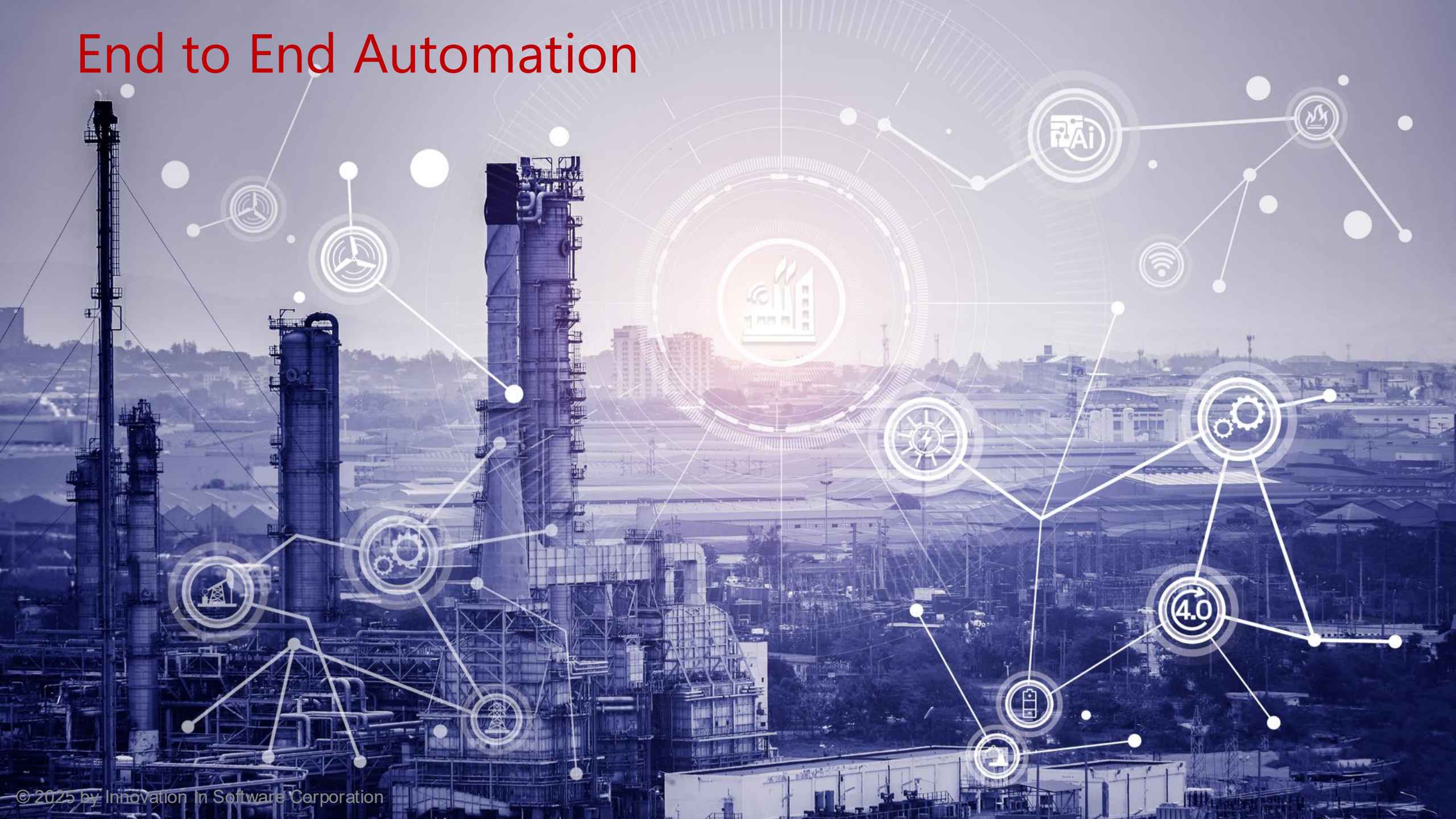
End to End Automation



TEKsystems Global Services

WORKFORCE DEVELOPMENT

Content Usage Parameters

Content refers to material including instructor guides, student guides, lab guides, lab or hands-on activities, computer programs, etc. designed for use in a training program

1

Content is subject to copyright protection

2

Content may only be leveraged by students enrolled in the training program

3

Students agree not to reproduce, make derivative works of, distribute, publicly perform and publicly display in any form or medium outside of the training program

4

Content is intended as reference material only to supplement the instructor-led training

Kubernetes POD Scheduling

Pod Scheduling Overview

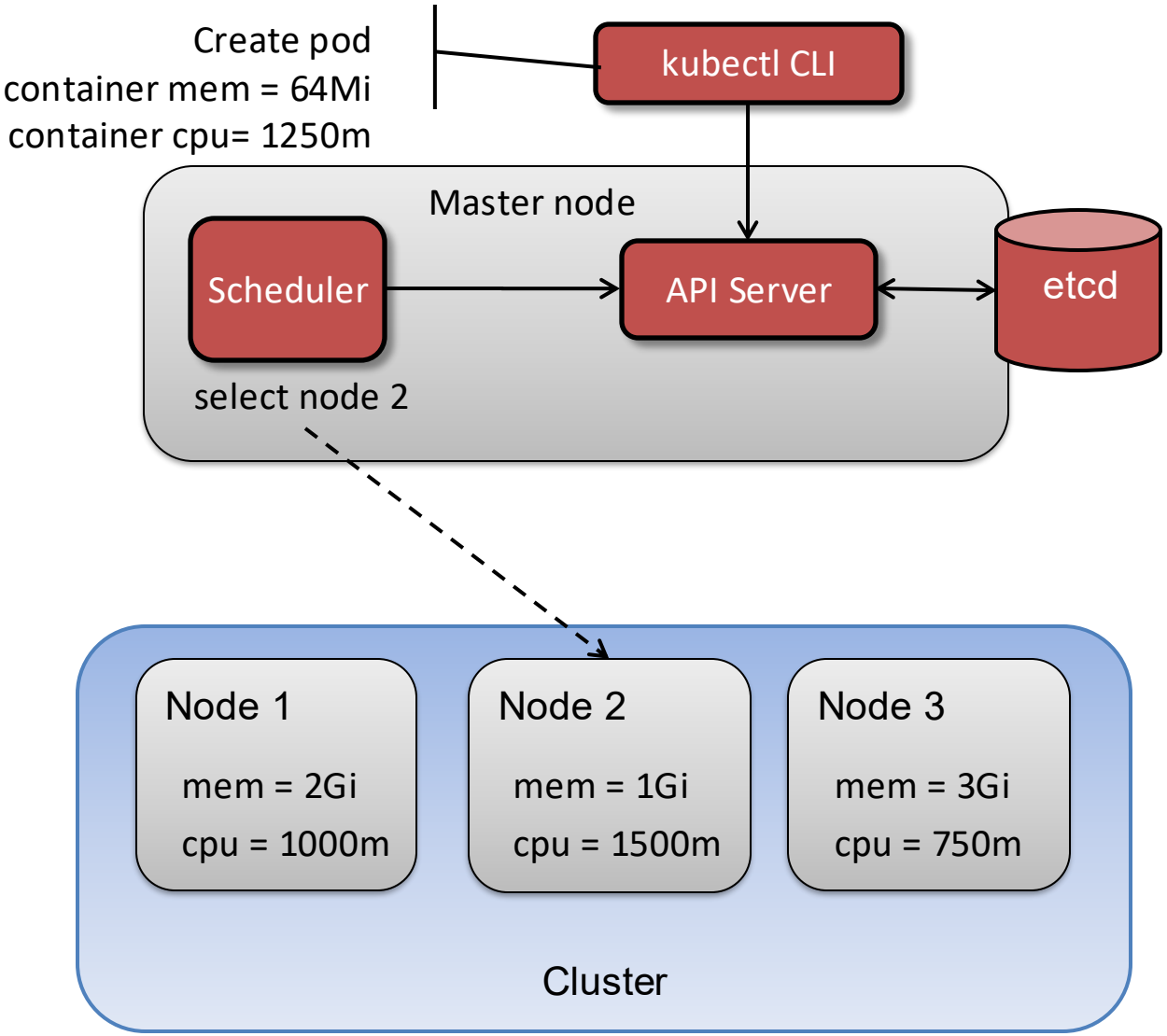
- kube-scheduler on the master node assigns new pod requests to appropriate worker nodes
- Default scheduler takes account of
  - Available node CPU/RAM
  - Resource requests from new pod – sum of resource requests of pod containers

Default scheduler will automatically

Schedule pod on node with sufficient free resources

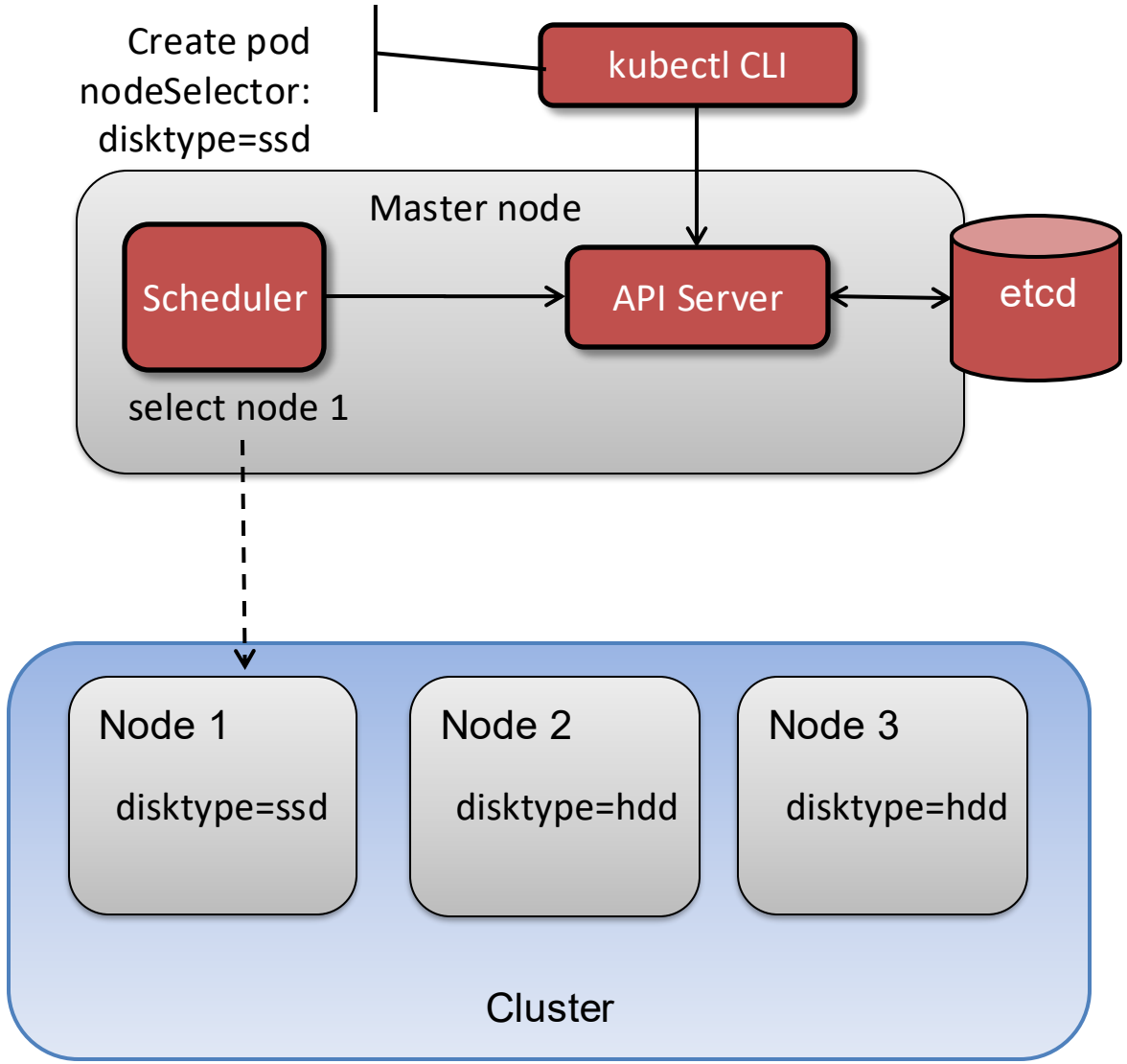
Spread pods across nodes in the cluster

Can specify custom schedulers for pods



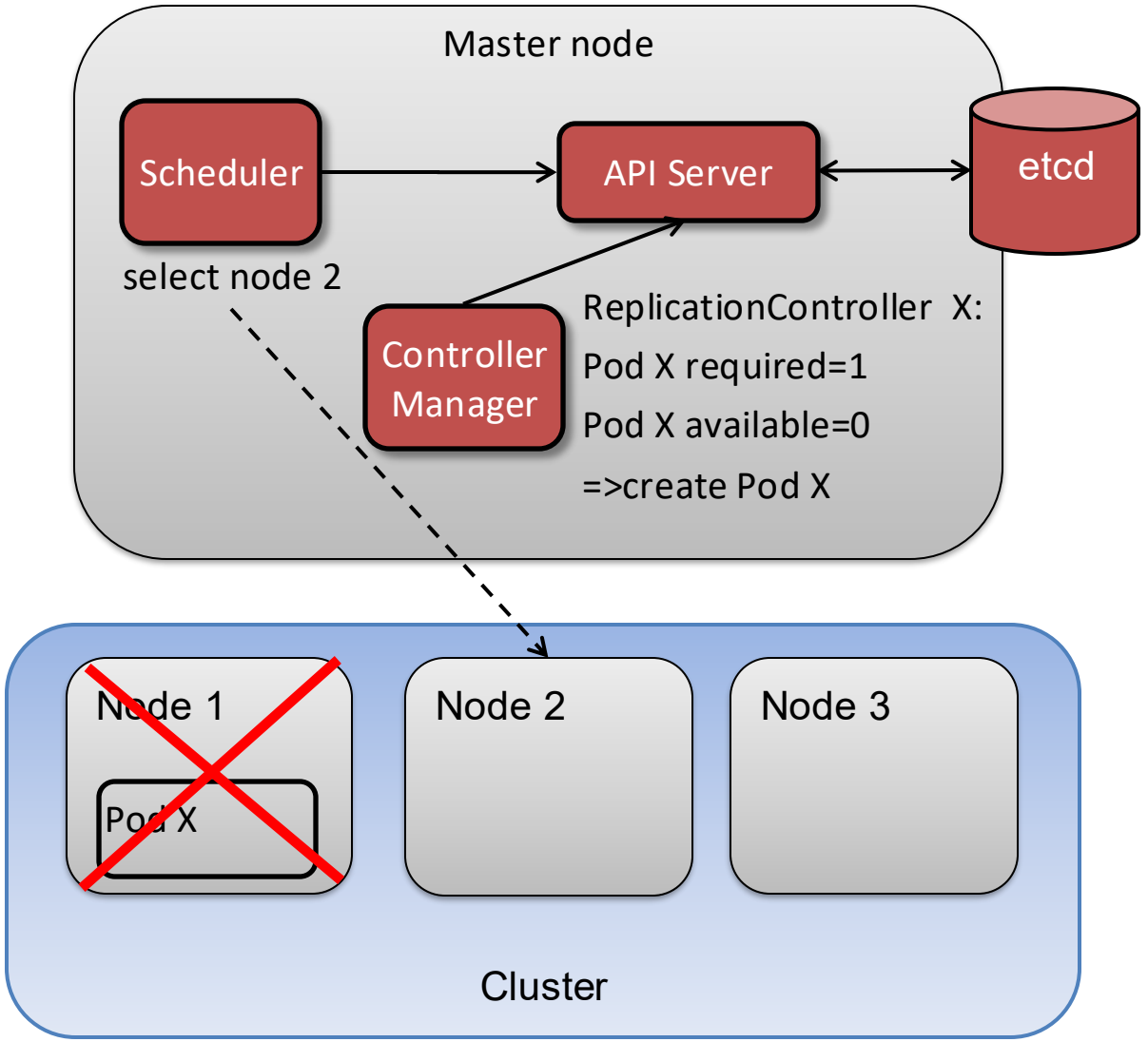
Controlling Pod Scheduling with Default Scheduler

- Nodes carry labels indicating topology and other resource notes
- Users can require pods to be scheduled on nodes with specific label(s) via a nodeSelector in container spec



Scheduling Pod Re-creation Driven by Control Loops

- kube-scheduler performs same node selection operation when new pod created due to e.g. node loss
- kube-controller-manager runs controllers like ReplicationController managing number of pod instances available
- kube-controller-manager will initiate request for new pods as needed, which will be scheduled by kube-scheduler per pod/container spec



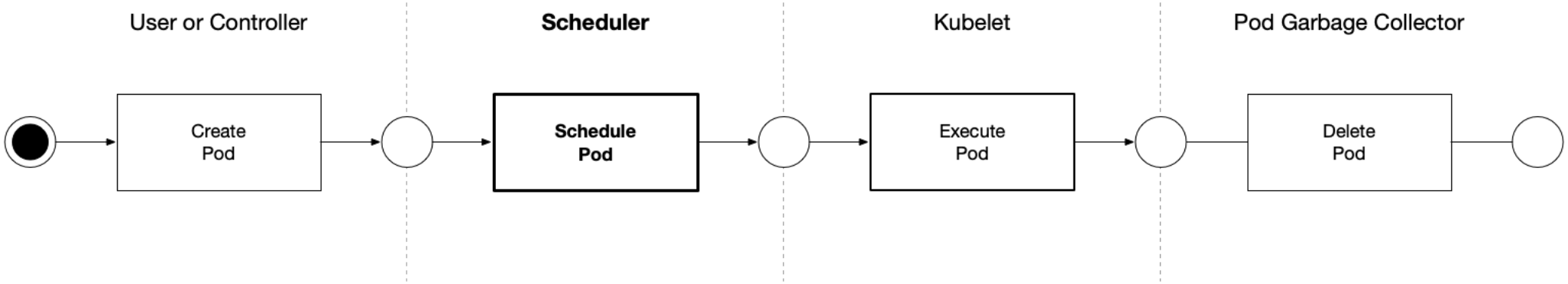
Kubernetes Scheduling

Node Selector

```
apiVersion: v1
kind: Pod
metadata:
      name: nginx
      labels:
          env: test
spec:
    container:
    - name: nginx
       image: nginx
       nodeSelector:
          disktype: ssd
```

Scheduling flow

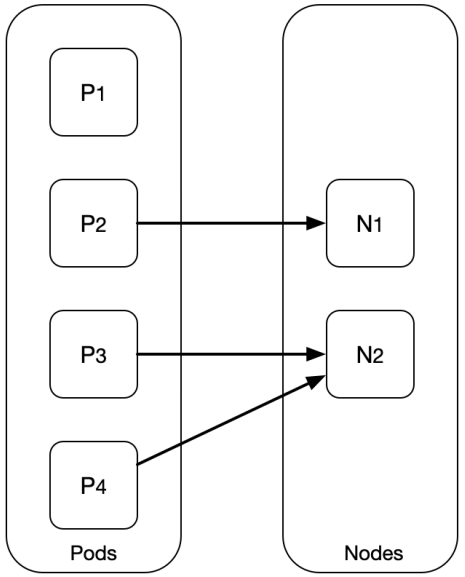
- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on, this is called a "placement"
  - Placement: a partial, non-injective assignement of a set of Pods to a set of Nodes.



Scheduler details

- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on, this is called a "placement"
  - Placement: a partial, non-injective assignement of a set of Pods to a set of Nodes.

Schedule $\in$ Pods $\nrightarrow$ Nodes



Partial

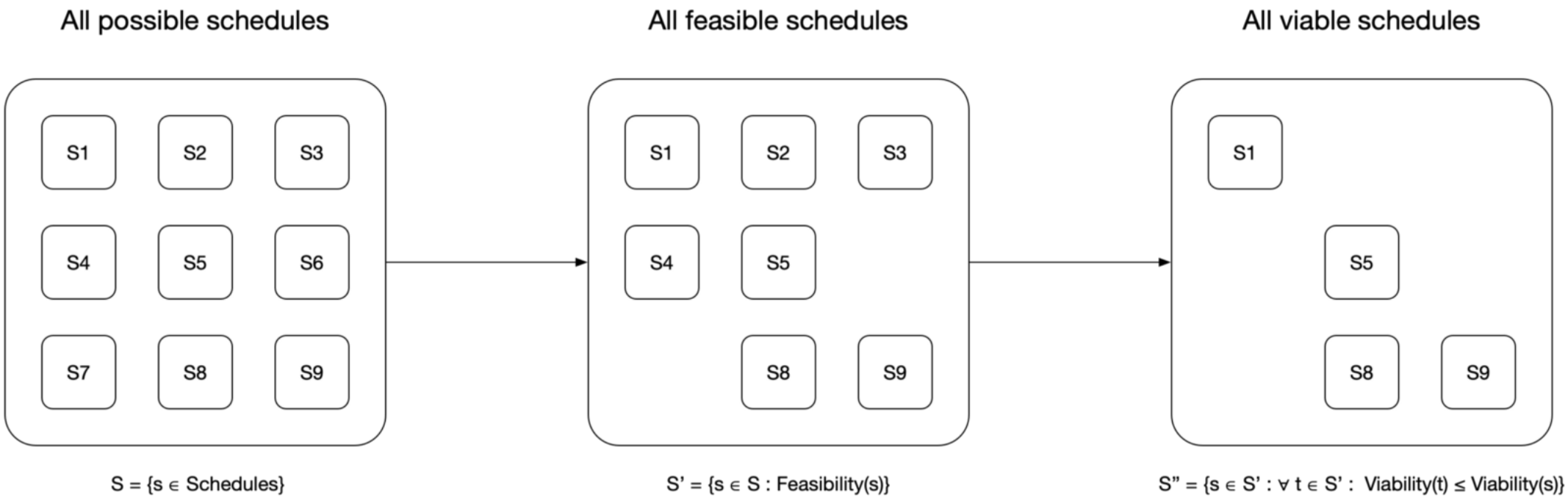
Some Pods may not be assigned a Node

Non-Injective

Some Pods may be assigned the same Node

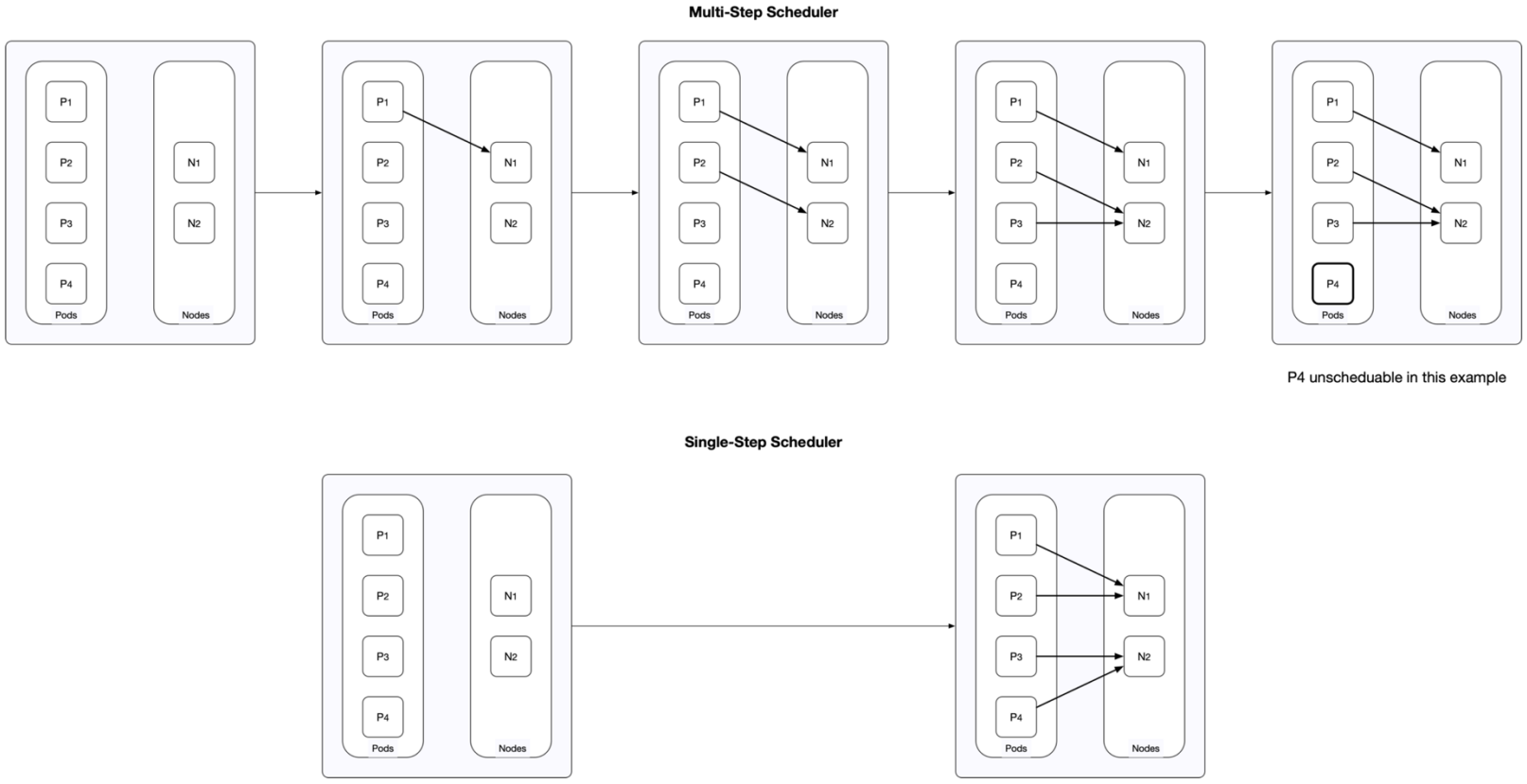
Scheduler decisions

- **Scheduler (kube-scheduler)**: selects nodes for newly created pods to run on, this is called a "placement"
  - Placement: a partial, non-injective assignement of a set of Pods to a set of Nodes.

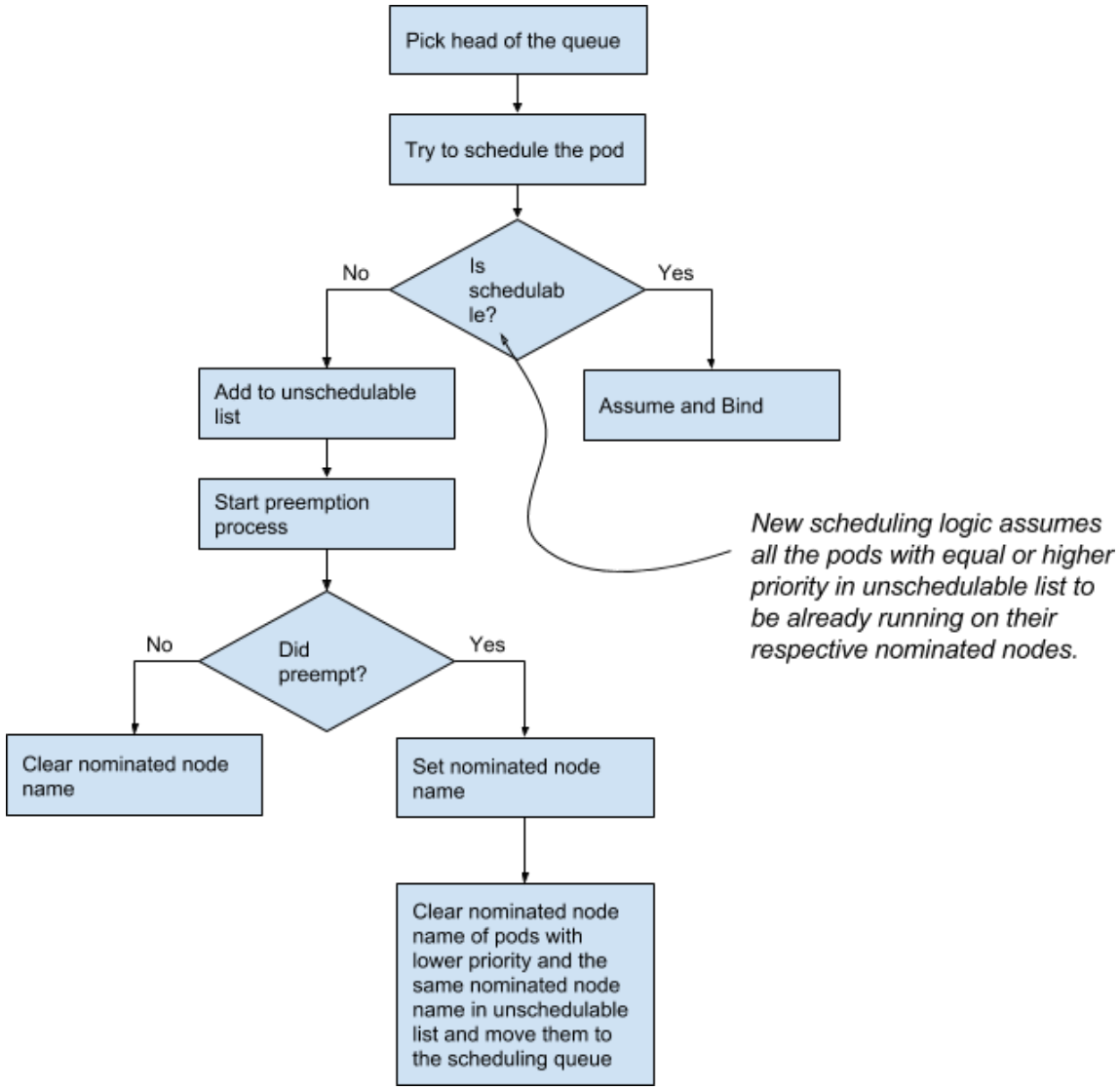


Scheduler decisions

- **Multi-step scheduler**
  - local optimum instead of global optimum

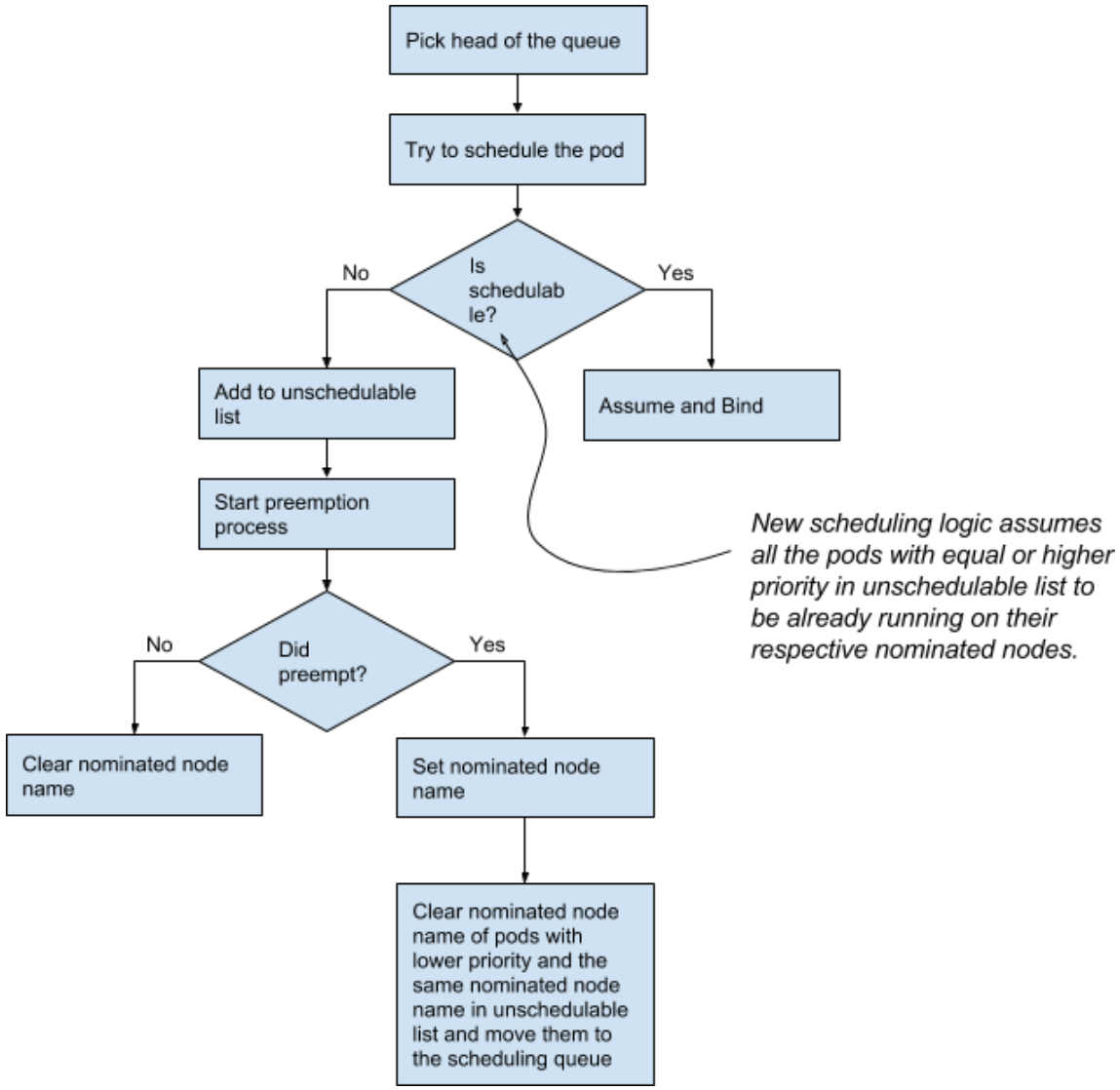


Scheduler preemption

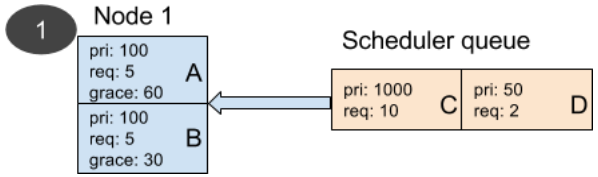


- **Preemption**
  - Lower priority Pods are destroyed to free up resources for higher priority Pods.

Scheduler preemption



- **Example**
  - 1 node in the cluster with capacity 10 units
  - 2 pods (A,B) running on the node with priority 100 and each using 5 units
  - Scheduler has 2 pods (C,D) in queue. Pod C needs 10 units and priority is 1000. Pod D needs 2 units and it's priority is 50
  - Scheduler determines that Pod C has highest priority and destroys (A,B) so it can schedule.



Kubernetes Advanced POD Scheduling

Kubernetes Scheduling

- Node Selector
- Node Affinity
- Pod Affinity
- Pod Anti Affinity
- 3rd party schedulers

Kubernetes Scheduling

- Node affinity/anti-affinity
  - generalizing the nodeSelector feature
- Node taints
  - prevent scheduling of pods to nodes unless pod ‘tolerates’ the taint
- Pod affinity/anti-affinity
  - control relative placement of pods

Node Affinity

The affinity/anti-affinity feature greatly expands the types of constraints you can express.

The key enhancements are:

- More expressive language (not just “AND of exact match”)
- Rule is “soft”/”preference” so if it doesn’t match Pods are still scheduled.

Kubernetes Scheduling

Node Affinity

```
affinity:
 nodeAffinity:
   requiredDuringSchedulingIgnoredDuringExecution:
     nodeSelectorTerms:
       - matchExpressions:
         - key: "failure-domain.beta.kubernetes.io/zone"
           operator: In
           values: ["us-central1-a"]
```

Kubernetes Scheduling

Node Affinity

```
affinity:
 nodeAffinity:
   preferredDuringSchedulingIgnoredDuringExecution:
     nodeSelectorTerms:
       - matchExpressions:
         - key: "failure-domain.beta.kubernetes.io/zone"
           operator: In
           values: ["us-central1-a"]
```

Kubernetes Scheduling

Weight example

```
affinity:
 nodeAffinity:
   preferredDuringSchedulingIgnoredDuringExecution:
   - weight: 5
     preference:
       - matchExpressions:
         - key: env
           operator: In
           values:
           - dev
    preferredDuringSchedulingIgnoredDuringExecution:
   - weight: 1
     preference:
       - matchExpressions:
         - key: team
           operator: In
           values:
           - engineering
```

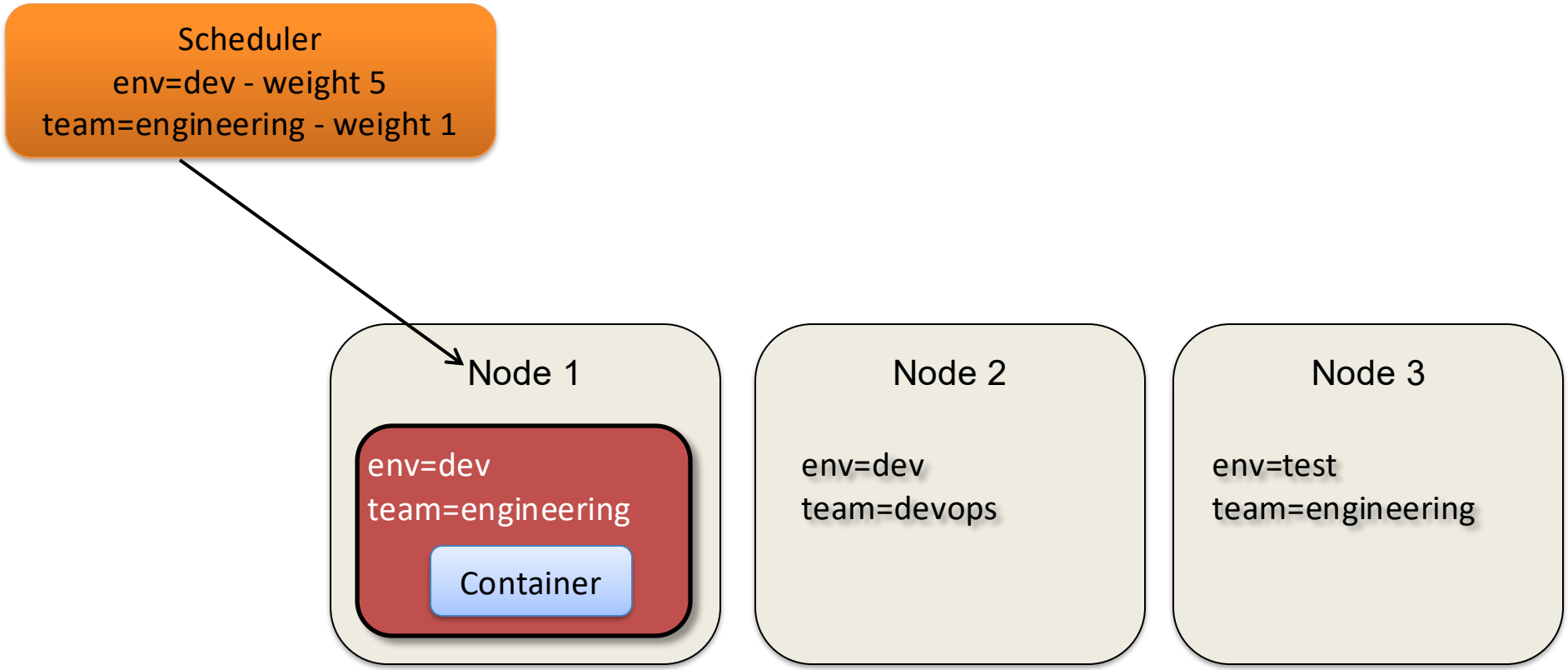
Node Affinity

Kubernetes supports "weighted scheduling"

- The higher this weighting the more weight is given to that rule.
- Scoring is done by summarizing weighting per node.
  - 2 rules weights 1 and 5
  - If both match = score of 6
  - If only 1 rule matches = score of 1
- Node with highest total weight score receives Pod.

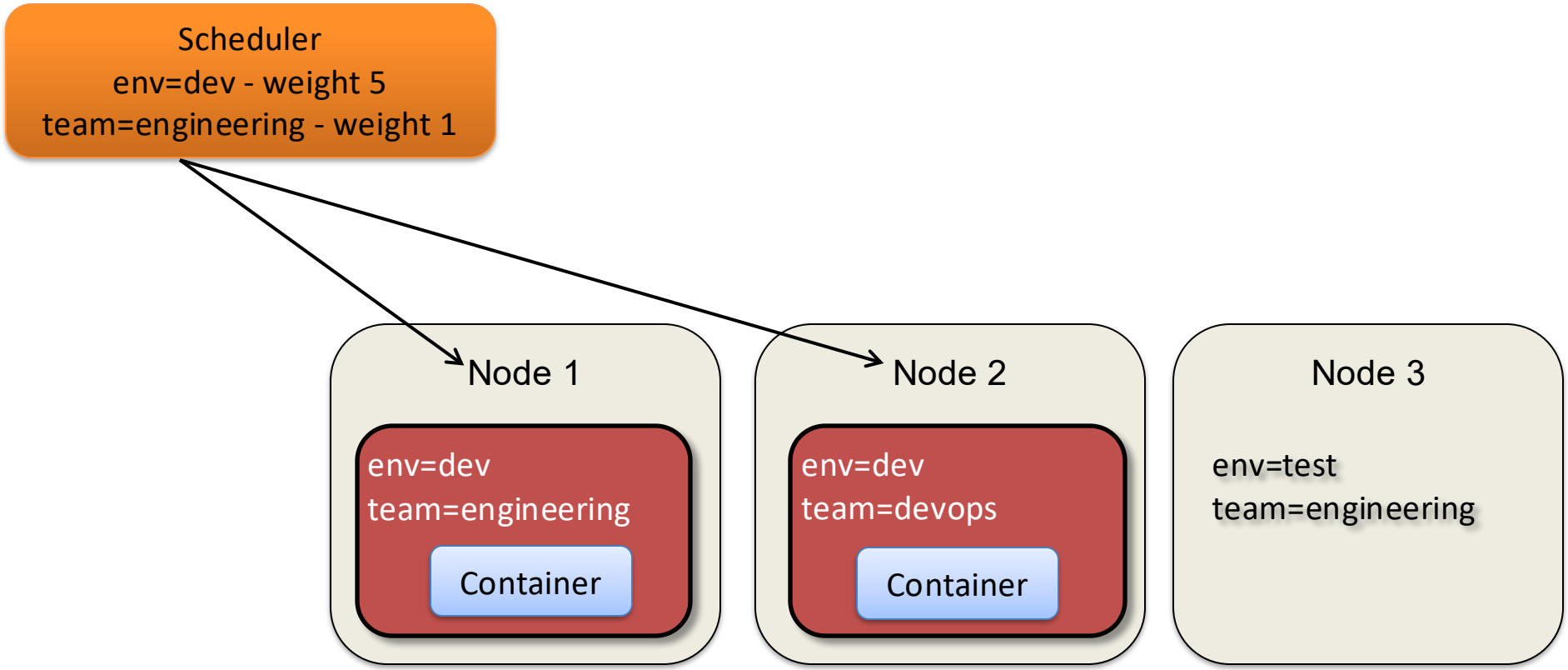
Node Affinity: Weight

- Schedule Pods onto nodes that have labels
  - env = dev
  - team = engineering



Node Affinity: Weight

- Schedule Pods onto nodes that have labels
  - env = dev
  - team = engineering



Operators

Valid operators

- In
- NotIn
- Exists
- DoesNotExist
- Gt (Greater Than)
- Lt (Less Than)

Node Taints

Allows you to mark (“taint”) a node

- No pods can be scheduled onto tainted nodes
- Useful when all or most PODs should not be scheduled on a node
- Mark your master node as schedulable only by Kubernetes system components
- Keep regular pods away from nodes that have special hardware so as to leave room for pods that need the special hardware.

Node Tolerations

To allow a POD to be scheduled onto a ‘tainted’ node it must have:

```
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"
  effect: "NoSchedule"
```

Pod Affinity

- Define how pods should be placed relative to one another
- Spread or pack pods within a service or relative to pods in other services

```
affinity:
   podAffinity:
     requiredDuringSchedulingIgnoredDuringExecution:
     - labelSelector:
         matchExpressions:
         - key: service
           operator: In
           values: [“S1”]
       topologyKey: failure-
domain.beta.kubernetes.io/zone
```

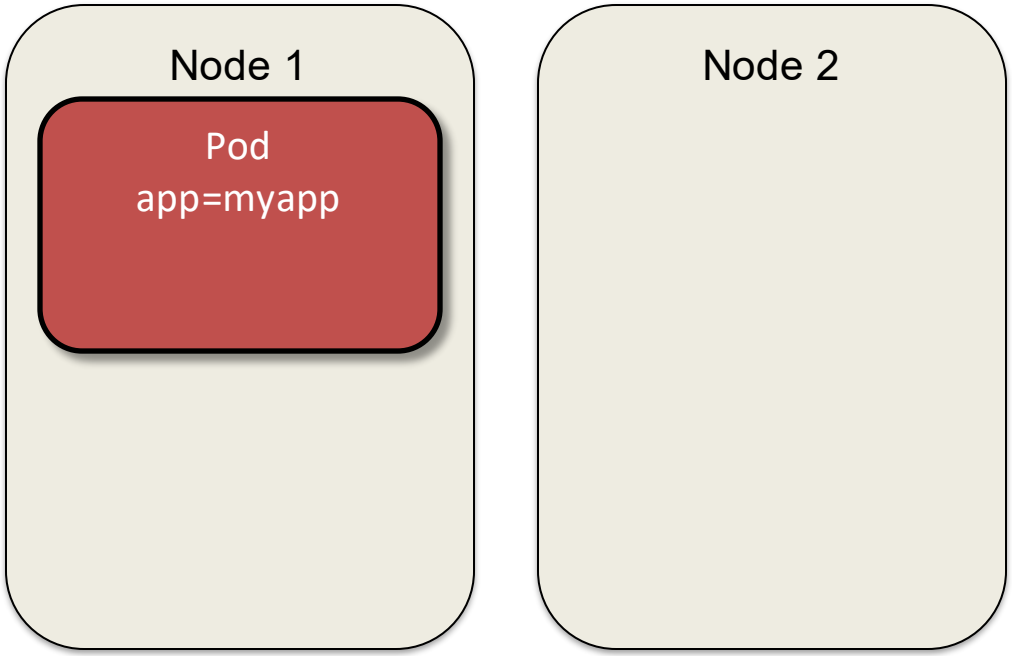
Pod Affinity

- Define how pods should be placed relative to one another
- Spread or pack pods within a service or relative to pods in other services

```
affinity:
   podAffinity:
     preferredDuringSchedulingIgnoredDuringExecution:
     - labelSelector:
         matchExpressions:
         - key: service
           operator: In
           values: [“S1”]
       topologyKey: failure-
domain.beta.kubernetes.io/zone
```

Pod Affinity

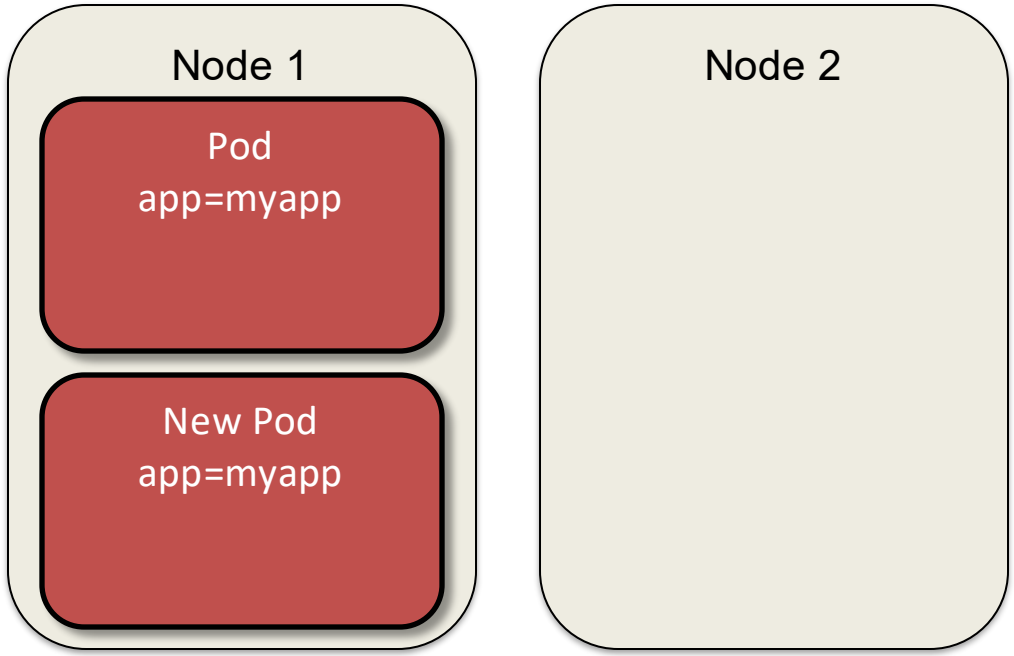
- Schedule Pods onto nodes that have same labels
  - app=myapp
  - operator: In
  - kubernetes.io/hostname




```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: "app"
          operator: In
          values:
          - myapp
      topologyKey: "kubernetes.io/hostname"
```

Pod Affinity

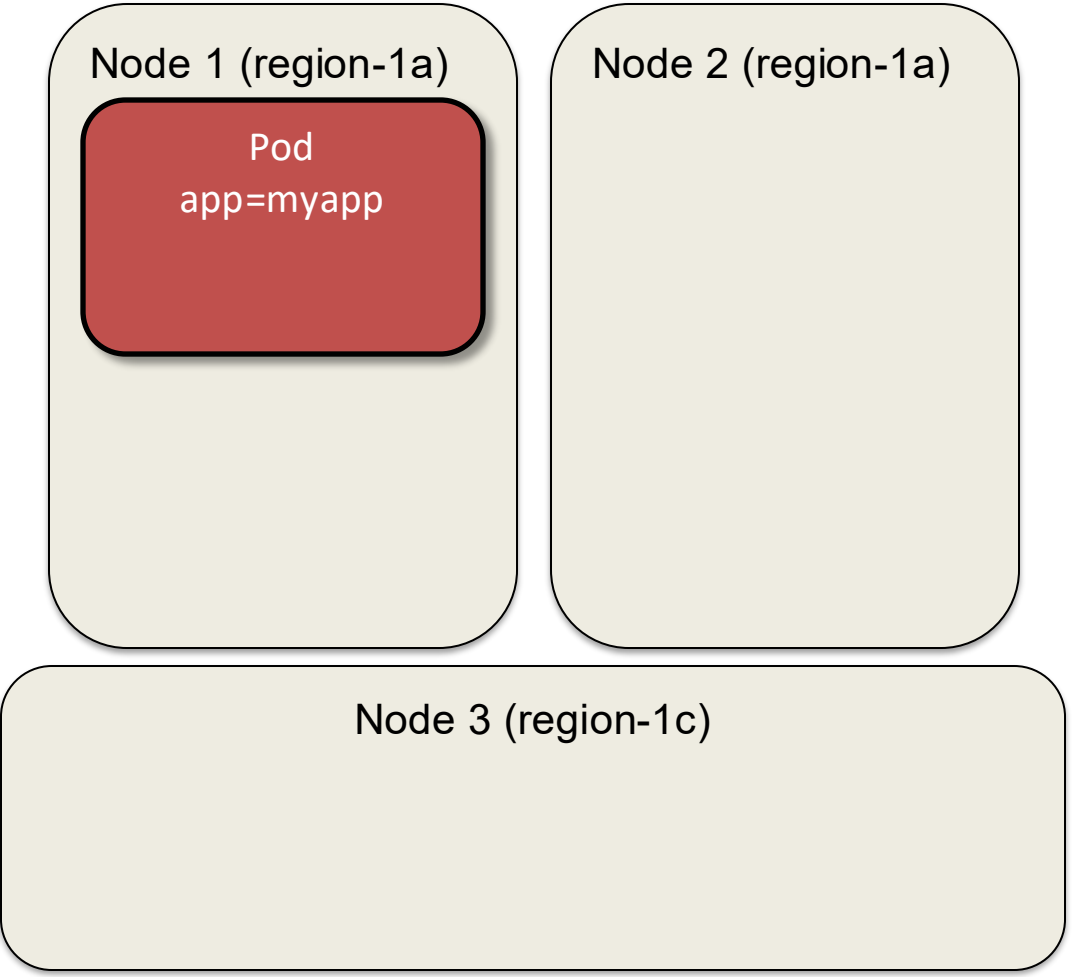
- Schedule Pods onto nodes that have same labels
  - app=myapp
  - operator: In
  - kubernetes.io/hostname




```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: "app"
          operator: In
          values:
          - myapp
      topologyKey: "kubernetes.io/hostname"
```

Pod Affinity

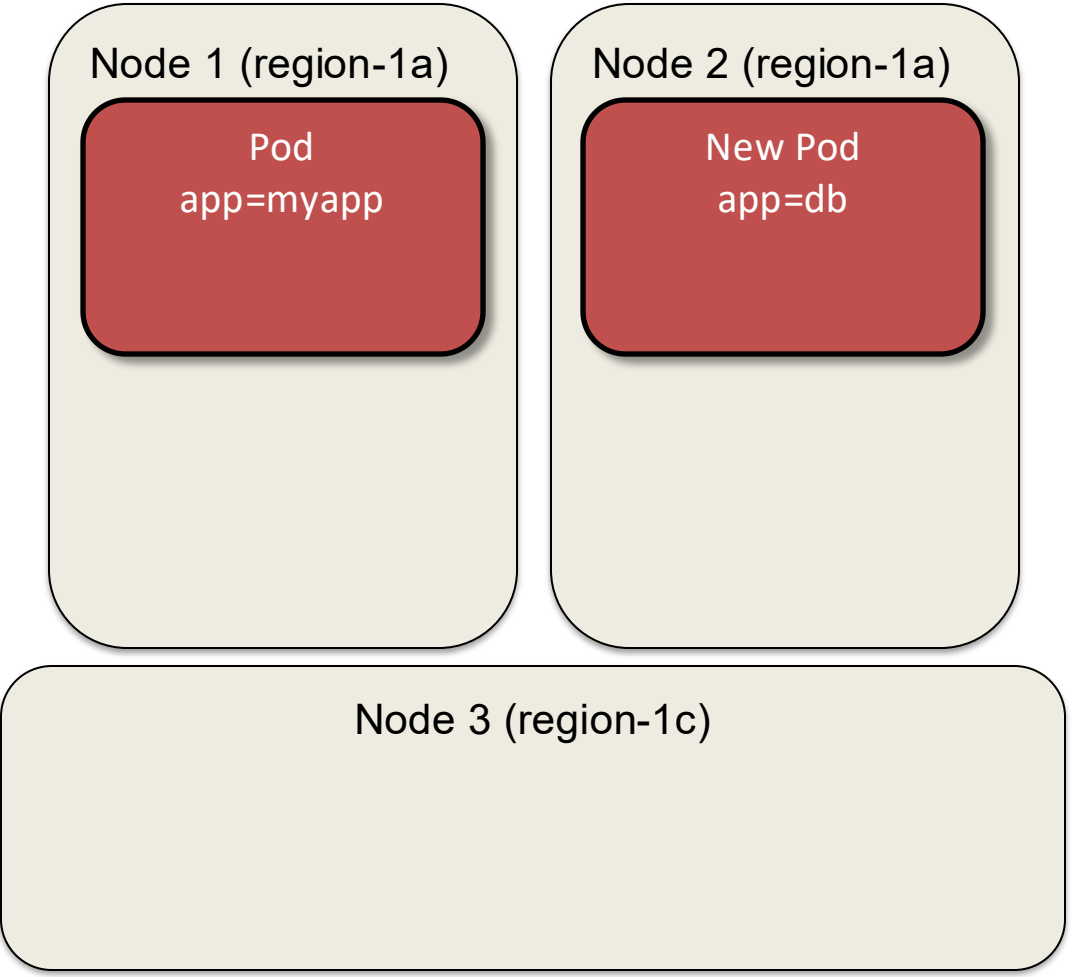
- Schedule Pods onto nodes in same region
  - app=db
  - failure-domain.beta.kubernetes.io/zone




```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: "app"
          operator: In
          values:
           - myapp
      topologyKey: " failure-domain.beta.kubernetes.io/zone"
```

Pod Affinity

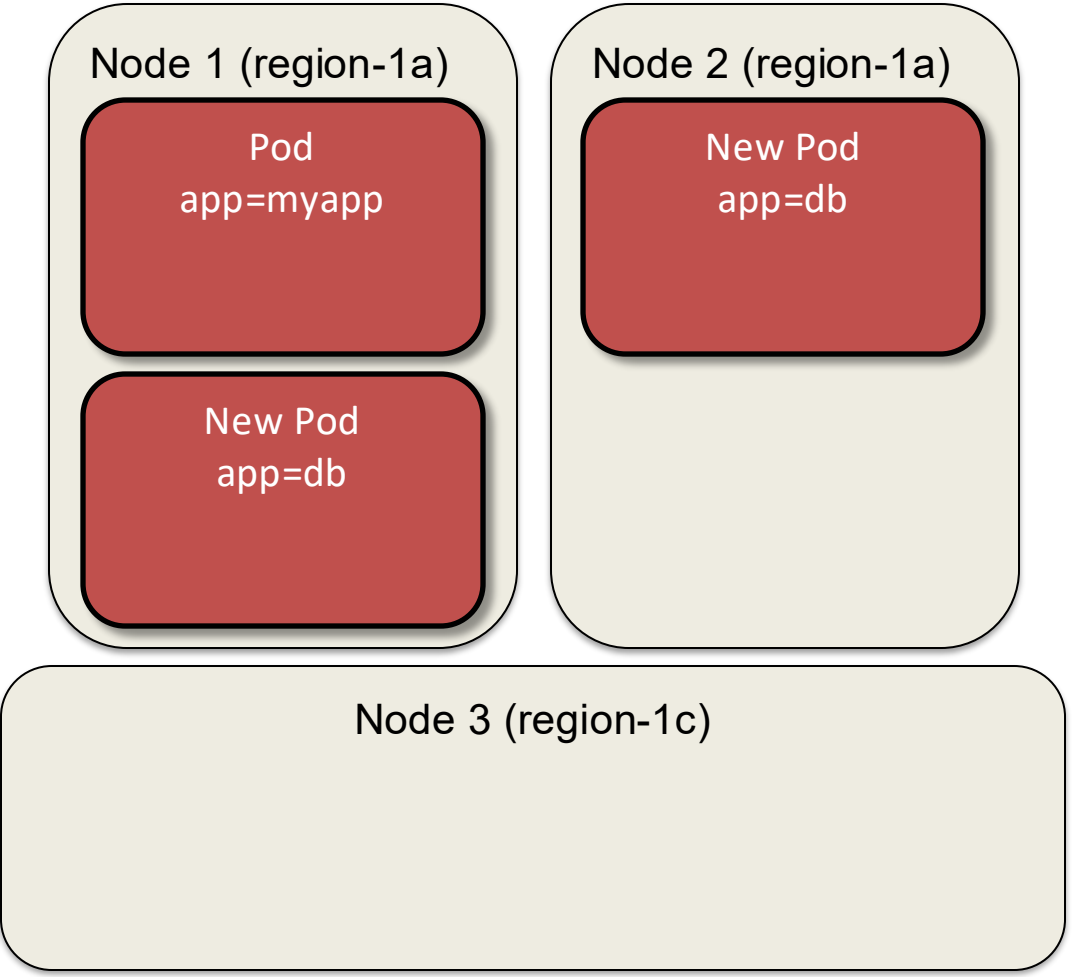
- Schedule Pods onto nodes in same region
  - app=db
  - failure-domain.beta.kubernetes.io/zone




```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: "app"
          operator: In
          values:
           - myapp
      topologyKey: " failure-domain.beta.kubernetes.io/zone"
```

Pod Affinity

- Schedule Pods onto nodes in same region
  - app=db
  - failure-domain.beta.kubernetes.io/zone




```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: "app"
          operator: In
          values:
           - myapp
      topologyKey: " failure-domain.beta.kubernetes.io/zone"
```

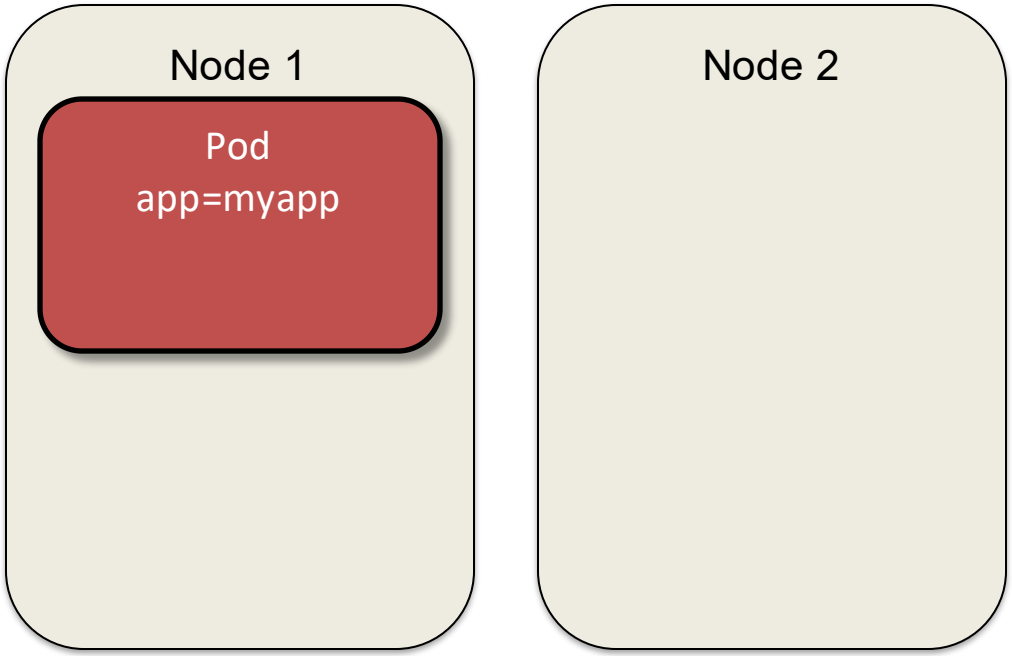
Pod Anti-Affinity

- Define how pods should be placed relative to one another
- Pods of different services run on different nodes.

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: service
          operator: In
          values: [“S1”]
      topologyKey: kubernetes.io/hostname
```

Pod Anti-Affinity

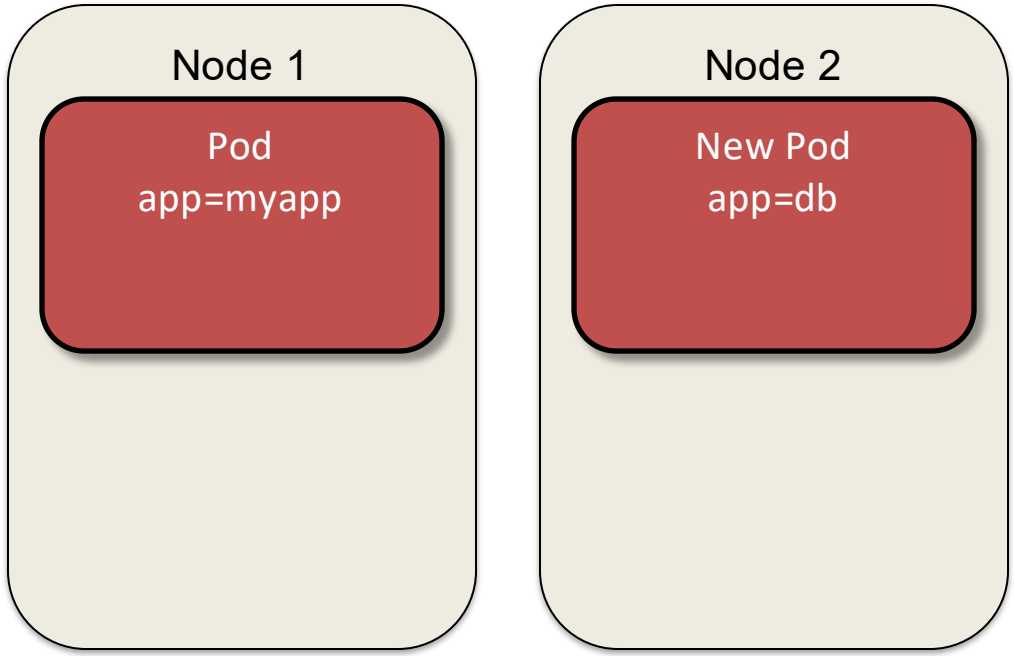
- Do not schedule Pods onto nodes with matching labels.
  - app=db
  - operator: In
  - kubernetes.io/hostname




```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: "app"
          operator: In
          values:
          - myapp
      topologyKey: "kubernetes.io/hostname"
```

Pod Anti-Affinity

- Do not schedule Pods onto nodes with matching labels.
  - app=db
  - operator: In
  - kubernetes.io/hostname




```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: "app"
          operator: In
          values:
          - myapp
      topologyKey: "kubernetes.io/hostname"
```

Lab: Scheduling

ConfigMap

ConfigMap

ConfigMap

- Many applications require configuration via:
  - Config Files
  - Command-Line Arguments
  - Environment Variables
- These need to be decoupled from images to keep portable
- ConfigMap API provides mechanisms to inject containers with configuration data
- Store individual properties or entire config files/JSON blobs
- Key-Value Pairs

ConfigMap

- Not meant for sensitive information
- PODs or controllers can use ConfigMaps

1. Populate the value of environment variables
2. Set command-line arguments in a container
3. Populate config files in a volume

```
kind: ConfigMap
apiVersion: v1
metadata:
    creationTimestamp: 2016-02-18T19:14:38Z
    name: example-config
    namespace: default
data:
    example.property.1: hello
    example.property.2: world
    example.property.file: |-
        property.1=value-1
        property.2=value-2
        property.3=value-3
```

ConfigMap from directory

- 2 files in docs/user-guide/configmap/kubectl
  - game.properties
  - ui.properties

game.properties

```
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
```

ui.properties

```
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

ConfigMap from directory

```
kubectl create configmap game-config --from-file=docs/user-guide/configmap/kubectl
```

```
kubectl describe configmaps game-config
Name: game-config
Namespace: default
Labels: <none>
Annotations: <none>
Data
====
game.properties: 121 bytes
ui.properties: 83 bytes
```

ConfigMap from directory

```
kubectl get configmaps game-config -o yaml
```

```
apiVersion: v1
data:
    game.properties: |-
        enemies=aliens
        lives=3
        …
ui.properties: |
        color.good=purple
        …
kind: ConfigMap
metadata:
    creationTimestamp: 2016-02-18T18:34:05Z
    name: game-config
    namespace: default
    resourceVersion: "407"-
    selfLink: /api/v1/namespaces/default/configmaps/game-config
    uid: 30944725-d66e-11e5-8cd0-68f728db1985
```

ConfigMap from files

```
kubectl create configmap \
game-config-2 \
--from-file=file1 \
--from-file=file2
```

```
kubectl get configmaps \
game-config-2 \
-o yaml
```

```
apiVersion: v1
data:
    game.properties: |-
        enemies=aliens
        lives=3
        …
ui.properties: |
        color.good=purple
        …
kind: ConfigMap
metadata:
    creationTimestamp: 2016-02-18T18:52:05Z
    name: game-config-2
    namespace: default
    resourceVersion: "516"-
    selfLink: /api/v1/namespaces/default/configmaps/game-config-2
    uid: b4952dc3-d670-11e5-8cd0-68f728db1985
```

ConfigMap options

--from-file=/path/to/directory

--from-file=/path/to/file1 (/path/to/file2)

Literal key=value: --from-literal=special.how=very

ConfigMap in PODs

- Populate Environment Variables

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: special-config
    namespace: default
data:
    special.how: very
    special.type: charm
```

OUTPUT

```
SPECIAL_LEVEL_KEY=very
SPECIAL_TYPE_KEY=charm
```

```
apiVersion: v1
kind: Pod
metadata:
    name: dapi-test-pod
spec:
  containers:
      - name: test-container
        image: busybox
        command: ["/bin/sh", "-c, "env" ]
        env:
            - name: SPECIAL_LEVEL_KEY
              valueFrom:
                 configMapKeyRef:
                     name: special-config
                     key: special.how
  restartPolicy: Never
```

ConfigMap Restrictions

- ConfigMaps must be created before they are consumed
- ConfigMaps can only be referenced by objects in the same namespace
- Quota for ConfigMap size not implemented yet
- Can only use ConfigMap for PODs created through API server.

Secrets

What secrets do applications have?

- Database credentials
- API credentials & endpoints (Twitter, Facebook etc.)
- Infrastructure API credentials (Google, Azure, AWS)
- Private keys (TLS, SSH)
- Many more!

Application Secrets

It is a bad idea to include these secrets
in your code.

- Accidentally push up to GitHub with your code
- Push into your file storage and forget about
- Etc.

Application Secrets

There are bots crawling GitHub searching for secrets

Real life example:
Dev put keys out on GitHub, woke up next morning with a ton of emails and missed calls from Amazon

- 140 instances running under Dev's account.
- $2,375 worth of Bitcoin mining

Create Secret

- Designed to hold all kinds of sensitive information
- Can be used by Pods (filesystem & environment variables) and the underlying kubelet when pulling images

```
apiVersion: v1
kind: Secret
metadata:
   name: mysecret
Type: Opaque
data:
    password: mmyWfoidfluL==
    username: NyhdOKwB
```

Pod Secret

```
kind: Pod
metadata:
   name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
    - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
```

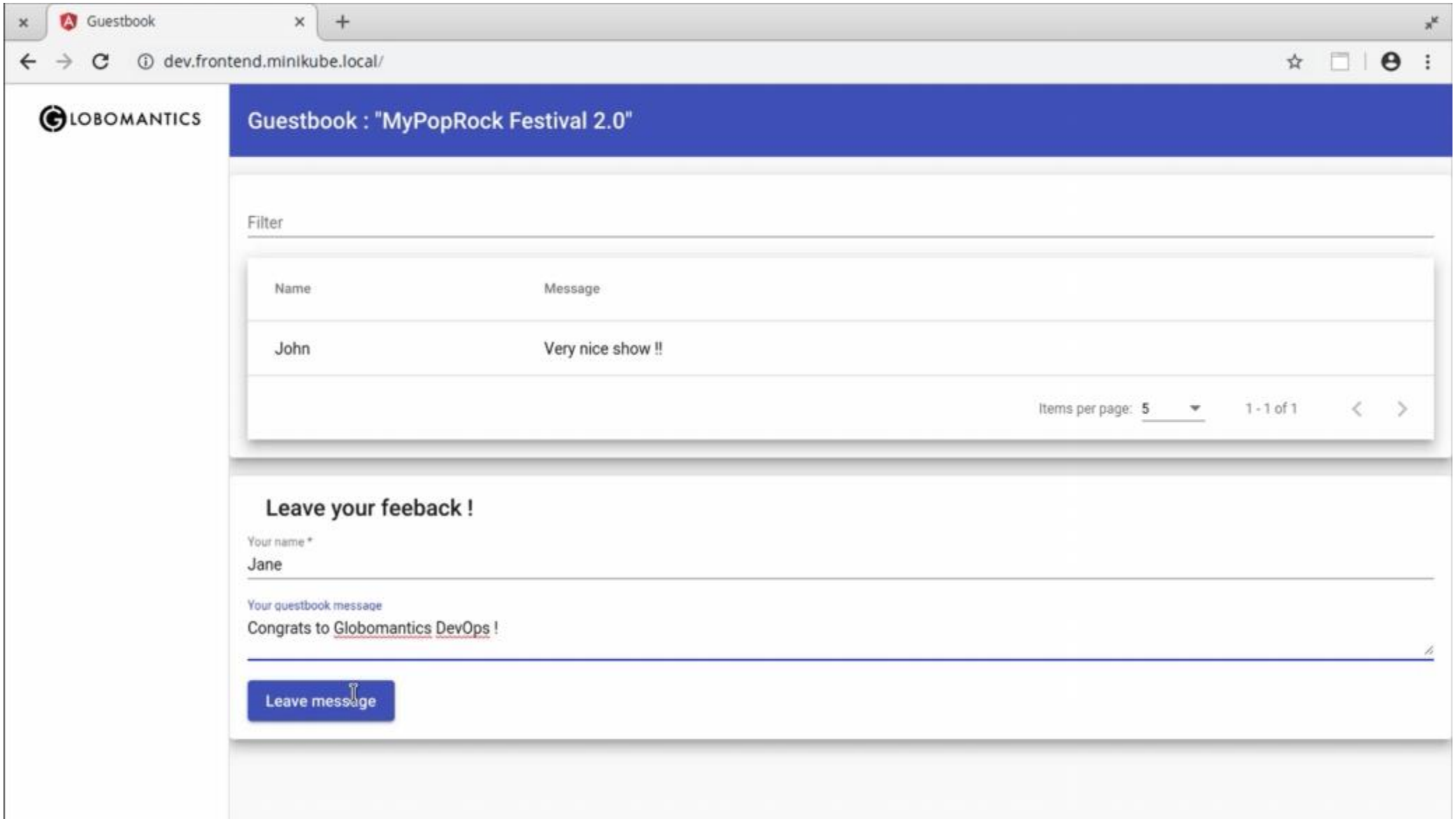
Volume Secret

```
spec:
  containers
  - name: mycontainer
    image: redis
    volumeMounts:
    - name: “secrets”
      mountPath: “/etc/my-secrets”
      readOnly: true
  volumes:
   - name: “secrets”
      secret:
        secretName: “mysecret”
```

Labs: ConfigMap & Secrets

Helm: Package management

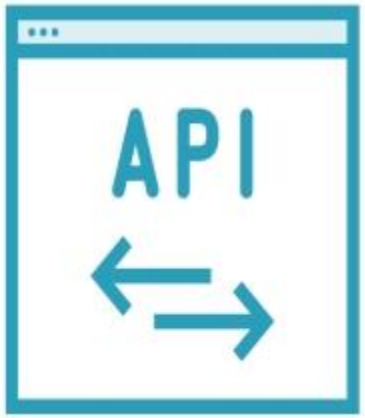
Helm sample application



Sample application components

ConfigMap
Pod
Service
Ingress

Frontend



Secret
Pod
Service

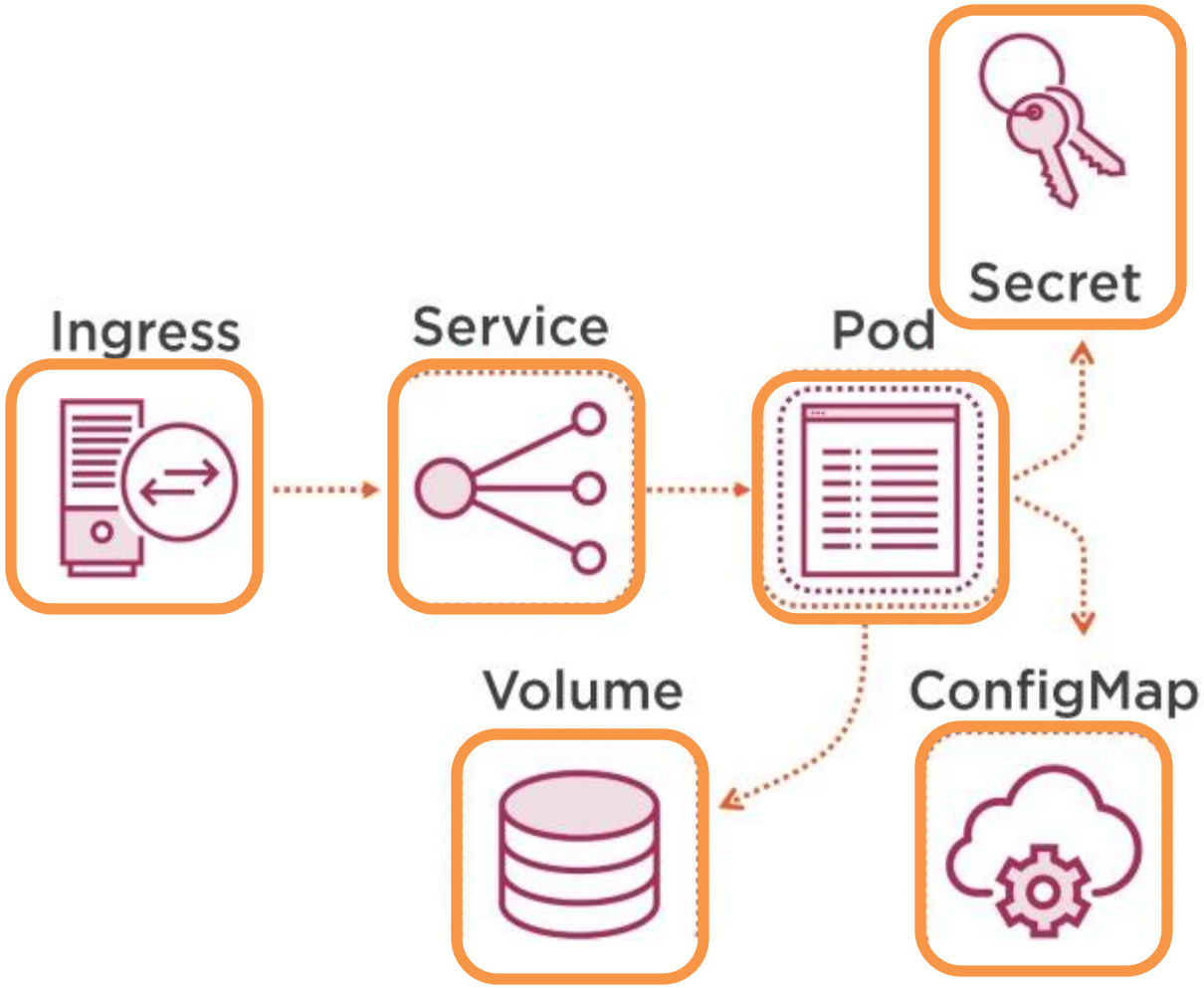
Backend API

Secret
PV
PVC
Pod
Service

Database

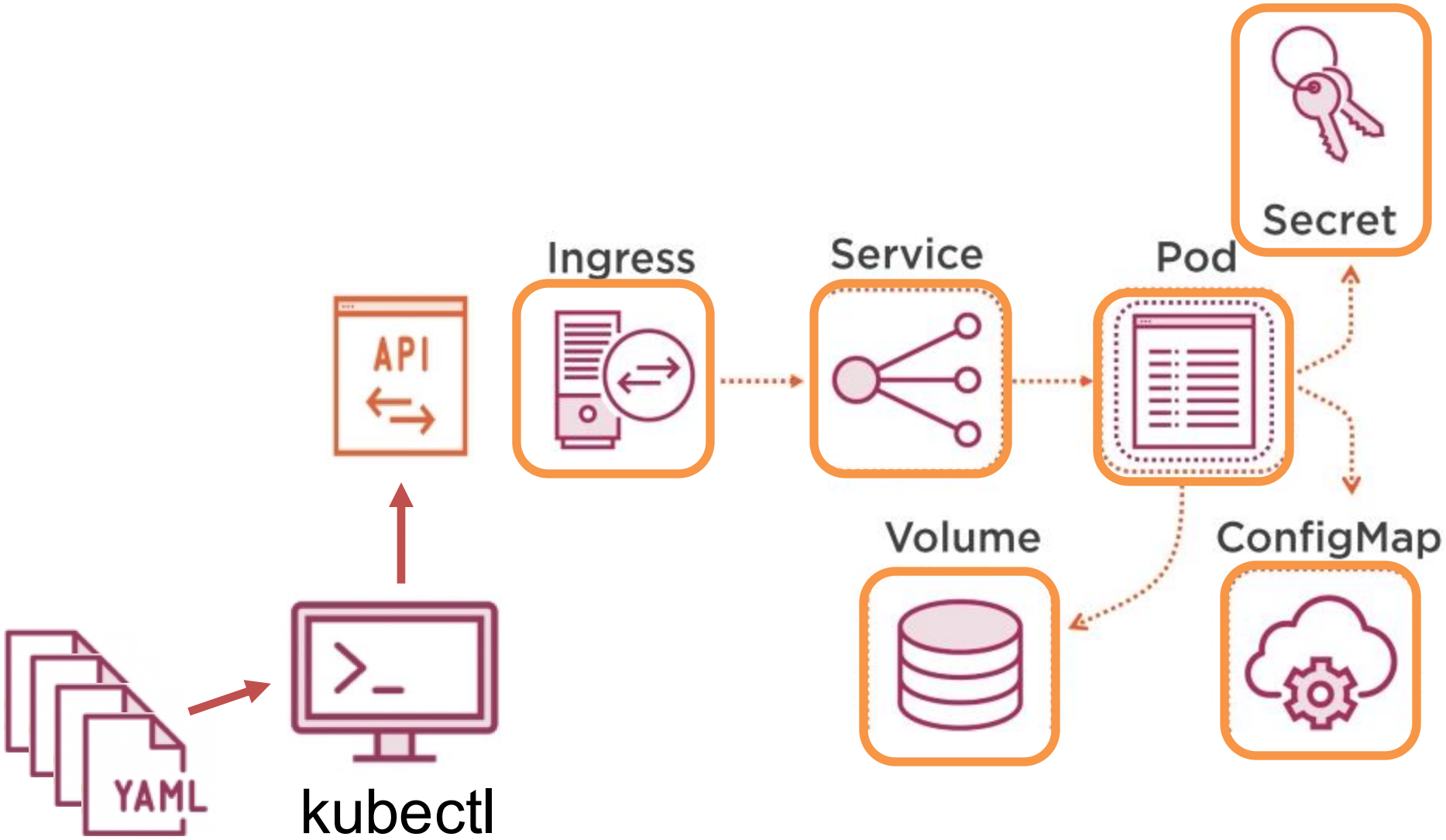
Native Kubernetes way

- Application
- Container
- Pod
- Service
- Ingress
- ConfigMap
- Secrets
- Volumes: PV, PVC, Storage



Native Kubernetes way

- Limitations
  - Packaging
  - Versioning



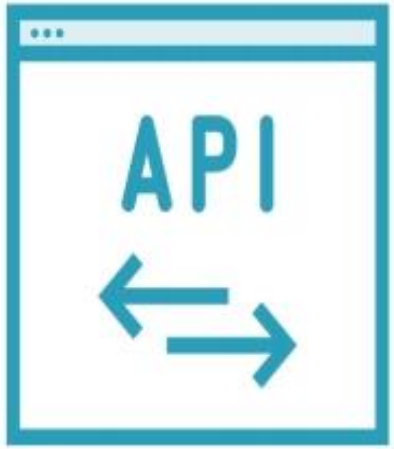
Guestbook: version 1

- ConfigMap
- Pod
- Service
- Ingress

Frontend

Guestbook: version 2

- ConfigMap
- Pod
- Service
- Ingress



- Secret
- Pod
- Service

- Secret
- PV
- PVC
- Pod
- Service

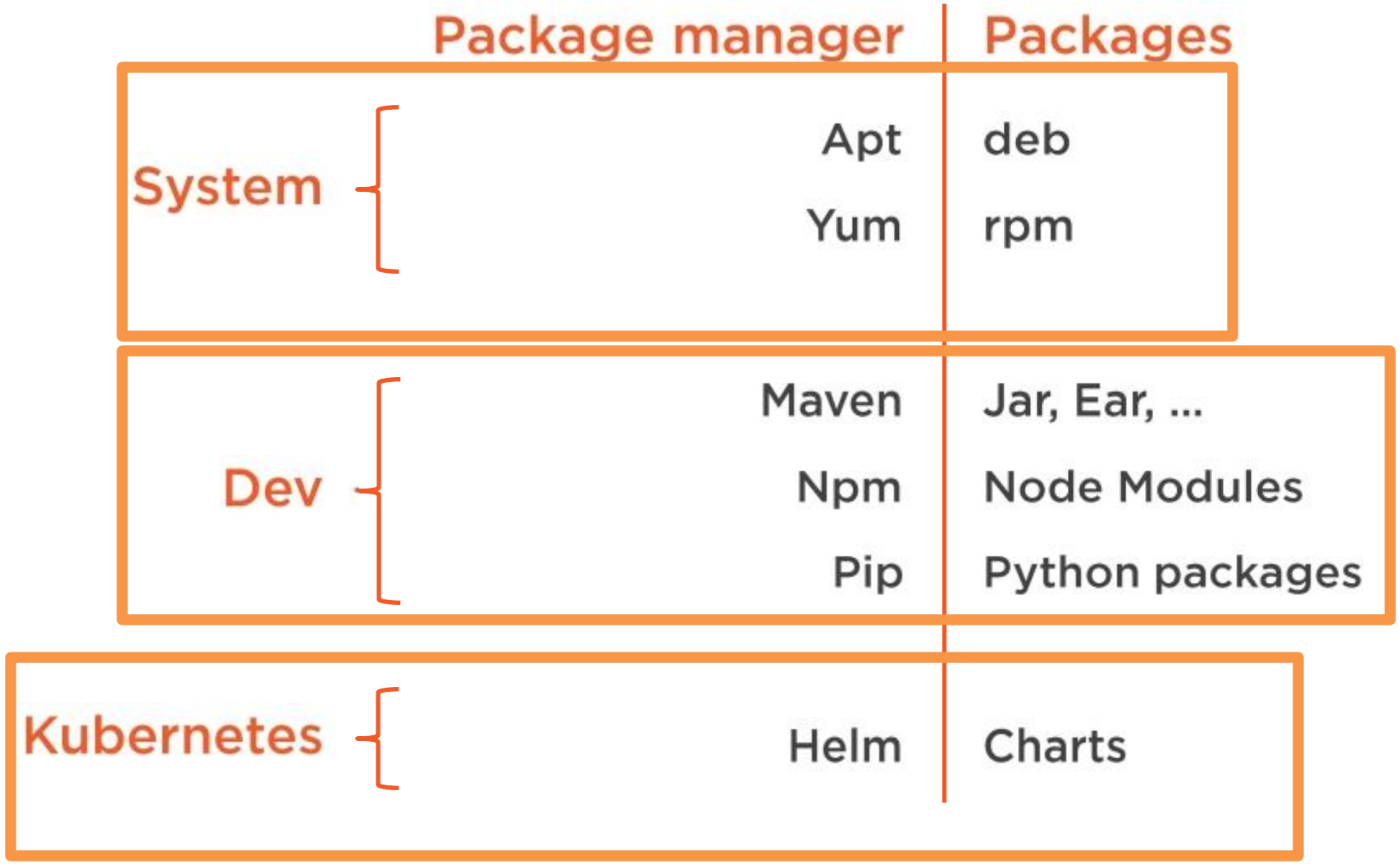
Frontend

Backend

Databas
e

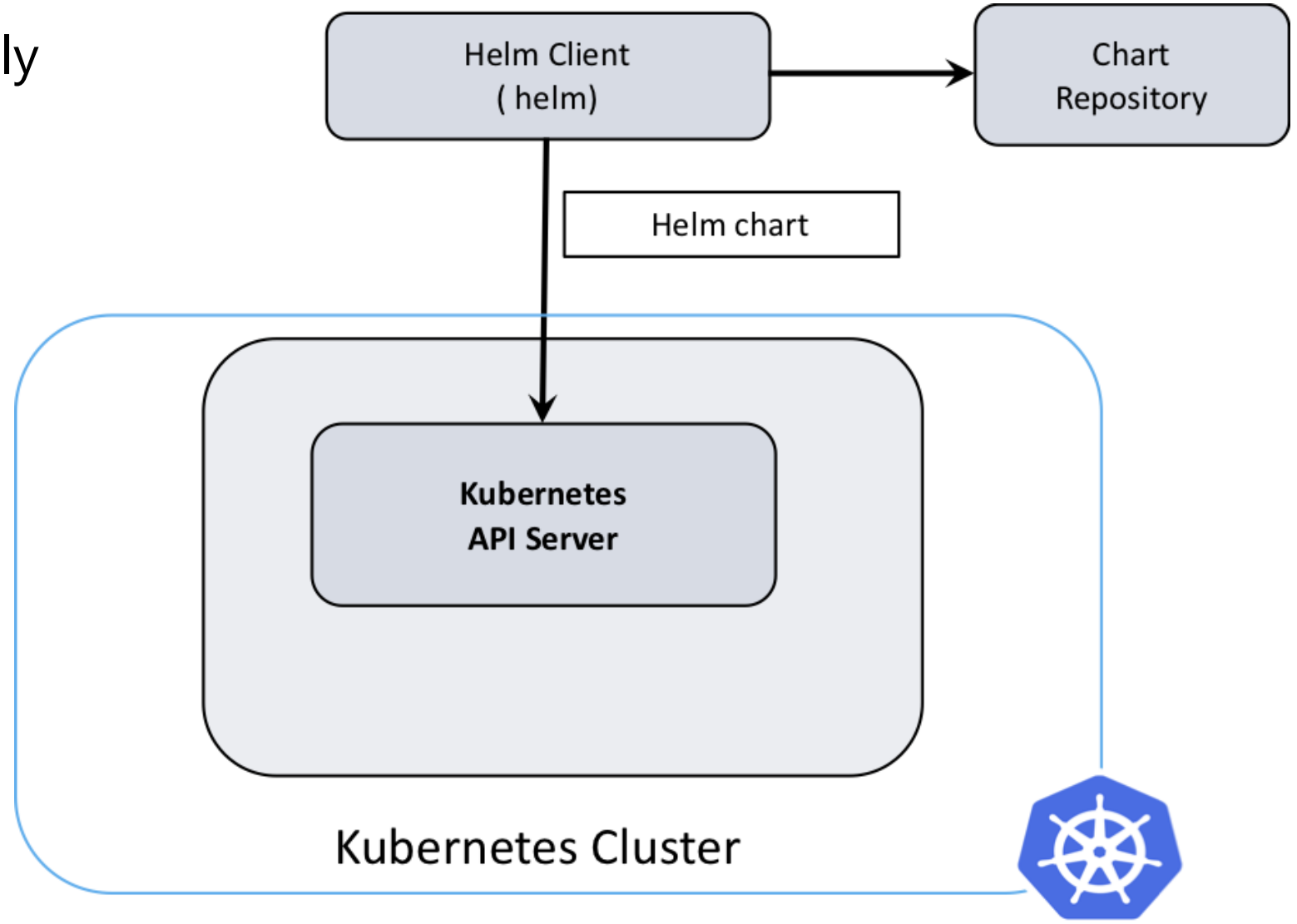
Painful to manage

Helm features



Helm architecture

- Helm:
  - develop charts locally
  - command-line



Helm features

Charts

Templates

Dependencies

Repositories

Helm Chart Structure

Helm Chart structure

- nginx-demo
  - Chart.yaml

- Chart properties
  - name
  - version
  - more..

Helm Chart structure

- nginx-demo
  - Chart.yaml
  - README.md

- Document chart
  - Overrides
  - Maintainer
  - Instructions

Helm Chart structure

- nginx-demo
  - Chart.yaml
  - README.md
- templates
  - deployment.yaml
  - ingress.yaml
  - service.yaml

- Kubernetes object definitions
  - customizable YAML templates

Helm Chart structure

- nginx-demo
  - Chart.yaml
  - README.md
- templates
  - deployment.yaml
  - ingress.yaml
  - service.yaml
- values.yaml

- Kubernetes object definitions
  - customizable YAML templates
    - Provide default values.

Helm Chart structure

- nginx-demo
  - Chart.yaml
  - README.md
- templates
  - deployment.yaml
  - ingress.yaml
  - service.yaml
  - NOTES.txt
- values.yaml

- Provide helpful output after installation
  - How to access application
  - Create user/pass

Helm Chart structure

- nginx-demo
  - Chart.yaml
  - README.md
- templates
  - deployment.yaml
  - ingress.yaml
  - service.yaml
  - NOTES.txt
  - tests
    - test-connection.yaml
- values.yaml

- You can create tests to confirm Chart works as expected.

Helm Chart structure

- nginx-demo
  - Chart.yaml
  - README.md
  - requirements.yaml
- templates
  - deployment.yaml
  - ingress.yaml
  - service.yaml
  - NOTES.txt
  - tests
    - test-connection.yaml
- values.yaml

- Define sub-charts and dependencies

Helm Chart.yaml

Chart.yaml

```
apiVersion: v1
appVersion: "1.0"
description: A Helm chart for Kubernetes
name: nginx-demo
version: 0.1.0
```

Major,Minor,Patch (SemVer 2.0)

- App you are installing

Helm Chart.yaml

Chart.yaml

```yaml
apiVersion: v1
appVersion: "1.0"
description: A Helm chart for Kubernetes
name: nginx-demo
version: 0.1.0
```

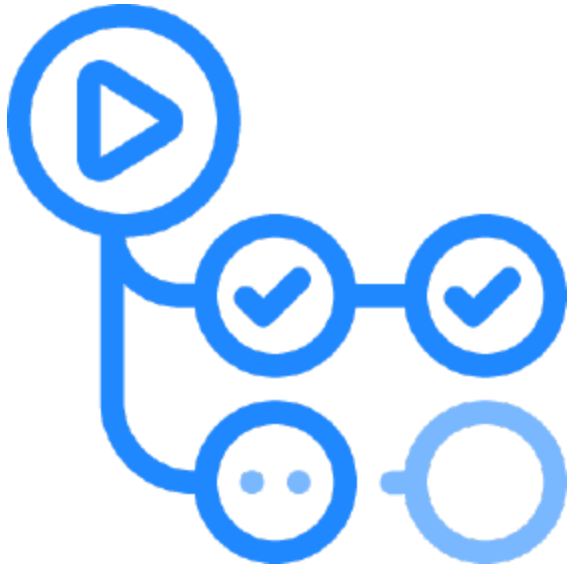
Major,Minor,Patch (SemVer 2.0)

- Version of Helm chart

Lab: Helm

GITHUB ACTIONS

WHAT IS GITHUB ACTIONS?



GitHub Actions is a powerful automation platform integrated directly into GitHub. It enables developers to automate workflows for building, testing, and deploying applications directly from their GitHub repositories. By using GitHub Actions, teams can streamline their CI/CD pipelines, ensuring faster and more reliable software delivery.

Core Concept: Automate tasks in response to events, such as code commits, pull requests, or manual triggers.

Key Use Cases:

- Automatically run tests and linting when code is pushed.
- Build and deploy applications to staging or production environments.
- Perform regular maintenance tasks, such as dependency updates or security scans.

Example: A developer commits code to a repository, triggering a workflow that tests the code, builds the application, and deploys it to a staging server.

KEY CONCEPTS OF GITHUB ACTIONS

Workflows: Defined automation processes written in YAML files. These describe the steps GitHub Actions should perform.

Triggers: Events in the repository (e.g., push, pull_request, schedule) that start a workflow.

Jobs: A collection of steps executed in sequence or parallel within a workflow.

Runners: Virtual machines or containers where the jobs are executed.

Example: A workflow triggered on a pull_request event runs linting and testing jobs.

BENEFITS OF GITHUB ACTIONS

Integration: Fully integrated with GitHub repositories, reducing the need for external CI/CD tools.

Scalability: Supports multiple runners, allowing workflows to execute across various operating systems and environments.

Customization: Offers pre-built actions from the GitHub Marketplace and supports creating custom actions for unique use cases.

Community Support: A large ecosystem of contributors provides reusable actions and templates.

For instance, developers can use actions from the marketplace to deploy code to AWS, Azure, or Google Cloud.

WORKFLOW CONFIGURATION WITH YAML

yaml

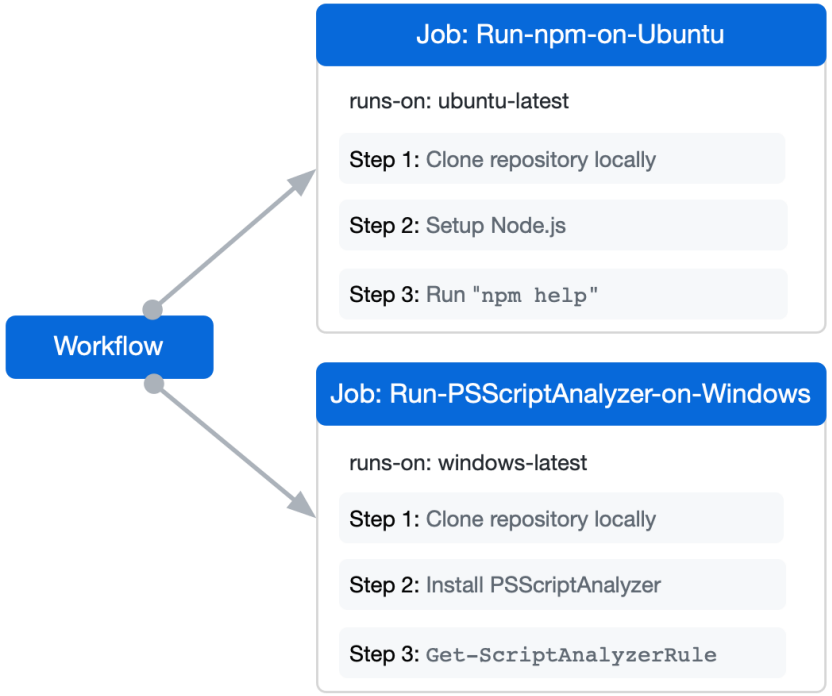
```yaml
name: Build and Test
on: push
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run Tests
        run: npm test
```

GitHub Actions **workflows** are configured using simple YAML files stored in the repository under .github/workflows. These files define the sequence of tasks to automate.

Key Elements:

- **Name:** A descriptive name for the workflow (e.g., “Build and Test”).
- **Triggers:** Define the events that start the workflow (e.g., push, pull_request).
- **Jobs:** Specify tasks to perform, such as testing, building, or deploying.
- **Steps:** Individual actions within a job, such as running a script or installing dependencies.

EXTENSIVE SUPPORT FOR RUNNERS



GitHub Actions provides a variety of environments for running workflows:

Hosted Runners:

- Pre-configured environments maintained by GitHub.
- Support for Windows, Linux, and macOS.
- Scalable and easy to use for most CI/CD needs.

Self-Hosted Runners:

- Custom environments managed by users.
- Ideal for workflows requiring specific hardware, software, or network configurations.

Example: A team uses a self-hosted runner to execute workflows on an internal Kubernetes cluster.

GITHUB MARKETPLACE

```yaml
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Install dependencies
        uses: actions/setup-node@v3
        with:
          node-version: 16

      - name: Deploy to AWS
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        run: aws s3 sync ./build s3://my-bucket
```

The **GitHub Marketplace** provides a centralized platform to explore and integrate pre-built actions into your workflows. These actions, created by GitHub or the community, automate complex tasks, allowing developers to focus on their core application logic.

Access Pre-Built Actions: Browse thousands of ready-to-use actions designed to handle tasks like testing, deployments, notifications, and more. The Marketplace offers a wide variety of actions suitable for various languages, platforms, and services.

Examples of Popular Actions:

- **Deployment Actions:** Actions for deploying to AWS, Azure, Google Cloud, or Kubernetes.
- **Testing Actions:** Run Selenium tests, execute unit tests, or validate infrastructure with Terratest.
- **Notification Actions:** Automate communication by sending messages to Slack, Teams, or email upon workflow completion.

GITHUB MARKETPLACE

yaml

```yaml
name: Build and Deploy Workflow

# Trigger the workflow on pushes to the main branch
on:
  push:
    branches:
      - main

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest  # Use the latest Ubuntu runner

    steps:
      # Step 1: Plugin to check out the repository code
      # This action pulls the code from the repository into the workflow runner.
      - name: Checkout repository
        uses: actions/checkout@v3  # Plugin from GitHub Marketplace
```

The **GitHub Marketplace** provides a centralized platform to explore and integrate pre-built actions into your workflows. These actions, created by GitHub or the community, automate complex tasks, allowing developers to focus on their core application logic.

Access Pre-Built Actions: Browse thousands of ready-to-use actions designed to handle tasks like testing, deployments, notifications, and more. The Marketplace offers a wide variety of actions suitable for various languages, platforms, and services.

Examples of Popular Actions:

- **Deployment Actions:** Actions for deploying to AWS, Azure, Google Cloud, or Kubernetes.
- **Testing Actions:** Run Selenium tests, execute unit tests, or validate infrastructure with Terratest.
- **Notification Actions:** Automate communication by sending messages to Slack, Teams, or email upon workflow completion.

GITHUB ACTIONS PLUGINS OVERVIEW

GitHub Actions plugins are reusable components that extend the functionality of workflows by integrating external tools and services. These plugins simplify the setup of tasks and make workflows modular and efficient.

A plugin is any action defined in a GitHub repository or available in the GitHub Marketplace that provides specific functionality, such as setting up environments or interacting with cloud services.

Examples of Plugins:

- **Setup plugins:** Actions that configure environments (e.g., actions/setup-node to configure Node.js).
- **Deployment plugins:** Tools to manage deployments (e.g., actions/deploy-pages for GitHub Pages).
- **Utility plugins:** General tools like actions/checkout for cloning repositories into workflows.

HOW PLUGINS WORK

yaml

```yaml
name: CI/CD Workflow

on:
  push:
    branches:
      - main

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest

    steps:
      # Step 1: Plugin to check out the repository code
      - name: Checkout code
        uses: actions/checkout@v3  # Plugin from GitHub Marketplace

      # Step 2: Plugin to set up Node.js environment
      - name: Setup Node.js
        uses: actions/setup-node@v3  # Plugin from GitHub Marketplace
        with:
          node-version: 16
```

GitHub Actions plugins function as essential building blocks in workflows, automating tasks and ensuring consistency. They allow developers to focus on core logic by handling repetitive processes like setup, deployment, and testing. The key components of how plugins work include:

- **Integration:** Plugins are added in YAML workflows, performing tasks like environment setup or deploying applications seamlessly.
- **Input Parameters:** Plugins allow configurable inputs, such as specifying versions or paths, to tailor them to workflow requirements.
- **Secrets Management:** Sensitive data like API keys or tokens is securely stored and accessed through GitHub's built-in secrets.
- **Output Data:** Plugins generate outputs that subsequent steps in the workflow can use, enabling flexible and dynamic processes.

BEST PRACTICE FOR USING PLUGINS

To make the most of GitHub Actions plugins, it is important to follow best practices that ensure workflows remain secure, efficient, and maintainable. These practices include:

- **Verified Sources:** Choose plugins from GitHub Marketplace or reputable repositories to avoid security vulnerabilities.
- **Specify Plugin Versions:** Pin plugins to fixed versions (e.g., @v3) to ensure workflow stability and prevent unexpected updates.
- **Optimize Plugin Usage:** Limit the number of plugins to essential tasks, selecting lightweight options to improve performance.
- **Regular Updates:** Update plugins periodically to benefit from new features, bug fixes, and security patches.

WHAT ARE BUILD PIPELINES

Build pipelines are automated workflows that manage the steps required to build, test, and prepare software for deployment. They provide a structured way to ensure that code changes are validated and production-ready at every stage of the development lifecycle.

A build pipeline automates activities like compiling code, resolving dependencies, and generating deployable artifacts, ensuring repeatability and consistency.

- **Stages of a Pipeline:** Typical stages include build (source code compilation), test (validation of code quality), and deploy (packaging and preparing for release).
- **Benefits:** Build pipelines reduce manual errors, accelerate feedback loops for developers, and ensure the software adheres to quality standards before deployment.

KEY COMPONENTS OF A BUILD PIPELINE

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Build application
        run: make build

      - name: Run tests
        run: make test
```

Build pipelines rely on several critical components that work together to automate software delivery effectively. These components are designed to handle various stages of development and ensure the pipeline is efficient and reliable.

- **Source Control Integration:** Pipelines monitor repositories for code changes and trigger workflows automatically upon commits or pull requests.
- **Build Stage:** This stage compiles the source code, resolves dependencies, and generates binaries or other deployable artifacts.
- **Testing Stage:** Automated tests—such as unit, integration, or end-to-end tests—are run to ensure the code behaves as expected and remains stable.
- **Artifact Generation:** Pipelines create deployable outputs, such as container images or packaged files, which are ready for deployment.

BEST PRACTICES FOR BUILD PIPELINES

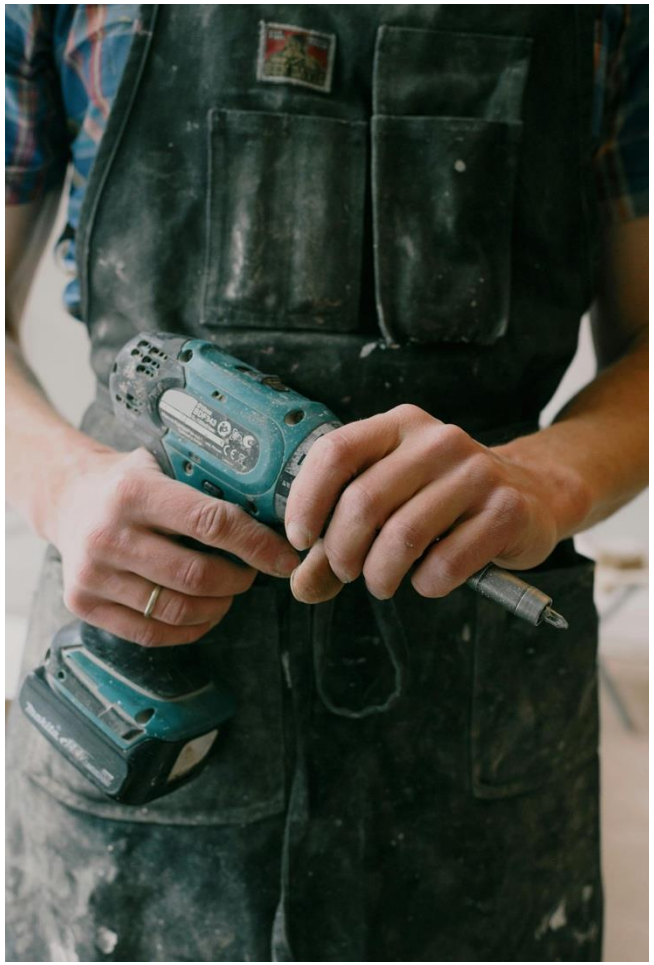
yaml

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Build application
        run: make build

  # Parallel test jobs to run tests
  # on multiple Node.js versions
  test:
    runs-on: ubuntu-latest
    needs: build  # Ensures the 'test' job runs
                  # only after the 'build' job
    strategy:
      matrix:
        node: [14, 16, 18]  # Creates a matrix to run
                            # tests on Node.js versions
                            # 14, 16, and 18 concurrently
    steps:
      - name: Run tests on Node.js ${{ matrix.node }}
        run: npm test
```

To make build pipelines efficient, maintainable, and reliable, it's essential to follow industry best practices. These practices optimize workflows, reduce errors, and ensure smooth operation across stages:

- **Use Modular Stages:** Divide pipelines into separate stages (build, test, deploy) to isolate issues and make debugging easier.
- **Fail Fast:** Configure pipelines to stop execution immediately upon encountering an error to save time and resources.
- **Parallel Execution:** Optimize performance by running independent tasks, such as testing across environments, in parallel.
- **Monitor and Optimize:** Use logs and analytics to identify bottlenecks, optimize performance, and ensure pipelines remain efficient.
- **Pipeline as Code:** Store pipeline configurations as code in the repository, enabling version control and collaboration.

WHAT ARE BUILD TRIGGERS?



Build triggers are events or conditions that automatically initiate the execution of a pipeline. They eliminate the need for manual intervention and ensure that workflows are executed in response to specific actions or schedules, enabling seamless automation.

Definition: Build triggers define when and how a pipeline starts, based on events like code changes, pull requests, or timed schedules.

Types of Triggers: Triggers can be event-driven (e.g., on a commit or pull request) or time-based (e.g., scheduled builds).

Benefits: Build triggers ensure builds are consistent, reduce manual steps, and allow teams to catch issues early by automating workflow execution.

COMMON BUILD TRIGGER TYPES

yaml

```yaml
name: Build and Test Pipeline

# Define triggers for the workflow
on:
  # Push events: Trigger when changes are pushed to the main branch or tags
  push:
    branches:
      - main
    tags:
      - v*

  # Pull request events: Trigger when a PR is opened or updated, targeting the main branch
  pull_request:
    branches:
      - main

  # Scheduled triggers: Run every day at 2 AM using cron syntax
  schedule:
    - cron: '0 2 * * *'

  # Manual trigger: Allow users to manually start the workflow through the Actions UI
  workflow_dispatch:
    inputs:
      environment:
        description: 'Target environment (e.g., staging, production)'
        required: true
        default: staging
```

Triggers in GitHub Actions are highly customizable and can adapt to various workflow needs. Key types include:

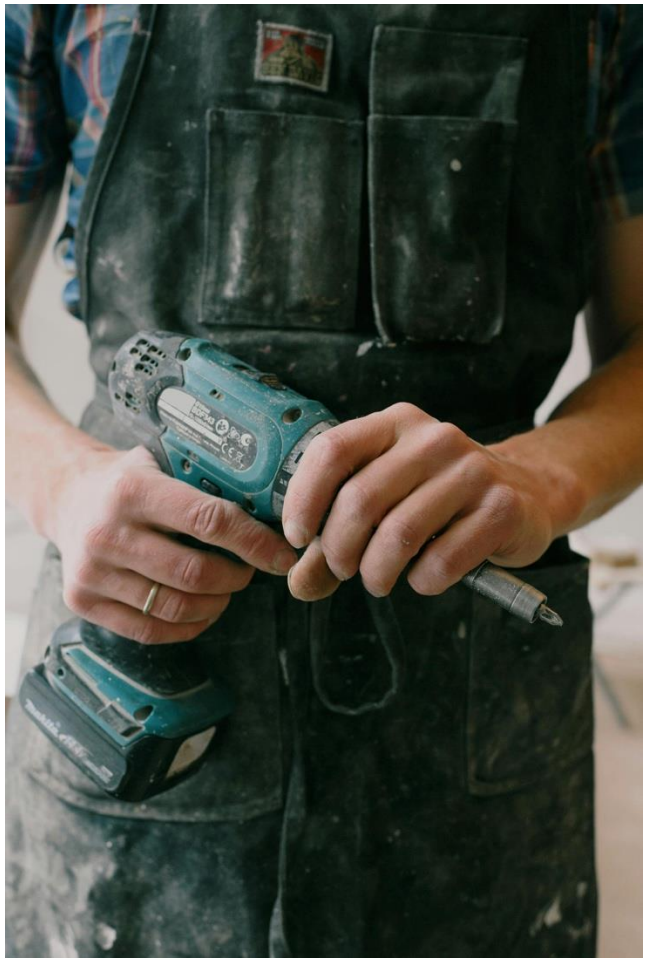
Push Events: Automatically trigger workflows when changes are pushed to a branch or a tag. Useful for continuous integration on the main branch.

Pull Requests: Start workflows when a pull request is opened or updated. Helps validate code changes before merging.

Scheduled Triggers: Use cron syntax to schedule builds at specific intervals, such as nightly tests or weekly deployments.

Manual Triggers: Enable workflows to be triggered manually through the GitHub Actions UI or via API calls for ad-hoc needs.

BEST PRACTICES FOR BUILD TRIGGERS



Configuring build triggers thoughtfully ensures workflows run efficiently and only when necessary. Best practices include:

Scope Triggers to Relevant Changes: Limit triggers to specific branches (e.g., main) or tags to avoid running workflows on irrelevant updates.

Use Filters: Apply paths or paths-ignore to trigger workflows only when specific files or directories are modified.

Combine Triggers: Use a combination of triggers, such as push and pull_request, to address different scenarios like direct commits and code reviews.

Avoid Over-Triggering: Ensure workflows aren't triggered excessively, which could waste resources or cause unnecessary delays.

WHAT ARE ENVIRONMENT VARIABLES?

```yaml
env:
  NODE_ENV: production

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Print variable
        run: echo $NODE_ENV
```

Environment variables are dynamic values that store configuration data, secrets, or other information required by workflows during execution. They allow developers to customize workflows without hardcoding values directly into the pipeline.
Environment variables are key-value pairs accessible to workflows and steps, used for storing reusable configuration like API keys, URLs, or environment-specific settings.

Scope of Variables: Variables can be scoped globally (workflow-wide), per job, or per step. This flexibility ensures that sensitive or environment-specific data is kept secure and accessible where needed.

Benefits: Environment variables simplify workflows, improve reusability, and make pipelines easier to maintain by centralizing configuration data.

USING ENVIRONMENT VARIABLES IN WORKFLOWS

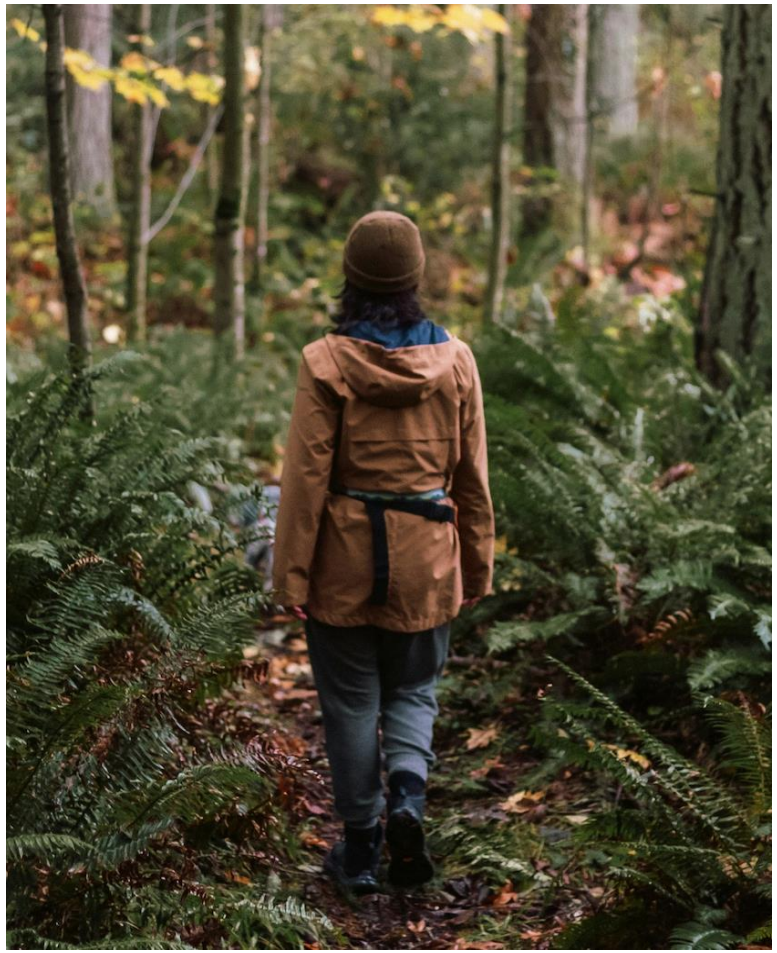
yaml

```yaml
name: Variable Scope Example

on:
  push:
    branches:
      - main

env:  # Workflow-level variables
  WORKFLOW_VAR: "This is accessible to all jobs and steps"

jobs:
  build:
    runs-on: ubuntu-latest

    env:  # Job-level variables
      JOB_VAR: "This is accessible only within the build job"

    steps:
      - name: Print workflow variable
        run: echo "Workflow variable: $WORKFLOW_VAR"

      - name: Print job variable
        run: echo "Job variable: $JOB_VAR"

      - name: Define and print step variable
        run: |
          STEP_VAR="This is only accessible within this step"
          echo "Step variable: $STEP_VAR"
```

Environment variables in GitHub Actions can be set and accessed at various levels, providing flexibility in managing configuration data. Env vars can apply at many different levels:

- **Workflow Level:** Variables defined at this level are accessible throughout the entire workflow.
- **Job Level:** Variables can be scoped to a specific job, ensuring other jobs cannot access them.
- **Step Level:** For maximum isolation, variables can be defined for individual steps.

BEST PRACTICES FOR ENVIRONMENT VARIABLES



Managing environment variables effectively ensures security, maintainability, and clarity in workflows. Key practices include:

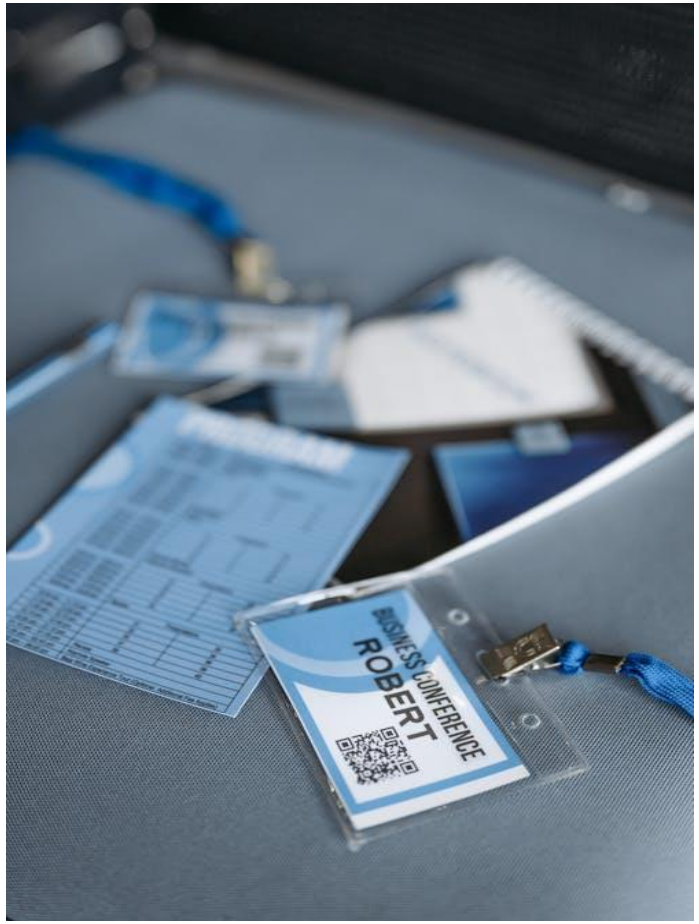
Limit Scope: Define variables only at the level where they are needed (workflow, job, or step) to minimize unintended access.

Avoid Hardcoding: Use variables for values that may change across environments (e.g., development, staging, production) to simplify updates.

Secure Secrets: Store sensitive information like API keys and credentials in GitHub Secrets, which encrypts the data.

Document Variables: Clearly document the purpose and scope of each variable in the repository's README or pipeline documentation.

GITHUB ACTIONS CREDENTIALS



Credentials are sensitive data such as authentication tokens, API keys, or certificates that workflows use to access external services securely. Proper management of credentials is critical for maintaining the security of your workflows and systems.

Credentials refer to any sensitive information required by workflows to interact with external services or perform secure operations (e.g., accessing cloud platforms or private repositories).

- **Storage:** GitHub Secrets is the primary method for securely storing credentials, encrypting the data and restricting access.
- **Usage:** Credentials enable workflows to authenticate securely without exposing sensitive data directly in the pipeline.

MANAGING CREDENTIALS

yaml

```yaml
name: Secure Credentials Example

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      # Step 1: Checkout the repository code
      - name: Checkout code
        uses: actions/checkout@v3

      # Step 2: Configure AWS Credentials securely using GitHub Secrets
      - name: Configure AWS Credentials
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}  # Injected secret
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}  # Injected secret
        run: echo "AWS credentials configured"

      # Step 3: Deploy application to AWS using injected credentials
      - name: Deploy to AWS
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        run: |
          aws s3 sync ./build s3://my-bucket
```

Credentials are securely managed through GitHub Secrets and are injected into workflows when needed. Key considerations for managing credentials include:

GitHub Secrets: Use the repository's Secrets feature to store credentials securely. Secrets are accessible only to authorized workflows.

Accessing Secrets: Secrets are injected as environment variables and accessed using ${{ secrets.SECRET_NAME }} in workflows.

Scoped Access: Secrets can be scoped at the repository, organization, or environment level to limit access to only the workflows that need them.

BEST PRACTICES FOR CREDENTIALS

Securely managing credentials ensures the safety of your workflows and external services. Follow these practices to maintain security:

Use GitHub Secrets: Always store sensitive information like API keys or tokens in GitHub Secrets instead of hardcoding them in workflows.

Scope Secrets Carefully: Restrict access to secrets based on the principle of least privilege. For example, use environment-level secrets for production credentials.

Rotate Secrets Regularly: Update and rotate secrets periodically to reduce the risk of compromise.

Audit Secret Usage: Regularly review workflows to ensure secrets are being used securely and that no unnecessary secrets are defined.

PARAMETERIZATION OVERVIEW

Parameterization allows workflows and pipelines to use dynamic inputs instead of hardcoding values. By leveraging parameters, pipelines can adapt to different environments, configurations, or inputs, improving flexibility and maintainability.

Parameterization introduces variables that are set dynamically, enabling workflows to behave differently based on the inputs provided.

- **Use Cases:** Deploying to multiple environments (e.g., dev, staging, prod), running tests for different configurations, or customizing resource provisioning.
- **Benefits:** Improves reusability, reduces redundancy, and simplifies updates by centralizing configurable values.

DEFINING AND USING PARAMETERS

```yaml
name: CI/CD Workflow with Parameters

on:
  # Trigger the workflow manually with inputs
  workflow_dispatch:
    inputs:
      environment:
        description: 'Target environment (staging, production)'  # Workflow Input
        required: true
        default: staging

jobs:
  build:
    runs-on: ubuntu-latest

    # Define environment variables for the entire job
    env:
      API_URL: https://api.example.com  # Environment Variable
      DEPLOY_REGION: us-east-1

    steps:
      # Step 1: Checkout repository
      - name: Checkout code
        uses: actions/checkout@v3

      # Step 2: Print workflow input and environment variables
      - name: Print parameters
        run: |
          echo "Target environment: ${{ inputs.environment }}"
          echo "API URL: $API_URL"
          echo "Deploy region: $DEPLOY_REGION"

  test:
    runs-on: ubuntu-latest

    # Use a matrix to run tests across multiple configurations
    strategy:
      matrix:
        node-version: [14, 16, 18]  # Matrix Parameter for Node.js versions
        os: [ubuntu-latest, windows-latest]  # Matrix Parameter for operating systems

    steps:
      # Step 1: Setup Node.js with the current matrix version
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node-version }}
```

GitHub Actions workflows allow for dynamic behavior by defining and consuming parameters. These parameters control workflow execution and can be defined at different levels or passed from external sources.

Workflow Inputs: Inputs are set at the workflow level, ideal for manual triggers or reusable workflows. For example, users can specify target environments like staging or production when triggering a workflow.

Environment Variables: These store configuration values shared across multiple steps or jobs, such as API URLs or deployment regions, enabling centralized and reusable settings.

Matrix Parameters: Matrices enable running jobs in parallel with varying configurations, such as testing across Node.js versions or operating systems, ensuring broader coverage and efficiency.

BEST PRACTICES FOR PARAMETERIZATION

Effectively parameterizing workflows ensures better reusability, security, and maintainability. Key practices include:

Validate Inputs: Use conditional logic to validate parameter values and prevent invalid configurations.

Keep Parameters Relevant: Only define parameters necessary for the workflow's flexibility and avoid overcomplicating pipelines.

Combine with Secrets: Parameterize sensitive data with GitHub Secrets for security while still enabling flexibility.

Document Parameters: Clearly document what each parameter does and its acceptable values to aid understanding and maintainability.

END-TO-END INFRASTRUCTURE DEPLOYMENT PIPELINE

In this walkthrough, we will build a complete end-to-end infrastructure deployment pipeline using GitHub Actions and Terraform. This pipeline will automate the provisioning of infrastructure and deployment of resources in a secure and reusable manner.

What You Will Learn:

- How to set up a GitHub Actions pipeline for infrastructure deployment.
- Incorporating Terraform to automate provisioning tasks.
- Using parameters, environment variables, and credentials for flexibility and security.
- Testing, validating, and monitoring the pipeline to ensure reliability.

Key Features of the Pipeline:

- Triggered on changes to the repository.
- Fully parameterized for multi-environment support (e.g., staging, production).
- Securely manages sensitive credentials with GitHub Secrets.

CONFIGURE TERRAFORM STEPS

yaml

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2
        with:
          terraform_version: 1.4.0

      - name: Initialize Terraform
        run: terraform init

      - name: Plan Terraform
        run: terraform plan

      - name: Apply Terraform
        run: terraform apply -auto-approve
```

The pipeline should automate Terraform workflows by including three critical steps: initialize, plan, and apply. These steps streamline infrastructure provisioning and ensure consistency.

Initialize: Runs terraform init to set up the working directory, download provider plugins, and configure the backend for state management.

Plan: Executes terraform plan to preview infrastructure changes, showing additions, deletions, or modifications before applying them.

Apply: Runs terraform apply to provision or update resources like virtual machines, storage, or networking, ensuring the infrastructure matches the desired state.

SETTING UP THE PIPELINE

```yaml
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
```

Creating an end-to-end deployment pipeline starts with setting up the repository and defining the workflow. These foundational steps ensure a structured and reusable approach to infrastructure deployment.

Repository Configuration:

- **Create GitHub Repository:** Set up a repository to store all relevant Terraform configuration files (e.g., main.tf, variables.tf) and the workflow YAML file (.github/workflows/deploy.yml). This ensures that infrastructure code is version-controlled and accessible.

Workflow Triggers:

- **Define Pipeline Triggers:** Configure events to initiate the pipeline. For example, use a push event to automatically start the pipeline when changes are pushed to the main branch.
- **Scope Triggered Events:** Limit triggers to specific branches or tags (e.g., only the main branch) to prevent accidental or irrelevant workflow executions.

ENVIRONMENT SPECIFIC CONFIGURATION

yaml

```yaml
inputs:
  environment:
    description: 'Target environment'
    required: true

env:
  AWS_REGION: us-east-1

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Initialize Terraform
        run: terraform init -backend-config="env/${{ inputs.environment }}.tfstate"
```

To make the pipeline reusable and adaptable, incorporate parameters and environment-specific settings. These additions ensure workflows are flexible, handle different environments, and simplify updates without duplicating logic.

Parameterize Target Environments: Use GitHub Actions inputs or env to dynamically specify environments like staging or production. This approach allows the pipeline to adjust its behavior based on the chosen environment, such as stricter validation for production or experimental features for staging.

Environment Variables: Define variables for Terraform settings, such as AWS regions or instance types, to centralize configurations. By storing values like AWS_REGION or INSTANCE_TYPE in environment variables, the pipeline remains consistent across environments, reducing the need for code changes.

MANAGING CREDENTIaLS

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2
        with:
          terraform_version: 1.4.0

      - name: Apply Infrastructure
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        run: |
          terraform apply -auto-approve
```

To securely manage credentials in a deployment pipeline, incorporate GitHub Secrets and configure workflows to access them safely. This ensures sensitive data like API keys and cloud provider credentials are protected throughout the deployment process.

Store Secrets: Add credentials such as AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY in the repository's Secrets section to encrypt and restrict access.

Inject Secrets into Workflows: Use secrets within the workflow by mapping them to environment variables, allowing secure access during Terraform execution.

Lab: GitHub Actions