories**
D. It embeds manifests into container images

POP QUIZ:

What enables ArgoCD to perform Continuous Delivery?

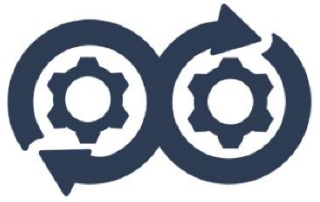
A. Manual kubectl commands
B. Automatic synchronization of Git changes to the cluster
C. Container restarts and polling of git changes
D. Load balancers applying manifest to the cluster

POP QUIZ:

What enables ArgoCD to perform Continuous Delivery?

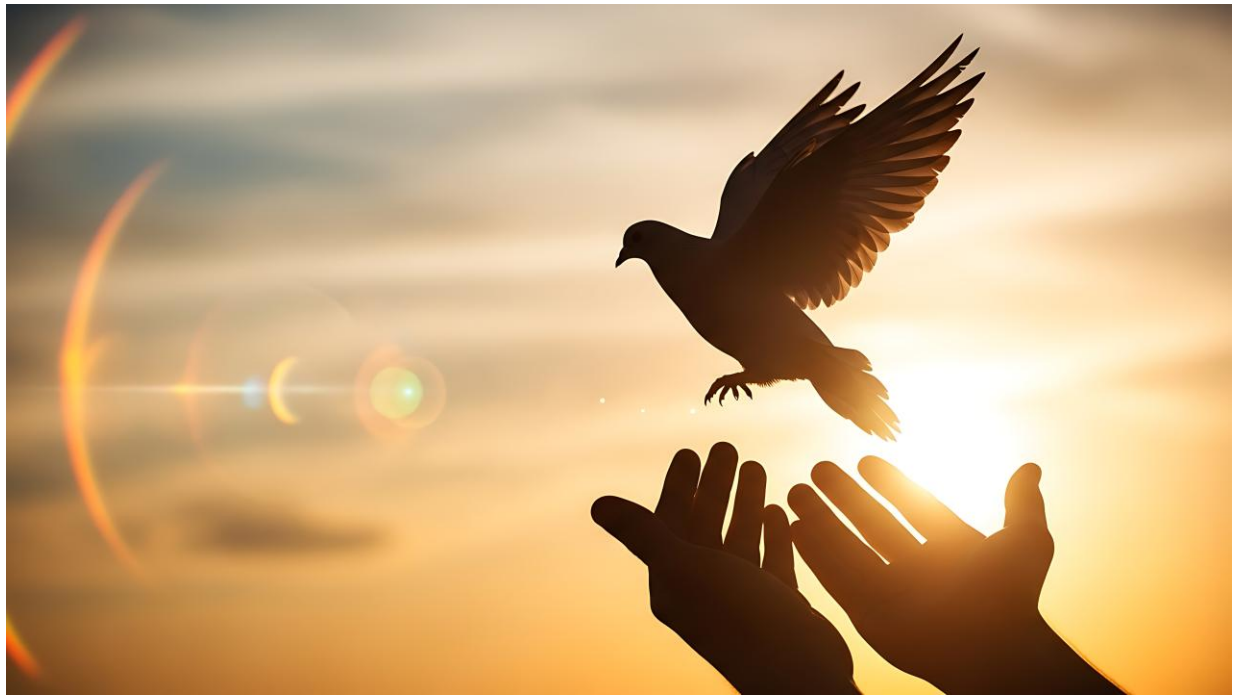
A. Manual kubectl commands
B. **Automatic synchronization of Git changes to the cluster**
C. Container restarts and polling of git changes
D. Load balancers applying manifest to the cluster

Lab: Kubernetes ArgoCD



Kubernetes Automations

Semantic Release



Semantic Release is a versioning and release approach that uses **structured version numbers** to communicate the impact of changes.

- Uses **Semantic Versioning (MAJOR.MINOR.PATCH)**
- **MAJOR:** Breaking changes
- **MINOR:** New features, backward compatible
- **PATCH:** Bug fixes and small improvements

Semantic Release makes deployments predictable, improves rollback safety, and helps teams understand risk before promoting a release.

Semantic Release

Old: V1.0.0
New: V1.0.1

Indicates that upgrading should fix bugs or provide small enhancements with **low risk**

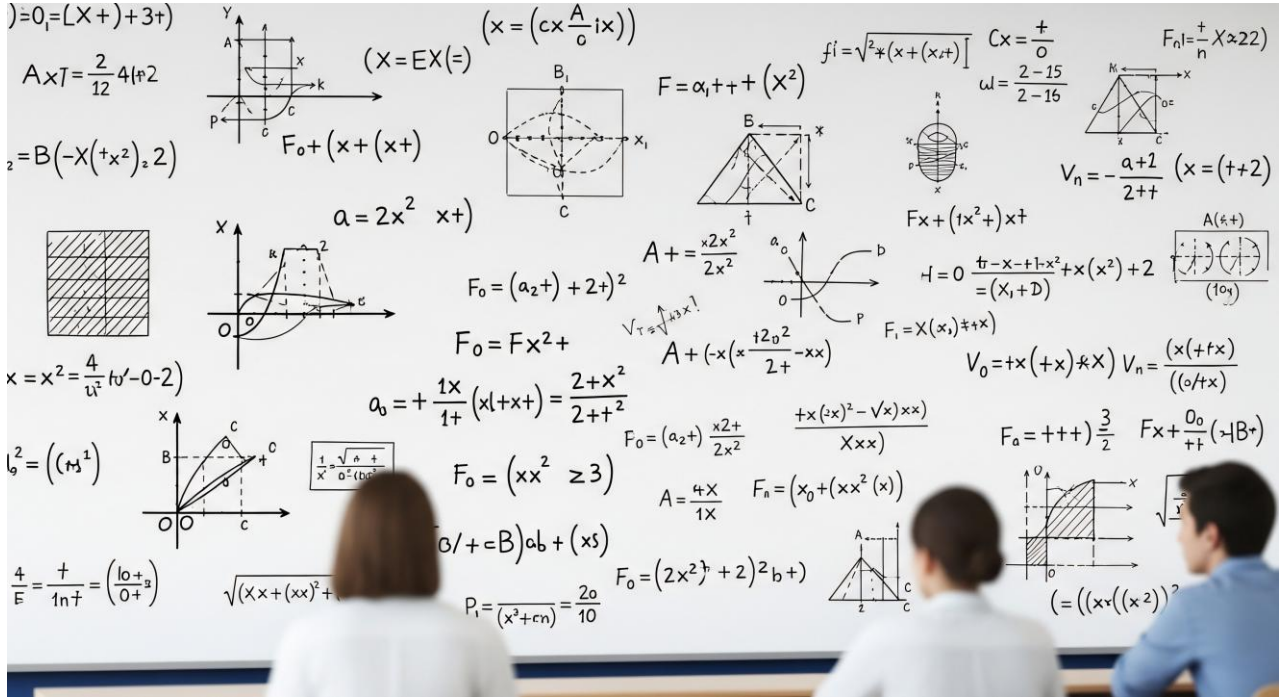
Old: V1.0.1
New: V1.1.0

Indicates that upgrading will add new features but should not require re-configuration. This is **medium risk**.

Old: V1.0.1
New: V2.0.0

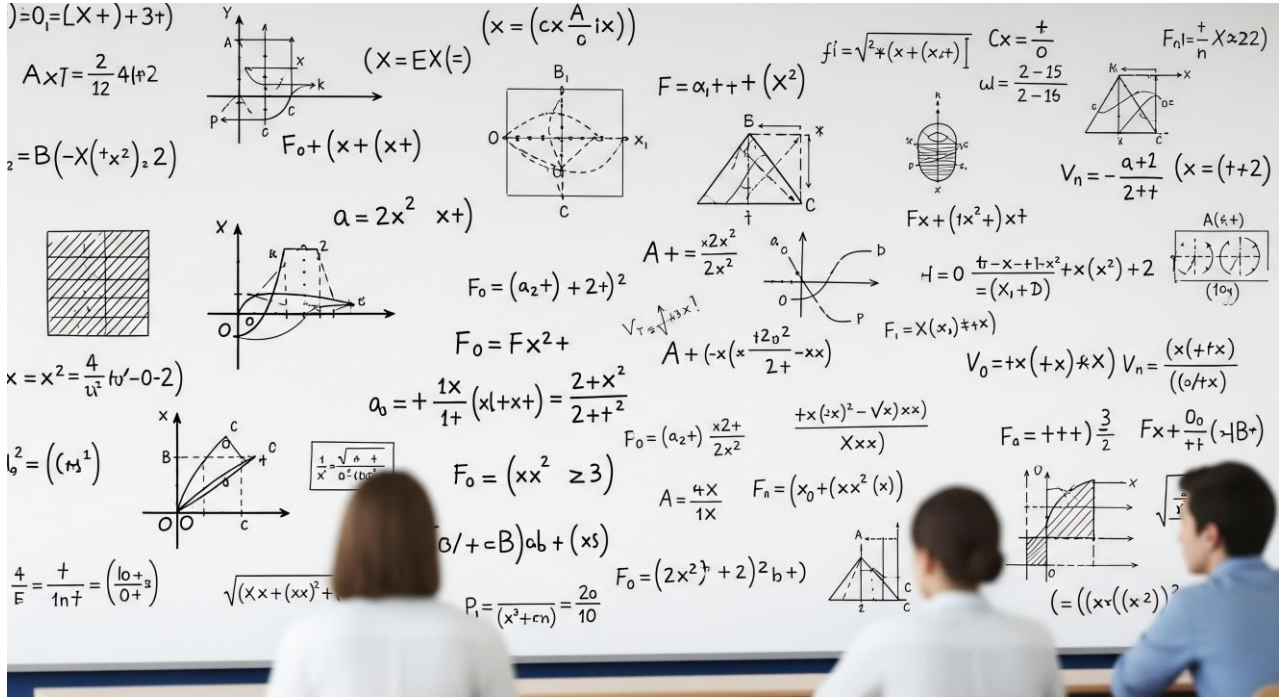
Indicates that upgrading will require different configurations that you will need to test carefully before deployment. This is **high risk**.

Environment Variables



Docker images can define **environment variables** without fixed values, allowing configuration to be supplied when the container starts. They can also be supplied when the image is built.

Environment Variables



- Environment variables can be declared in a **Dockerfile** without values
- Values are provided **at runtime** when the container is launched
- Containers are configurable even though images are read-only
- Commonly used for secrets, connection strings, and feature toggles
- Examples include database passwords or enabling/disabling debug mode

This approach allows the same image to run in multiple environments with different configurations.

Environment Variables

```
FROM python:3.8-alpine
WORKDIR /py-app
COPY . .
RUN pip3 install flask

# Accept a build argument and set
environment variable
ARG APP_VERSION=latest
ENV APP_VERSION=$APP_VERSION

EXPOSE 8080
CMD ["python3", "main.py"]
```

```
import os
from flask import Flask

app = Flask(__name__)
version = os.getenv("APP_VERSION", "unknown")

@app.route('/')
def index():
    return f'Hello Argo CD {version}!'

app.run(host='0.0.0.0', port=8080)
```

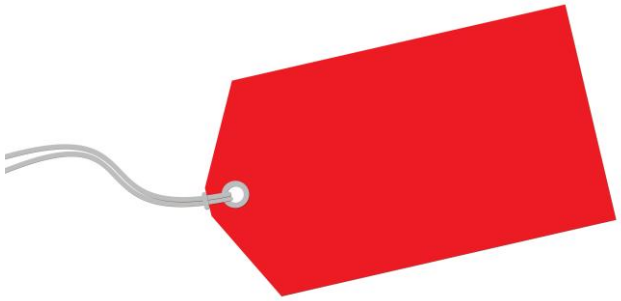
```
- name: Build and Push
  uses: docker/build-push-action@v5
  with:
    context: .
    push: true
    tags: docker.io/${{ secrets.REGISTRY_USER
}}/example-application:${{ env.TAG_NAME }}
    build-args: |
      APP_VERSION=${{ env.TAG_NAME }}
```

In this example, APP_VERSION is used as **both a build argument and an environment variable**.

- APP_VERSION is defined as a **build argument**, so it **must be provided at build time**
- The CI/CD workflow supplies this value from its own environment variables
- The Dockerfile then exposes it as an **environment variable inside the container**
- The application reads it at runtime using os.getenv() and displays it in responses

This pattern allows version information to be injected during the build while still being accessible to the running application.

Tagging Each Image



•ArgoCD supports automatic image updates through an optional **Image Updater**, enabling true continuous deployment.

- **ArgoCD Image Updater** monitors container registries for new image tags that match defined patterns (e.g., dev, uat, prod)
- It requires installing a **CRD (Custom Resource Definition)** to define image update behavior
- When a new image is detected, the Image Updater **modifies the Git repository** by updating the image tag
- ArgoCD then detects the Git change and synchronizes the deployment

In the upcoming lab, we achieve the same outcome using a **CI/CD pipeline** that updates the image tag in Git (via sed). While simpler and less robust than ArgoCD Image Updater, it demonstrates the same GitOps principle: **Git changes drive deployments**.

POP QUIZ:

What is the primary purpose of Docker environment variables?

A. To modify the container image at runtime
B. To configure application behavior without rebuilding the image
C. To manage Kubernetes networking
D. To store secrets permanently inside the image

POP QUIZ:

What is the primary purpose of Docker environment variables?

A. To modify the container image at runtime
B. **To configure application behavior without rebuilding the image**
C. To manage Kubernetes networking
D. To store secrets permanently inside the image

POP QUIZ:

When are Docker environment variable values typically provided?

A. Only at image build time
B. Only inside the application code
C. At container runtime
D. When the Docker daemon starts

POP QUIZ:

When are Docker environment variable values typically provided?

A. Only at image build time
B. Only inside the application code
C. **At container runtime**
D. When the Docker daemon starts

POP QUIZ:

What does Semantic Versioning communicate to teams?

A. The deployment environment
B. The infrastructure cost
C. The impact and risk of a release
D. The container runtime

POP QUIZ:

What does Semantic Versioning communicate to teams?

A. The deployment environment
B. The infrastructure cost
C. **The impact and risk of a release**
D. The container runtime

POP QUIZ:

What best describes Continuous Deployment?

A. Deployments triggered manually by engineers
B. Deployments that require change approval
C. Automatic deployment of every approved change
D. Deployments only during maintenance windows

POP QUIZ:

What best describes Continuous Deployment?

A. Deployments triggered manually by engineers
B. Deployments that require change approval
C. **Automatic deployment of every approved change**
D. Deployments only during maintenance windows

POP QUIZ:

In a GitOps model, how are new image versions typically deployed?

A. By manually editing running Pods
B. By restarting Kubernetes nodes
C. By pushing images directly into the cluster
D. By updating image tags in Git and synchronizing

POP QUIZ:

In a GitOps model, how are new image versions typically deployed?

A. By manually editing running Pods
B. By restarting Kubernetes nodes
C. By pushing images directly into the cluster
D. **By updating image tags in Git and synchronizing**

POP QUIZ:

What is the main advantage of updating image tags in Git rather than directly in the cluster?

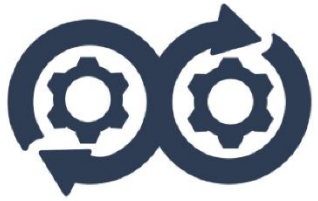
A. Git remains the source of truth and changes are auditable
B. It improves container startup time
C. It removes the need for CI/CD
D. Kubernetes applies changes faster

POP QUIZ:

What is the main advantage of updating image tags in Git rather than directly in the cluster?

A. **Git remains the source of truth and changes are auditable**
B. It improves container startup time
C. It removes the need for CI/CD
D. Kubernetes applies changes faster

Lab: Kubernetes Automations



Congratulations

Today you learned how to automate infrastructure at a production level:

- How **regions, AZs, ASGs, and load balancers** create resilient architectures
- Why **immutable deployments** are safer and more consistent
- How **Packer builds Golden AMIs** ready to run your application on boot
- How Terraform can **automatically deploy the latest AMIs**
- How ASGs perform **rolling updates** to keep services online

You're now working with the same patterns used in real enterprise CI/CD pipelines.