Open Source Tools of Kubernetes for DevOps- 1

Kubernetes has revolutionized how we deploy, manage, and scale containerized applications.
To support DevOps practices in Kubernetes environments, a rich ecosystem of open source tools has evolved. These tools streamline CI/CD, enable GitOps, enhance observability, enforce security policies, and simplify cluster operations. From deployment automation to monitoring, cost optimization, and governance, each tool is purpose-built to tackle specific challenges in the Kubernetes lifecycle. Leveraging these tools helps teams achieve agility, consistency, and resilience in managing cloud-native workloads.

Deployment & GitOps
Argo CD – Declarative GitOps continuous delivery tool for Kubernetes.
Flux CD – Another great GitOps tool that integrates tightly with Kubernetes.
Helmfile – Open-source tool for managing Helm charts.

CI/CD Integration
Jenkins X – Jenkins specifically for Kubernetes; automated CI/CD with GitOps.
Tekton – Kubernetes-native CI/CD pipelines with custom resources.
Drone CI – Open-source CI/CD tool built around containers.
CircleCI (Open Source) – Open-source version of CircleCI.

Monitoring & Observability
Prometheus
Grafana
Loki – Log aggregation system by Grafana Labs.
Jaeger – Prometheus Distributed tracing, useful for microservices debugging.

Thanos – Open-source tool extending Prometheus for highly available, scalable long-term storage.

Prometheus Operator – Open-source operator for managing Prometheus deployments on Kubernetes.

Security & Compliance

Trivy – Vulnerability scanner for container images, file systems, and Git repos.

Kube-bench – Checks your clusters against CIS Kubernetes security benchmarks.

Kubesec – Static analysis for Kubernetes manifests.

OPA (Open Policy Agent) – Enforce policies on Kubernetes resources.

Falco – Open-source runtime security monitoring tool for Kubernetes and containers.

Namespace & Resource Management

Kustomize – Customize Kubernetes YAMLs without forking.

Helm – Package manager for Kubernetes (charts).

Skaffold – Open-source tool for continuous development of Kubernetes applications.

Deployment & GitOps

Argo CD – Declarative GitOps continuous delivery tool for Kubernetes.

Argo CD is a **GitOps** tool that automates the deployment and management of applications in Kubernetes using a **declarative** approach. By syncing the **desired state** of applications stored in a Git repository with the live Kubernetes environment, Argo CD ensures that your applications are always in the correct state.

It constantly monitors the Git repository for any updates, and when changes are detected, it automatically applies them to the Kubernetes cluster. This means that your deployment configurations are always controlled by Git, providing a single source of truth.

Key Features:

- **Declarative setup**: Application deployment is defined as code, ensuring consistency and repeatability.

- **Continuous delivery**: Automatically applies changes from Git to your Kubernetes cluster.

- **Rollback made easy**: If needed, you can easily revert changes from Git.

- **Version control**: Keep track of all changes and deployments in Git.

- **Scalability**: Supports managing large, complex Kubernetes environments.

Argo CD simplifies Kubernetes application management by making deployments and updates automated, transparent, and reliable, all while leveraging Git as the source of truth.

Installation of Argo CD for DevOps

1. Install Argo CD in Kubernetes:

Create a Namespace for Argo CD: First, create a namespace where Argo CD will be installed:

```
kubectl create namespace argocd
```

Apply the Installation Manifest: Apply the installation manifest to deploy Argo CD in the created namespace:

```
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

2. Install Argo CD CLI:

Download the Argo CD CLI: You can download the Argo CD CLI from the official releases page or use Homebrew for macOS:

```
brew install argocd
```

Alternatively, you can download the CLI directly from Argo CD Releases.

3. Access Argo CD API Server:

You can expose the Argo CD API server using either a LoadBalancer or port forwarding.

Using LoadBalancer: If you want to expose Argo CD via a LoadBalancer, run the following command:

```
kubectl patch svc argocd-server -n argocd -p '{"spec": {"type": "LoadBalancer"}}'
```

After this, you can get the external IP address of the service:

```
kubectl get svc -n argocd
```

You can then access the Argo CD API server at the external IP address.

Using Port Forwarding: If you prefer using port forwarding, run the following command:

```
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

Your Argo CD server will now be accessible at http://localhost:8080.

4. Retrieve the Initial Password for the Admin Account:

To retrieve the initial password for the admin account, run the following command:

```
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d; echo
```

This will print the initial password that you can use to log in.

5. Login Using the CLI:

Now, you can log in to Argo CD using the CLI.

Login Command: Replace <ARGOCD_SERVER> with either the external IP (if using LoadBalancer) or localhost:8080 (if using port forwarding):

```
argocd login <ARGOCD_SERVER>
```

For example, if you are using port forwarding, the command would be:

```
argocd login localhost:8080
```

Enter Credentials:

Username: admin

Password: Use the password retrieved from the previous step.

After logging in successfully, you can start using Argo CD to manage your applications.

Argo CD UI: How to Use It

✅ Access Argo CD UI

If you exposed Argo CD via:

- **Port Forwarding**:
  Visit: http://localhost:8080

- **LoadBalancer**:
  Visit: http://<EXTERNAL-IP>:<PORT>

🔐 Login to Argo CD UI

- **Username:** admin

Password:
Retrieve it using this command:

```
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d; echo
```

🧭 Argo CD UI Overview

Once logged in, you'll see:

- **Applications Dashboard**
  Visual representation of each app’s sync status, health, and resources.

- **App Details View**
  Click on any app to:

  - View live deployment architecture

  - Inspect YAML of each resource

    - Sync or refresh the app
    - Roll back to a previous state
- **Sync Status**
    - ✅ Synced: Git and cluster are in sync.
    - ❌ OutOfSync: Git has changes not yet applied.
- **Health Status**
    - Healthy, Progressing, Degraded

✨ Basic Actions in UI

- **Create App**:
  Click + New App → Fill Git repo details, path, namespace, cluster, etc.
- **Sync App**:
  Click your app → SYNC button to manually sync with Git.
- **Rollback**:
  Click History and Rollback → choose an older commit → Rollback.
- **Delete App**:
  App view → click App Details → DELETE (bottom right).

💻 Argo CD CLI Commands

Make sure you're logged in:

```
argocd login localhost:8080
```

🚀 Create an Application

```
argocd app create nginx-app \
  --repo https://github.com/YOUR_USERNAME/argo-nginx-gitops.git \
  --path k8s-manifests \
  --dest-server https://kubernetes.default.svc \
  --dest-namespace default
```

🔄 Sync the Application

```
argocd app sync nginx-app
```

📊 Get App Status

```
argocd app get nginx-app
```

📜 List All Applications

```
argocd app list
```

🧪 Check App Health

```
argocd app health nginx-app
```

📚 View App History

```
argocd app history nginx-app
```

🔙 Rollback to Previous Revision

```
argocd app rollback nginx-app <REVISION_NUMBER>
```

🗑️ Delete Application

```
argocd app delete nginx-app --yes
```

🧠 Pro Tips

- Always keep the Git repo clean and organized.
  Enable **auto-sync** if you want Argo CD to apply changes without manual sync:

  ```
   argocd app set nginx-app --sync-policy automated
  ```

Flux CD – Flux CD is a tool that helps you **automate and manage application deployments** in Kubernetes using **Git**. It follows the **GitOps** approach, which means your application settings (like configuration files) are stored in a Git repository, and Flux CD continuously watches that repo for any updates.

When it sees a change—like a new version of your app or an updated configuration—it **automatically applies those changes to your Kubernetes cluster**. This way, what's in Git is always what's running in your cluster (this is called **"desired state"**).

Key points for beginners:

- **Git as the single source of truth**: Everything is controlled through Git.
- **Continuous delivery**: Changes are deployed automatically.
- **Kubernetes native**: Designed to work tightly with Kubernetes.

- **Easy rollback**: If something goes wrong, you can revert changes from Git.

Flux CD makes life easier for developers and DevOps engineers by ensuring deployments are consistent, traceable, and easy to manage.

Installing Flux CD

Set Up Kubernetes CLI:

```
brew install fluxcd/tap/flux
```

Install kubectl: Confirm installation:

```
kubectl version --client
```

Check Flux Requirements:

```
flux check --pre
```

Bootstrap Flux: Bootstrap Flux in your cluster:

```
flux bootstrap github \
  --owner=GITHUB_USER \
  --repository=GITHUB_REPO \
  --branch=main \
  --path=./clusters/my-cluster \
  --personal
```

🚀 Why Bootstrap Flux?

Bootstrapping Flux sets up:

- Flux components (like source-controller, kustomize-controller, etc.) inside your cluster.
- A GitOps pipeline that watches your specified GitHub repository.
- Automatic syncing of Kubernetes manifests from your repo to your cluster.

Essentially, you're telling Flux:

> “This is my GitHub repo. Watch this path and keep my cluster in sync with whatever is defined there.”

🌱 Flux CD Project: GitOps-based Todo App Deployment on Kubernetes

🎯 Project Goal

Use **Flux CD** to automatically deploy a simple **Todo web application** (e.g., React frontend + Flask or Node.js backend + MongoDB) to a local or cloud Kubernetes cluster. All deployment manifests are stored in Git and managed by Flux CD.

📁 Project Structure

```
gitops-todo-app/
├── clusters/
│   └── kind/
│       └── flux-sync.yaml       # Points to the app repo for Flux to sync
├── apps/
│   └── todo/
│       ├── kustomization.yaml
│       ├── deployment.yaml
│       ├── service.yaml
│       └── ingress.yaml
└── README.md
```

🧰 Tools Used

Flux CD
Kind (local K8s) or Minikube
GitHub (as the GitOps source)
Kustomize
Docker
Helm (optional for packaging)
Any app (e.g., Node.js + MongoDB or Flask app)

🔧 Steps to Build the Project

Step 1: Prepare Kubernetes Cluster

- Use **Kind** to create a local Kubernetes cluster.

Install Flux CD:

```
flux install
```

Step 2: Bootstrap GitOps Repo

- Create a new GitHub repository (e.g., gitops-todo-app).

Bootstrap your cluster with Flux:

```
flux bootstrap github \
  --owner=your-github-username \
  --repository=gitops-todo-app \
  --branch=main \
  --path=clusters/kind
```

Step 3: Deploy the Todo App

- Create Kubernetes manifests for your app in apps/todo/.
- Add a kustomization.yaml to manage deployment files.

Apply a Kustomization object in clusters/kind/flux-sync.yaml:

yaml

```
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: todo-app
  namespace: flux-system
spec:
  interval: 1m
  path: ./apps/todo
  prune: true
  sourceRef:
    kind: GitRepository
    name: flux-system
  targetNamespace: default
```

Step 4: Push Changes and Watch Flux Apply Them

- Make a change in apps/todo/deployment.yaml (e.g., image version).
- Commit and push to Git.
- Flux will detect and apply the change.

✅ What You'll Learn

How Flux CD syncs changes from Git to K8s
Structuring GitOps-friendly Git repositories
Using Kustomize with Flux
Rolling out updates automatically
Git-based rollback

🚀 1. Automated Deployment of a Todo App

Scenario:
You built a simple **Todo app** (React + Node.js + MongoDB) and want to deploy it on Kubernetes using GitOps.

Use Case:

- Store manifests in Git
- Use Flux to automatically deploy app updates when code/config changes
- Automatically roll out new Docker image versions

Skills Gained:
GitOps basics, Kubernetes manifest handling, automation through Git, continuous delivery.

🛠️ 2. Environment Promotion: Dev → Staging → Prod

Scenario:
You manage three environments (dev, staging, prod) and want to promote changes in a controlled way.

Use Case:

- Use branches or folders (dev/, staging/, prod/) in Git
- Flux watches each environment path separately
- Promote by opening PRs and merging to the next env

Skills Gained:
Multi-env GitOps, PR-based workflow, safe deployments.

🔁 3. Rolling Updates with Automatic Rollback

Scenario:
You want to roll out a new version of your app, but automatically roll back if something breaks.

Use Case:

- Flux applies the new version from Git
- Health checks fail? You revert the Git commit
- Flux reverts your cluster automatically

Skills Gained:
Safe deployments, understanding desired state, Git-based rollback.

🧩 4. Modular App Deployment with Kustomize

Scenario:
You have multiple microservices and want to deploy them independently but under one GitOps repo.

Use Case:

- Use Kustomize overlays for each microservice
- Create separate Flux Kustomization objects
- Manage individual components via Git

Skills Gained:
Component-based deployments, advanced Kustomize usage, separation of concerns.

🔐 5. Manage Kubernetes Secrets with GitOps

Scenario:
You want to store secrets securely and deploy them via Git.

Use Case:

- Use **Sealed Secrets** or **External Secrets Operator**
- Secrets are encrypted and stored in Git
- Flux decrypts and applies them in the cluster

Skills Gained:
Secure secret management, integrating secrets with GitOps.

📦 6. Deploying Helm Charts with Flux

Scenario:
Your app is packaged as a Helm chart and you want GitOps-style control over it.

Use Case:

- Define HelmRepository and HelmRelease in Flux
- Push version updates to Git
- Flux installs/upgrades Helm charts automatically

Skills Gained:
Helm + Flux integration, chart-based deployments.

📊 7. Monitoring GitOps Deployments

Scenario:
You want visibility into what Flux is doing in your cluster.

Use Case:

- Use **Flux UI**, Grafana, or logs
- Observe syncs, failures, successes
- Monitor drift and sync status

Skills Gained:
Observability, debugging GitOps issues, understanding Flux controller behavior.

Helmfile – Open-source tool for managing Helm charts.

Helmfile is a free, open-source tool that helps you manage and use **Helm charts** more easily. Helm charts are like templates used to install applications in **Kubernetes**.

Instead of installing each Helm chart one by one, Helmfile lets you **list all your Helm charts in one YAML file**. This way, you can install or update everything at once. It helps you:

- Keep things organized
- Work with different environments (like dev, test, or production)
- Save your setup in version control (like Git)
- Make your deployments easier and faster

Install Helmfile

For macOS/Linux (Homebrew):

```
brew install helmfile
```

For Linux (Download binary):

```
wget https://github.com/helmfile/helmfile/releases/download/v0.156.0/helmfile_0.156.0_linux_amd64.tar.gz
tar -zxvf helmfile_0.156.0_linux_amd64.tar.gz
sudo mv helmfile /usr/local/bin/
```

Build from Source (requires Go):

```
git clone https://github.com/helmfile/helmfile.git
cd helmfile
make build
sudo mv ./bin/helmfile /usr/local/bin/
```

Verify Installation:

```
helmfile --version
```

🚀 Full Step-by-Step Guide: Deploy Grafana + MySQL with Helmfile on AWS EKS

🛠️ Prerequisites

Ensure you have these tools installed:

```
aws --version
eksctl version
kubectl version --client
helm version
helmfile version
```

If anything's missing, let me know — I can help with install commands too.

🔹 Step 1: Create EKS Cluster

```
# Use eksctl to create an EKS cluster in the us-east-1 region with 2 nodes of type t3.medium
eksctl create cluster \
  --name devops-demo \            # The name of the cluster
  --region us-east-1 \            # AWS region where the cluster will be created
  --nodes 2 \                     # Number of worker nodes
  --node-type t3.medium \         # Instance type for worker nodes
  --managed                       # Managed node group (AWS handles scaling, patching, etc.)
```

This step creates the EKS cluster and takes 10–15 minutes to complete. AWS sets up all the required infrastructure, such as VPC, IAM roles, etc.

🔹 Step 2: Verify Cluster Access

```
# Ensure your kubeconfig is updated so kubectl can interact with your new cluster
aws eks --region us-east-1 update-kubeconfig --name devops-demo

# Verify the worker nodes are up and running
kubectl get nodes
```

This command updates the kubeconfig and checks that the worker nodes are in a ready state.

🔹 Step 3: Prepare Helmfile Project Locally

Create a new directory and navigate into it:

```
mkdir helmfile-grafana-mysql && cd helmfile-grafana-mysql
mkdir -p environments values
```

Now, create the following files and add the corresponding configurations:

File: environments/dev.yaml
yaml

```
namespace: dev  # Define the namespace for the dev environment
```

This file specifies the namespace for the dev environment.

File: values/grafana-values.yaml

yaml

```
adminUser: admin
adminPassword: admin123

service:
  type: LoadBalancer  # Exposing Grafana via LoadBalancer

persistence:
  enabled: false  # Disable persistence for this example (temporary data)
```

This configures Grafana with admin credentials and exposes it using a LoadBalancer.

File: values/mysql-values.yaml

yaml

```
auth:
  rootPassword: rootpass
  database: mydb
  username: myuser
  password: mypass

primary:
  service:
    type: ClusterIP  # MySQL service will only be accessible within the cluster
```

This file sets up MySQL with user authentication and a predefined database.

File: helmfile.yaml

yaml

```
environments:
  dev:
    values:
      - environments/dev.yaml  # Point to the environment configuration

releases:
  - name: grafana
    namespace: monitoring  # Grafana will be deployed in the monitoring
namespace
    chart: grafana/grafana
    version: 7.3.7
    values:
      - values/grafana-values.yaml  # Use Grafana's specific configuration

  - name: mysql
    namespace: database  # MySQL will be deployed in the database namespace
    chart: bitnami/mysql
    version: 9.4.3
    values:
      - values/mysql-values.yaml  # Use MySQL's specific configuration
```

The Helmfile defines the configuration for both Grafana and MySQL, specifying namespaces, charts, versions, and values files.

Step 4: Add Helm Repos

```
# Add the Grafana and Bitnami Helm repositories to fetch charts from them
helm repo add grafana https://grafana.github.io/helm-charts
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update  # Update the Helm repositories to get the latest charts
```

This step adds the required Helm repositories for Grafana and Bitnami charts.

🔹 Step 5: Create Namespaces (optional – Helm can auto-create)

```
# Creating Kubernetes namespaces for Grafana and MySQL deployment
kubectl create ns monitoring  # Namespace for Grafana
kubectl create ns database    # Namespace for MySQL
```

You can optionally create namespaces manually. Helm can also create them automatically.

🔹 Step 6: Apply Helmfile

```
# Apply the Helmfile to deploy Grafana and MySQL into your EKS cluster
helmfile -e dev apply  # Apply the configuration for the 'dev' environment
```

This applies the Helmfile, which deploys both Grafana and MySQL into the EKS cluster using the configurations you defined.

🔹 Step 7: Check Resources

```
# Check all the resources (pods, services, etc.) in the monitoring and database namespaces
kubectl get all -n monitoring
kubectl get all -n database
```

Use these commands to verify that your Grafana and MySQL services are deployed successfully.

🔹 Step 8: Access Grafana

```
# Get the EXTERNAL-IP of the Grafana service to access it via browser
kubectl get svc -n monitoring

# Login to Grafana with the credentials you set earlier
# Username: admin
# Password: admin123
```

Find the external IP of the Grafana service and use it to access Grafana in your browser.

🔹 Step 9: Cleanup (Optional)

```
# If you want to clean up your resources after testing, run the following commands
helmfile -e dev destroy  # Remove deployed resources using Helmfile

# Delete the EKS cluster to free up resources
eksctl delete cluster --name devops-demo --region us-east-1
```

Clean up the EKS cluster and the deployed resources when you're done.

✅ Summary

You have successfully:

- Created an EKS cluster
- Deployed Grafana and MySQL using Helmfile
- Exposed Grafana externally with a LoadBalancer

🔹 Use Case 1: Manage Multiple Helm Charts Together

Problem:
You’re deploying multiple services (like Grafana and MySQL), and managing them separately with individual helm install commands becomes error-prone.

How Helmfile Helps:
Helmfile lets you define multiple Helm charts in one declarative YAML (helmfile.yaml), apply them together, and version control everything.

Example:
Deploy Grafana and MySQL in one go using:

```
helmfile apply
```

🔹 Use Case 2: Consistent Environments Across Dev, Staging, and Prod

Problem:
You want different configs for dev, staging, and production, but keeping values files in sync is hard.

How Helmfile Helps:
Helmfile supports **environments** (like dev, prod) where you can define custom values for each.

Example:

```
helmfile -e dev apply
helmfile -e prod apply
```

And each environment can have its own values/*.yaml or environments/*.yaml.

🔹 Use Case 3: GitOps / CI-CD Integration

Problem:
You want to automate deployments via Jenkins/GitHub Actions in a declarative and version-controlled way.

How Helmfile Helps:
You check in the helmfile.yaml into your Git repo, then CI tools just run:

```
helmfile apply
```

This ensures your entire state is Git-controlled and repeatable.

🔹 Use Case 4: Declarative Infrastructure-as-Code for Helm

Problem:
Helm CLI is imperative (helm install, helm upgrade) — you can't just read a file and know what will be deployed.

How Helmfile Helps:
With helmfile.yaml, you **declaratively define** all your Helm releases, versions, namespaces, and values. Easier for teams to audit and review.

🔹 Use Case 5: Simplified Rollbacks

Problem:
Managing rollback history and reverting changes across multiple Helm releases is hard manually.

How Helmfile Helps:
You can use helmfile diff to preview changes **before** applying and helmfile sync or destroy to rollback environments cleanly.

🔹 Use Case 6: Managing Third-Party Applications

Problem:
You're deploying third-party services like Prometheus, Grafana, NGINX Ingress, etc., and you want to automate the whole stack.

How Helmfile Helps:
You add official charts (like Bitnami, Grafana) to helmfile.yaml and configure everything centrally.

🔹 Use Case 7: DRY (Don't Repeat Yourself) Config Management

Problem:
You're repeating common values or settings (e.g., namespaces, chart versions) across multiple helm install commands.

How Helmfile Helps:
You can use yaml anchors, reusable value files, shared config in one place—making your setup DRY and maintainable.

🔹 Use Case 8: Managing Infrastructure on EKS/GKE/AKS

Problem:
You need to install a stack of monitoring/logging/storage tools after creating a Kubernetes cluster.

How Helmfile Helps:
Run a single command to bring up all components like:

- Grafana
- MySQL/Postgres

- Loki
- Prometheus
- NGINX Ingress
- Cert-Manager

With one config file, deploy it all in a clean, idempotent way.

🔹 Use Case 9: Team Collaboration

Problem:
Your team needs a shared, consistent way to deploy Helm charts.

How Helmfile Helps:
All configurations are in Git. Anyone can clone the repo, run helmfile apply, and get the exact same environment setup.

🔹 Use Case 10: Previewing and Auditing Changes

Problem:
You want to know **what will change** before deploying.

How Helmfile Helps:
Run:

```
helmfile diff
```

to see changes before applying them. Great for controlled deployments and avoiding surprises in production.

CI/CD Integration

Jenkins X – Jenkins specifically for Kubernetes; automated CI/CD with GitOps.

Jenkins X is a free, open-source tool that helps you **automate the process of building, testing, and deploying applications** — especially in **Kubernetes** environments.

Think of it as a smarter version of Jenkins, but built specifically for **cloud-native** apps that run on Kubernetes.

With Jenkins X, you can:

- Automatically build and test your code every time you make a change
- Deploy your app to different environments (like dev, staging, or production)
- Use **GitOps**, which means your code and deployment settings are stored and managed in Git

It also creates **automatic preview environments** for each branch or pull request, so you can test changes before merging them.

In short, Jenkins X helps you **speed up software delivery** and **manage Kubernetes deployments more easily and automatically**.

Install Jenkins X (JX)

Step 1: Install Prerequisites

Make sure these tools are installed:

kubectl
helm
git
docker

Also, ensure you have access to a Kubernetes cluster.

Step 2: Install Jenkins X CLI

For Linux:

```
curl -L https://github.com/jenkins-x/jx/releases/download/v3.10.118/jx-linux-amd64.tar.gz | tar xzv
sudo mv jx /usr/local/bin
```

For macOS:

```
brew install jenkins-x/jx/jx
```

Step 3: Verify Installation

```
jx version
```

Step 4: Install Jenkins X on the Cluster

```
jx admin operator
```

📌 Project Name: jenkins-x-kind-hello

🔰 Introduction

This project demonstrates how to set up a Jenkins X environment on a Kubernetes cluster using kind and deploy a simple Python Flask "Hello World" app with GitOps workflows. It focuses on automating CI/CD for cloud-native applications.

🧱 Project Structure

```
jenkins-x-kind-hello/
├── hello-app/
│   ├── app.py
│   ├── Dockerfile
│   ├── requirements.txt
├── .lighthouse/
│   └── jenkins-x/
│       └── triggers.yaml
├── jenkins-x.yml
├── skaffold.yaml
└── README.md
```

💻 hello-app/app.py

This file contains a simple Python Flask application.

python

```
from flask import Flask
```

```
# Initialize Flask app

app = Flask(__name__)


# Define the route for the home page

@app.route('/')

def hello():

    return "Hello from Jenkins X on kind!"


# Run the app

if __name__ == "__main__":

    app.run(host="0.0.0.0", port=5000)
```

Explanation:

The Flask module creates a basic web server.
The route '/' responds with a simple string: "Hello from Jenkins X on kind!".
The app runs on all available network interfaces (0.0.0.0) on port 5000.

🐳 hello-app/Dockerfile

This file defines the Docker image for the Flask app.

dockerfile

```
FROM python:3.9
```

```
# Set the working directory inside the container

WORKDIR /app


# Copy all files from the current directory into the container

COPY . .


# Install required dependencies

RUN pip install flask


# Expose port 5000 to the outside world

EXPOSE 5000


# Define the command to run the app

CMD ["python", "app.py"]
```

Explanation:

A Python 3.9 image is used as the base.
The current directory is copied into the container.
The flask dependency is installed using pip.
The container listens on port 5000 (configured by Flask).
The container runs the app.py file when started.

📦 hello-app/requirements.txt

This file lists the Python dependencies required for the app.

```
nginx
flask
```

Explanation:

- This file ensures that Flask is installed in the container. It could be expanded to include other dependencies as needed.

⚙️ skaffold.yaml

This file is used by Skaffold to automate building and deploying the app.

yaml

```
apiVersion: skaffold/v2beta10
kind: Config
metadata:
  name: hello-app

build:
  artifacts:
    - image: hello-app
      context: hello-app


deploy:
  helm:
    releases:
      - name: hello-app
        chartPath: charts/hello-app
        values:
          image:
            repository: hello-app
```

Explanation:

- The skaffold.yaml configures the build and deployment process.
- It specifies that the image hello-app will be built from the hello-app/ directory.
- The app will be deployed using Helm, which is a Kubernetes package manager.

🚦 jenkins-x.yml (CI/CD pipeline config)

This file defines the pipeline that Jenkins X uses for CI/CD.

yaml

```
name: ci
on:
  pull_request:
    branches:
      - '*'
  push:
    branches:
      - main

pipelineConfig:
  pipelines:
    pullRequest:
      pipeline:
        stages:
          - name: Build and Preview
            steps:
              - name: build
                image: ghcr.io/jenkins-x/jx-boot:3.10.92
                command: jx
                args:
                  - step
                  - build
                  - --preview
    release:
      pipeline:
        stages:
          - name: Release
            steps:
              - name: release
                image: ghcr.io/jenkins-x/jx-boot:3.10.92
                command: jx
                args:
                  - step
                  - release
```

Explanation:

- This file defines Jenkins X pipelines for different Git events.
- The pullRequest section specifies that on every pull request, Jenkins X will build and create a preview environment.

- The release section defines the steps to release the app when changes are merged.

🚀 Deployment Steps

Follow these steps to deploy the app using Jenkins X on a kind Kubernetes cluster.

Install Prerequisites Install Docker, Kubernetes CLI, kind, and Jenkins X CLI.

```
 sudo apt install docker.io kubectl
```

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.22.0/kind-linux-amd64 && chmod +x ./kind && sudo mv ./kind /usr/local/bin/
```

```
curl -L https://github.com/jenkins-x/jx/releases/latest/download/jx-linux-amd64.tar.gz | tar xzv
```

```
sudo mv jx /usr/local/bin
```

Create a Kubernetes cluster with kind This will set up a Kubernetes cluster locally.

```
kind create cluster --name jx-cluster --image kindest/node:v1.24.0
```

Bootstrap Jenkins X Run this command to set up Jenkins X with GitOps in your Kubernetes cluster.

```
jx admin operator \

  --cluster-name jx-kind \
  --git-username=<your-github-username> \
  --git-token=<your-github-token> \
  --git-url=https://github.com/<your-username>/jx3-kubernetes \
  --domain=127.0.0.1.nip.io
```

Import Your App into Jenkins X Import the jenkins-x-kind-hello project into Jenkins X.

```
jx project import
--url=https://github.com/<your-username>/jenkins-x-kind-hello.git
```

Preview and Promote Jenkins X will automatically create preview environments for each PR. You can merge the PR to promote it to the dev environment or manually promote it to staging.

```
jx promote hello-app --env=staging
```

Access the App Use the following command to get the service URL.

```
kubectl get ingress -n jx
```

Example output:

```
http://hello-app.jx.127.0.0.1.nip.io/
```

🎯 Outcome

By following this project, you will have:

Set up a local Jenkins X environment using kind.
Built and deployed a Python Flask app with Jenkins X CI/CD pipelines.
Gained experience with GitOps workflows.
Learned how to use preview and promotion environments in Jenkins X.

---

1. CI/CD for Microservices-Based Applications

Use Case: In modern software development, microservices architectures are commonly used to break down complex applications into smaller, manageable services. Jenkins X, with its CI/CD pipeline setup, is perfect for automating the deployment and updates of microservices.

Scenario: A company is developing an e-commerce platform where each module (e.g., cart, payment, and user management) is a separate microservice. Jenkins X automates the build, test, and deployment of each microservice to Kubernetes clusters, ensuring that updates are seamlessly integrated into production.

How it Works:

- The hello-app example app can be extended to multiple microservices.
- The CI/CD pipeline triggers builds for each service on commits.
- The GitOps flow ensures that all configurations are versioned and easily replicated.

---

2. GitOps Workflow with Jenkins X

Use Case: GitOps is a modern approach where the state of your infrastructure and applications is stored in Git repositories. Jenkins X supports GitOps by automatically deploying changes whenever a pull request (PR) is merged.

Scenario: A company wants to ensure that their deployment process is fully automated and tracked in a Git repository, reducing the risk of human error. The team uses Jenkins X to implement GitOps in their Kubernetes environment.

How it Works:

- The hello-app is stored in a GitHub repository.
- Jenkins X watches the repository and triggers a build and deployment pipeline every time code is pushed or a PR is merged.
- The state of the app, along with all configurations (e.g., Helm charts), is tracked in Git.

---

3. Automating Testing and Deployment for Open-Source Projects

Use Case: Open-source projects often need automated testing and deployment pipelines to make contributions easier and maintain code quality.

Scenario: An open-source project using Flask for the backend wants to ensure that every pull request is automatically tested and deployed to a preview environment before being merged into the main branch.

How it Works:

- Every time a contributor opens a pull request, Jenkins X automatically triggers the build and deploys the changes to a preview environment.
- The contributors can review the changes at a dedicated preview URL, ensuring that the new changes do not break the app before they are merged.
- Once the PR is reviewed, it can be merged, triggering the pipeline to deploy to the production environment.

4. Multi-Environment Deployment Strategy (Dev, Staging, Production)

Use Case: Using Jenkins X's pipelines, teams can automate the deployment process across different environments (Dev, Staging, Production), ensuring consistency across the development lifecycle.

Scenario: A software development team is working on a web application and needs to deploy to different environments for development, staging, and production. The team uses Jenkins X to handle the CI/CD pipeline for each environment.

How it Works:

- Jenkins X automatically deploys to a dev environment upon merging code into the main branch.
- Once the feature is fully tested, it is promoted to the staging environment for final checks.
- Once verified in staging, it can be manually promoted to production using the jx promote command.

---

5. Scaling Kubernetes Deployments with Jenkins X

Use Case: Large-scale deployments often need automatic scaling to handle traffic spikes. Jenkins X helps manage and scale Kubernetes deployments based on the defined pipelines.

Scenario: A startup has an application with fluctuating traffic patterns. They need to automatically scale their app based on incoming requests without manual intervention.

How it Works:

- The hello-app is deployed on a Kubernetes cluster with Horizontal Pod Autoscaling (HPA) configured.
- Jenkins X's CI/CD pipeline ensures that new app versions are automatically deployed.
- Kubernetes scales the app automatically based on defined resource metrics, ensuring high availability during traffic spikes.

---

6. Testing and Continuous Delivery for Python Applications

Use Case: For Python applications like web apps or microservices, automating the testing and deployment pipeline ensures that updates are smoothly delivered with reduced risk.

Scenario: A Python-based web application is under active development. Developers want to ensure that every new feature or bug fix is automatically tested and deployed without manual intervention.

How it Works:

- Jenkins X runs tests automatically on every push to the repository, ensuring that the application functions correctly before it's deployed.
- The pipeline includes automated linting, unit tests, integration tests, and deployment to a test/staging environment.
- The app is then deployed to production after approval in staging, minimizing downtime and manual effort.

---

7. Onboarding New Developers to CI/CD Processes

Use Case: A development team wants to onboard new developers and teach them best practices for CI/CD pipelines, GitOps, and Kubernetes deployments using a practical project.

Scenario: New team members are unfamiliar with CI/CD tools like Jenkins X and Kubernetes. The team uses this project as an introductory course to teach new hires how to automate builds, tests, and deployments.

How it Works:

- The project provides a simple Flask app with Jenkins X already configured for CI/CD.

- New developers can follow the setup instructions to get the app running on their local Kubernetes cluster (kind).

- They can then experiment with modifying the app, making changes, and seeing how Jenkins X automates the build and deployment pipeline.

---

8. Automating Infrastructure as Code (IaC) Deployment

Use Case: Infrastructure as Code (IaC) allows teams to manage infrastructure using code. Jenkins X can help automate the deployment of IaC configurations using CI/CD pipelines.

Scenario: A team is managing their Kubernetes infrastructure with Helm charts and Kubernetes manifests stored in Git. They want Jenkins X to automatically deploy their infrastructure changes whenever a Git commit is made.

How it Works:

- Infrastructure configurations (like Helm charts) are stored in the same Git repository as the application code.

- Jenkins X triggers deployments of both the infrastructure and application whenever changes are made to the Git repository.
- This ensures that the infrastructure is always in sync with the application code.

---

9. Disaster Recovery with Jenkins X Backups

Use Case: Ensuring disaster recovery for applications is critical. Jenkins X can automate backups of critical deployment configurations and application states, providing a robust disaster recovery mechanism.

Scenario: A financial service app needs to ensure that in the event of a disaster, the app's state can be restored quickly. Jenkins X helps manage backup and recovery processes for Kubernetes applications.

How it Works:

- Kubernetes cluster state, application configurations, and deployments are version-controlled using Git.
- In case of failure, the Jenkins X pipeline can redeploy the app from the latest known good configuration stored in Git.

---

comparison between **Jenkins** and **Jenkins X** in a chart format:

Aspect	Jenkins	Jenkins X
Core Focus	Traditional CI/CD with flexibility	Cloud-native CI/CD with GitOps integration

Deployment	Runs on VMs, containers, or servers	Runs on Kubernetes
Pipeline Configuration	Jenkinsfiles (Declarative or Scripted)	YAML-based pipelines with Tekton
GitOps Support	Requires custom setup or plugins	Native GitOps workflow
Environment Management	Manual setup for environments	Automated environment creation
Scaling	Manual scaling of build agents	Automatic scaling with Kubernetes
Helm Chart Integration	Needs plugins or manual configuration	Native Helm support
Multi-cloud/Cluster	Requires manual configuration	Built-in multi-cluster and multi-cloud support
Ease of Use	Steeper learning curve for beginners	Optimized for Kubernetes and cloud-native apps

---

The choice between **Jenkins** and **Jenkins X** depends on your specific use case, project requirements, and the environment you're working in. Here are some factors to consider when determining which one is "better" for you:

When Jenkins is Better:

- **Traditional Infrastructure**: If you're working with legacy systems or traditional on-premise infrastructure (VMs, physical servers), Jenkins might be a better fit due to its flexibility and wide compatibility.

- **Customizable Workflows**: Jenkins is highly customizable and supports various types of integrations. If your project requires highly customized workflows or you need to integrate with many different tools, Jenkins can be tailored to meet these needs.

- **Non-Kubernetes Environments**: If you're not using Kubernetes or don't plan to, Jenkins provides more flexibility in terms of environments and infrastructure.

- **Mature Ecosystem**: Jenkins has been around for a long time and has a vast community with a large number of plugins and integrations for different tools and platforms.

- **Broader Support for Non-Cloud-Native Projects**: Jenkins is better suited for monolithic applications and non-cloud-native projects.

When Jenkins X is Better:

- **Cloud-Native and Kubernetes Environments**: Jenkins X is specifically designed for Kubernetes and cloud-native applications. If your project is built with microservices, containers, and you are using Kubernetes for orchestration, Jenkins X is optimized for that environment.

- **GitOps Workflow**: If you're following GitOps principles (where Git is the single source of truth), Jenkins X provides built-in support for GitOps, making it easier to implement and automate deployments based on Git changes.

- **Automated Setup and Scaling**: Jenkins X simplifies the setup process and automates many tasks, including scaling and environment management, making it more efficient if you're running a cloud-native, microservices-based application.

- **Multi-Cluster / Multi-Cloud**: Jenkins X has built-in support for managing deployments across multiple Kubernetes clusters and cloud environments, which is beneficial for more complex or large-scale applications.

- **Preview Environments**: Jenkins X automatically generates preview environments for every pull request, which is great for teams working in a fast-paced development cycle.

In Summary:

- If you are working with **cloud-native applications, Kubernetes**, and want to use **GitOps**, **Jenkins X** is a better choice due to its native Kubernetes integration, automatic environment management, and streamlined deployment workflows.

- If you're working in a **traditional** environment or need more **flexibility** and **customization** with a large ecosystem of plugins, **Jenkins** might be a better fit.

Final Decision:

- **Jenkins X** is ideal for modern, cloud-native workflows in Kubernetes environments.

- **Jenkins** remains a great choice for more flexible, traditional CI/CD pipelines where extensive customization and non-Kubernetes environments are required.

Tekton – Kubernetes-native CI/CD pipelines with custom resources.
Tekton – Kubernetes-native CI/CD Pipelines with CRDs

Tekton is a powerful tool that helps you **automate your software delivery process** (CI/CD – Continuous Integration and Continuous Deployment) **directly inside Kubernetes**. What makes it special is that it uses **Custom Resource Definitions (CRDs)**—a way to create custom Kubernetes objects.

With Tekton, you can define all your pipeline steps—like **building, testing, and deploying your app**—as part of your Kubernetes configuration. This means your CI/CD setup becomes part of your Kubernetes system, making it easier to manage and scale. It's **cloud-native**, highly **flexible**, and great for teams already working with Kubernetes.

Install Tekton Pipelines:

```
kubectl apply --filename https://storage.googleapis.com/tekton-releases/pipeline/latest/release.yaml
```

(Optional) Install Tekton CLI (tkn):

```
# For Linux

curl -LO https://github.com/tektoncd/cli/releases/latest/download/tkn_0.36.0_Linux_x86_64.tar.gz

tar xvzf tkn_0.36.0_Linux_x86_64.tar.gz -C /usr/local/bin tkn
```

Verify installation:

```
kubectl get pods --namespace tekton-pipelines

tkn version
```

🛠️ Project: CI/CD with Tekton on Kubernetes

Goal: Automate the process of building, testing, and deploying a simple app using Tekton Pipelines in a Kubernetes cluster.

🧠 What You’ll Learn

- Basics of Tekton (Tasks, Pipelines, PipelineRuns, etc.)
- How to write YAML files for Tekton CRDs
- Running CI/CD pipelines in a Kubernetes-native way
- Deploying apps automatically after code changes

🔧 Prerequisites

- Kubernetes cluster (use Kind or Minikube for local setup)
- kubectl installed
- Tekton Pipelines installed
- Basic Docker knowledge
- Sample app (we’ll use a simple Node.js or Python Flask app)

📁 Project Structure

Your app repo could look like this:

tekton-cicd-project/

```
├── app/
│   └── main.py (or app.js)
├── Dockerfile
├── k8s-deployment.yaml
├── tekton/
│   ├── task-build.yaml
│   ├── task-push.yaml
│   ├── task-deploy.yaml
│   ├── pipeline.yaml
│   └── pipelinerun.yaml
```

📦 Step-by-Step

Step 1: Install Tekton Pipelines

```
kubectl apply --filename
https://storage.googleapis.com/tekton-releases/pipeline/latest/release.yaml
```

Step 2: Create a Task to Build Docker Image

yaml

```
# tekton/task-build.yaml

apiVersion: tekton.dev/v1

kind: Task
```

```
metadata:
  name: build-docker-image
spec:
  steps:
    - name: build
      image: docker:20.10
      script: |
        #!/bin/sh
        docker build -t myapp:latest .
```

Step 3: Create a Task to Push to Docker Hub

yaml

```
# tekton/task-push.yaml
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: push-docker-image
spec:
  steps:
    - name: push
```

```
      image: docker:20.10
      script: |
        #!/bin/sh
        docker tag myapp:latest <your-dockerhub-username>/myapp:latest
        docker push <your-dockerhub-username>/myapp:latest
```

Step 4: Create a Task to Deploy to Kubernetes

yaml

```
# tekton/task-deploy.yaml
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: deploy-app
spec:
  steps:
    - name: deploy
      image: bitnami/kubectl
      script: |
        #!/bin/sh
        kubectl apply -f k8s-deployment.yaml
```

Step 5: Define the Pipeline

yaml

```
# tekton/pipeline.yaml
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: myapp-pipeline
spec:
  tasks:
    - name: build
      taskRef:
        name: build-docker-image
    - name: push
      runAfter: [build]
      taskRef:
        name: push-docker-image
    - name: deploy
      runAfter: [push]
      taskRef:
        name: deploy-app
```

Step 6: Trigger the PipelineRun

yaml

```yaml
# tekton/pipelinerun.yaml
apiVersion: tekton.dev/v1
kind: PipelineRun
metadata:
  name: myapp-pipeline-run
spec:
  pipelineRef:
    name: myapp-pipeline
```

Apply all the Tekton CRDs:

```
kubectl apply -f tekton/
```

✅ Final Result

Every time you trigger the PipelineRun, Tekton will:

1. Build the Docker image

2. Push it to Docker Hub

3. Deploy it to your Kubernetes cluster

✅ Use Cases for Tekton – Kubernetes-native CI/CD

1. Build and Deploy a Web Application Automatically

Scenario: You have a Node.js or Python app in GitHub.
Use Tekton for:

- Cloning the repo
- Building a Docker image
- Pushing the image to Docker Hub
- Deploying the app to Kubernetes

🧩 *Good for learning CI/CD basics and pipeline flow*

---

2. Run Tests Automatically Before Deployment

Scenario: You want to ensure your app works before deploying it.
Use Tekton for:

- Running unit or integration tests (e.g., pytest or npm test)
- Only deploying if tests pass

🧪 *Great use case for adding test automation into the pipeline*

---

3. Multi-environment Deployment (Dev → Staging → Prod)

Scenario: You want to promote changes from development to production.
Use Tekton for:

- Creating pipelines that deploy to a dev namespace
- Manual or automatic promotion to staging and production namespaces

🔁 *Teaches environment separation, approvals, and versioning*

---

4. Event-driven CI/CD with GitHub Webhooks + Tekton Triggers

Scenario: Automatically run pipeline when code is pushed to GitHub.
Use Tekton for:

- Listening to push events using Tekton Triggers
- Dynamically running pipelines with latest commit

🔔 *Helps understand event-based pipeline automation*

---

5. Scan Code or Images for Vulnerabilities (DevSecOps)

Scenario: Security scanning as part of CI/CD.
Use Tekton for:

- Running tools like Trivy or OWASP Dependency-Check as a Task

- Failing the pipeline if vulnerabilities are found

🔐 *Good starting point for DevSecOps practices*

---

6. Build Container Images without Docker Daemon (Kaniko)

Scenario: You want to build images in a Kubernetes cluster without Docker installed.
Use Tekton for:

- Using Kaniko Task to build/push images
- More secure and Kubernetes-native

🐳 *Perfect for Kubernetes environments without Docker installed*

---

7. Deploy Helm Charts via Tekton

Scenario: Your apps are packaged as Helm charts.
Use Tekton for:

- Writing Tasks to run helm upgrade --install
- Automating Helm deployments as part of pipeline

🎯 *Useful when working with microservices or chart-based apps*

Drone CI – Open-source CI/CD tool built around containers.

Drone CI is a free, open-source tool that helps you automatically build and deploy your code. It uses **containers** (like small, lightweight virtual machines) to run each step of your work.

You just write the steps you want to run in a simple **YAML file** (a text file with easy-to-read format).

Drone can connect with your code on **GitHub, GitLab, or Bitbucket**, and it runs each step inside a **Docker container**.

It's **lightweight**, **easy to install**, and a great choice if you're new to CI/CD and working with containers.

Basic installation steps:

1. **Create a .env file with your secrets:**

env

```
DRONE_GITEA_SERVER=https://try.gitea.io
DRONE_RPC_SECRET=supersecret
DRONE_SERVER_HOST=yourdomain.com
DRONE_SERVER_PROTO=https
```

2. **Create docker-compose.yml:**

yaml

```
version: '3'

services:
  drone-server:
    image: drone/drone:latest
    ports:
      - 80:80
      - 443:443
    volumes:
      - ./drone:/data
    env_file:
      - .env

  drone-runner:
    image: drone/drone-runner-docker:latest
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    environment:
      - DRONE_RPC_PROTO=https
      - DRONE_RPC_HOST=drone-server
      - DRONE_RPC_SECRET=supersecret
```

3. **Start the services:**

   ```
   docker-compose up -d
   ```

Drone CI project This is perfect for someone just starting out with **CI/CD, Docker, and Drone CI**. 🌱

📁 Folder Structure

```
drone-flask-demo/
├── app.py             # Your Flask web application
├── requirements.txt   # Python dependencies
├── Dockerfile         # Docker image definition
├── .drone.yml         # Drone CI pipeline configuration
```

```
└── tests/
    └── test_app.py      # A simple test
```

1️⃣app.py – Your Flask Application

python

```python
# Import Flask module
from flask import Flask

# Create a Flask web application
app = Flask(__name__)

# Define a route for the homepage
@app.route("/")
def home():
    return "Hello from Drone CI!"  # This message will show in the browser or API

# Start the app when this file runs
if __name__ == "__main__":
    # Run on all network interfaces (0.0.0.0) and port 5000
    app.run(host="0.0.0.0", port=5000)
```

2️⃣requirements.txt – Dependencies

text

```text
# Flask is used to create a web app
flask

# Pytest is used for testing your app
pytest
```

3️⃣ tests/test_app.py – Simple Test

```python
# A simple test case for learning
def test_home():
    # Check if the string contains "Drone"
    assert "Drone" in "Hello from Drone CI!"
```

4️⃣ Dockerfile – To Build Docker Image

```dockerfile
# Use Python 3.10 slim version as the base image
FROM python:3.10-slim

# Set working directory in container
WORKDIR /app

# Copy the dependency list to container and install
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copy all files to the container
COPY . .

# Run the app when the container starts
CMD ["python", "app.py"]
```

5️⃣.drone.yml – Drone CI Pipeline File

yaml

```yaml
# Tell Drone this is a Docker-based pipeline
kind: pipeline
type: docker
name: default

steps:
  # Step 1: Install Python packages
  - name: install dependencies
    image: python:3.10
    commands:
      - pip install -r requirements.txt

  # Step 2: Run your test
  - name: run tests
    image: python:3.10
    commands:
      - pytest tests/

  # Step 3: Build and push Docker image to Docker Hub
  - name: docker build and push
    image: plugins/docker
    settings:
      repo: your-dockerhub-username/drone-flask-demo  # Change this!
      tags: latest
      username:
        from_secret: docker_username
      password:
        from_secret: docker_password
```

📝 **Change** your-dockerhub-username to your Docker Hub username.

🔐 Add Secrets in Drone

Use Drone CLI or UI to add your Docker Hub credentials securely:

```
drone secret add \
  --repository your-github-username/drone-flask-demo \
  --name docker_username \
  --value your-dockerhub-username

drone secret add \
  --repository your-github-username/drone-flask-demo \
  --name docker_password \
  --value your-dockerhub-password
```

✅ How to Run It All

Local Test

```
# (Optional) Create a virtual environment
python3 -m venv venv
source venv/bin/activate

# Install packages
pip install -r requirements.txt

# Run test
pytest tests/

# Build Docker image
```

```
docker build -t drone-flask-demo .

# Run the app
docker run -p 5000:5000 drone-flask-demo
```

🚀 Setup Drone CI (Quick View)

1. **Install Drone CI** (or use https://cloud.drone.io).

2. **Connect your GitHub repo** with .drone.yml file.

3. **Add secrets** for Docker Hub login.

4. **Push your code to GitHub** — Drone CI will trigger pipeline and build image.

🧩 Use Cases for This Drone CI Project

🔹 1. Automated Testing for Python Projects

What happens:
Every time you push code to GitHub, Drone CI automatically runs your Python test files.

Why it matters:

- Helps catch bugs early

- Keeps your project always in a tested state

- Builds confidence before deployment

🛠️ Relevant Step:

yaml

```
- name: run tests
  image: python:3.10
  commands:
    - pytest tests/
```

🔹 2. Docker Image Building & Pushing

What happens:
Drone CI builds your Docker image and pushes it to Docker Hub after passing tests.

Why it matters:

- You don’t need to manually build or push images
- Makes images easily usable for deployment (e.g., to Kubernetes or Docker Compose)

🛠️ Relevant Step:

yaml

```
- name: docker build and push
  image: plugins/docker
  settings:
    repo: your-dockerhub-username/drone-flask-demo
```

🔹 3. CI/CD for Web App Deployment

What happens:
You use this pipeline to continuously integrate and deploy your Flask app to a server or cloud.

Why it matters:

- Great starting point to understand how CI/CD works
- Can be extended to AWS EC2, GCP, or Kubernetes deployments

🔹 4. Learning YAML & Pipelines

What happens:
You write your CI/CD logic in .drone.yml.

Why it matters:

- Learn structure of YAML files
- Understand pipelines, steps, secrets, and plugins

🔹 5. Secret Management for Credentials

What happens:
You store Docker credentials in Drone Secrets, not in code.

Why it matters:

- Secure way to handle sensitive data

- Teaches best practices for DevOps & CI tools

🔹 6. Portfolio or Resume Project

What happens:
You host this on GitHub with a working .drone.yml.

Why it matters:

Shows hands-on CI/CD knowledge
Impresses recruiters with practical implementation
Great to talk about in interviews

🌱 How You Can Extend It

- Use **FastAPI** instead of Flask

- Add **code coverage** report using coverage.py

- Deploy image to **Kubernetes** or **Heroku**

- Trigger **Slack notifications** after each build

CircleCI (Open Source) – Open-source version of CircleCI.

CircleCI (Open Source) is the **self-hosted version** of the popular CircleCI platform. It helps you **automate the process of building, testing, and deploying**

your code—also known as CI/CD (Continuous Integration and Continuous Deployment).

You can connect CircleCI to your **GitHub or Bitbucket** repositories. Then, you define your workflow in a **YAML file**, and CircleCI automatically runs the steps you describe—like checking your code, running tests, or deploying your app.

The open-source version gives you the freedom to **run CircleCI on your own servers or cloud infrastructure**, giving you more **control and flexibility**. At the same time, you still get the core benefits of CircleCI—like automation, integrations, and customizable pipelines.

📌 Project Introduction

CircleCI (Open Source) is a self-hosted version of the popular CircleCI platform used for Continuous Integration and Continuous Deployment (CI/CD). It allows developers to automate code testing, building, and deployment workflows on their own infrastructure. With CircleCI, you can connect your GitHub repository, write a simple .circleci/config.yml file, and have your app automatically tested and deployed on every push.

In this project, we will:

- Create a simple Python app with a unit test.
- Connect it to GitHub.
- Set up CircleCI Open Source.
- Write a YAML config file for automated testing.
- Run a pipeline to demonstrate CI.

⚙️ Tech Stack

- Language: Python
- Version Control: GitHub
- CI/CD Tool: CircleCI Open Source
- Infrastructure: Self-hosted or Docker (for local testing)

📁 Project Structure

arduino

```
circleci-python-demo/
├── app/
│   └── hello.py
├── tests/
│   └── test_hello.py
├── .circleci/
│   └── config.yml
├── requirements.txt
└── README.md
```

🔨 Steps to Build the Project

1. Create a Simple Python App

hello.py:

python

```
def say_hello(name):
    return f"Hello, {name}!"
```

2. Add a Unit Test

test_hello.py:

python

```
from app.hello import say_hello


def test_say_hello():
    assert say_hello("Swapna") == "Hello, Swapna!"
```

3. Define Requirements

requirements.txt:

```
nginx
pytest
```

4. Set Up CircleCI Configuration

.circleci/config.yml:

yaml

```
version: 2.1
jobs:
  test:
    docker:
      - image: cimg/python:3.10
    steps:
      - checkout
      - run:
          name: Install dependencies
          command: pip install -r requirements.txt
      - run:
          name: Run tests
          command: pytest
workflows:
  version: 2
  test-and-build:
    jobs:
```

- test

🚀 How to Run with CircleCI Open Source

1. Fork or clone the project to your GitHub account.

2. Install CircleCI Open Source using Docker or on a VM.
   Docs: https://github.com/CircleCI-Public/server-terraform

3. Connect your GitHub repository.

4. Push your code. CircleCI will automatically trigger the workflow.

5. View the status and logs in the CircleCI dashboard.

✅ Use Cases of CircleCI (Open Source)

Automated Testing
Every code commit triggers automated unit tests, ensuring that your codebase remains stable and bug-free.

Continuous Integration (CI)
Merge code from multiple developers smoothly by automatically checking integration issues on every push or pull request.

Continuous Deployment (CD)
Automatically deploy applications to staging or production environments after successful testing—speeding up delivery cycles.

Self-Hosted CI/CD for Privacy and Security
Run CI/CD pipelines within your own infrastructure, ideal for organizations needing strict data privacy and regulatory compliance.

Custom Workflows
Define custom build and test workflows based on branches, tags, or environment conditions using the .circleci/config.yml file.

Multi-Language Support
Supports projects in various languages like Python, JavaScript, Go, and Java—ideal for polyglot environments.

Parallelism and Caching
Split your test suite to run in parallel or cache dependencies for faster builds and reduced execution time.

Version Matrix Testing
Run the same tests across multiple versions of a language or dependency to ensure cross-version compatibility.

Integration with GitHub/GitLab/Bitbucket
Easily connect your repo and trigger pipelines automatically with each commit or pull request.

Custom Deployments
Trigger custom scripts or deployment actions—whether deploying Docker images, pushing to a cloud provider, or sending notifications.

🔄 CircleCI (Open Source) Installation (Server Version)
Pre-requisites:

At least 4 CPUs, 8GB RAM

Docker and Docker Compose

Steps:

Download Docker Compose config from CircleCI:

git clone https://github.com/CircleCI-Public/server-terraform.git
cd server-terraform
Set environment variables and configurations. You can find an example .env file or follow their guide for setting up secrets.

Start services:
docker-compose up -d
Access CircleCI Web UI: Go to http://localhost:80 in your browser and complete the setup wizard.

Monitoring & Observability

Prometheus: Prometheus is an **open-source tool** used for **monitoring your applications and systems**. It helps you **track performance and health** over time by collecting and storing **metrics**—like CPU usage, memory, request counts, and more.

It's designed especially for modern, cloud-based environments and works really well with **Kubernetes**. Prometheus lets you write powerful queries to analyze the collected data, and it can even **send alerts** if something goes wrong—like if a server goes down or usage is too high.

It's a great tool to **understand what's happening inside your applications and infrastructure**, all in real-time.

🔹 Introduction

Prometheus is an open-source monitoring and alerting toolkit designed for reliability and scalability. It collects real-time metrics from targets like servers or containers, stores them in a time-series database, and lets you query and visualize the data.

🔹 Installation

On Linux (Ubuntu):

# Download Prometheus

```
wget
https://github.com/prometheus/prometheus/releases/latest/download/prometheus-*.linux-amd64.tar.gz
```

# Extract the files

```
tar xvf prometheus-*.linux-amd64.tar.gz

cd prometheus-*.linux-amd64
```

# Start Prometheus

```
./prometheus
```

Access Prometheus at: http://localhost:9090

For Docker:

```
docker run -p 9090:9090 prom/prometheus
```

🔹 Sample Project

Goal: Monitor a simple Node.js or Python app exposing metrics.

Step 1: Create an app that exposes metrics at /metrics using a Prometheus client library.

Step 2: Update prometheus.yml to scrape your app:

yaml

```
scrape_configs:
  - job_name: 'myapp'
    static_configs:
      - targets: ['localhost:8000']
```

Step 3: Run Prometheus and access the metrics dashboard at localhost:9090.

Prometheus Monitoring Project (Python + Docker + Prometheus)

📁 Project Folder Structure

```
prometheus-python-demo/
│
├── app/
│   ├── app.py              # Python app exposing metrics
│   └── requirements.txt    # Dependencies
│
├── prometheus/
│   └── prometheus.yml      # Prometheus config
│
```

```
├── docker-compose.yml      # Runs all services together
└── app/Dockerfile          # Dockerfile for Python app
```

📜 app/app.py (Python Flask App with Prometheus metrics)

python

```
from flask import Flask
from prometheus_client import Counter, generate_latest

# Create a Flask web application
app = Flask(__name__)

# Create a counter metric to count how many times our app was accessed
REQUEST_COUNT = Counter("app_requests_total", "Total number of requests")

# Root endpoint: shows a welcome message
@app.route("/")
def index():
    REQUEST_COUNT.inc()  # Increase request count by 1
    return "Hello, Prometheus!"  # Response shown to the user
```

```
# Metrics endpoint: Prometheus will scrape data from here
@app.route("/metrics")
def metrics():
    return generate_latest(), 200, {"Content-Type": "text/plain"}


# Start the app on port 8000, accessible from outside
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

📜 **app/requirements.txt**

```
nginx
flask              # For creating the web app
prometheus_client  # For exposing Prometheus metrics
```

📜 **app/Dockerfile (To containerize Python app)**

dockerfile

```
# Use a small Python image
FROM python:3.9-slim
```

```
# Set working directory
WORKDIR /app

# Copy all files into the container
COPY . .

# Install dependencies
RUN pip install -r requirements.txt

# Run the app
CMD ["python", "app.py"]
```

📜 **prometheus/prometheus.yml (Prometheus config)**

yaml

```
global:
  scrape_interval: 5s  # How often to collect data (every 5 seconds)

scrape_configs:
  - job_name: 'python_app'  # Name of the job shown in Prometheus UI
    static_configs:
      - targets: ['python-app:8000']  # Where Prometheus will look for metrics
```

📜 docker-compose.yml (Runs Python app + Prometheus together)

```yaml
version: '3'


services:
  python-app:
    build: ./app  # Build image using app/Dockerfile
    ports:
      - "8000:8000"  # Expose port for our app


  prometheus:
    image: prom/prometheus  # Official Prometheus image
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
    ports:
      - "9090:9090"  # Expose Prometheus UI on port 9090
```

🚀 How to Run the Project

Open a terminal and run:

```
# Go to your project folder


cd prometheus-python-demo


# Start everything with Docker

docker-compose up --build
```

🔍 How to Check Output

- **Open browser:**
  - App: http://localhost:8000
  - Metrics: http://localhost:8000/metrics
  - Prometheus: http://localhost:9090

In Prometheus, search this metric:

nginx

app_requests_total

✅ What You Learned

How to expose metrics from your app
How Prometheus collects data

How to run everything easily using Docker

Explanation:

REQUEST_COUNT = Counter("app_requests_total", "Total number of requests")
→ Creates a Prometheus **Counter** metric.

Counter
→ Only increases. Used to count things like requests, errors, etc.

Why here?
→ To count how many times / is accessed.
Each visit runs REQUEST_COUNT.inc() to add +1.

Example:
Like a shop counter—each visitor adds one. Prometheus collects this to show traffic over time.

Prometheus Use Cases

Application Monitoring
Track request count, response time, error rates, etc., for web apps and APIs.

Infrastructure Monitoring
Monitor CPU, memory, disk usage, and network stats of servers, containers, or VMs.

Kubernetes Monitoring
Collect metrics from nodes, pods, and containers using exporters like kube-state-metrics.

Database Monitoring
Check database performance—queries per second, connections, replication lag.

Alerting System
Send alerts (via Alertmanager) when something goes wrong—like high CPU or server down.

Service Uptime Tracking
Monitor uptime of services using blackbox exporter (e.g., ping, HTTP checks).

CI/CD Monitoring
Track build times, test results, and deployment status in pipelines.

IoT or Custom Devices
Collect metrics from smart devices or sensors using custom exporters.

Grafana: Grafana is an **open-source tool** that lets you **see and interact with your monitoring data** using beautiful, easy-to-understand **dashboards**. It works with Prometheus and many other data sources to help you **visualize key metrics**—like server health, application performance, or website traffic.

Instead of just looking at raw numbers, Grafana turns your data into **charts, graphs, and alerts**, so you can quickly spot problems and understand trends. It's also **highly customizable**, so you can build dashboards that show exactly what matters to you.

Grafana makes it easier to **monitor systems in real-time** and take action when something goes wrong.

Install Grafana on a Linux system (Ubuntu/Debian-based):

✅ Grafana Installation Steps (Ubuntu/Debian)

Step 1: Update System

```
sudo apt update

sudo apt upgrade -y
```

Step 2: Install Dependencies

```
sudo apt install -y software-properties-common
```

Step 3: Add Grafana APT Repository

```
sudo add-apt-repository "deb https://packages.grafana.com/oss/deb stable main"
```

Step 4: Add GPG Key

```
sudo wget -q -O /usr/share/keyrings/grafana.key https://packages.grafana.com/gpg.key
```

Step 5: Update and Install Grafana

```
sudo apt update

sudo apt install grafana -y
```

Step 6: Enable and Start Grafana Service

```
sudo systemctl enable grafana-server

sudo systemctl start grafana-server
```

Step 7: Check Status

```
sudo systemctl status grafana-server
```

Step 8: Access Grafana Web UI

Open your browser and go to:
📍 http://<your-server-ip>:3000
(Default port is 3000)

- Default **Username**: admin
- Default **Password**: admin (you’ll be prompted to change it)

📌 Optional: Open Firewall (if enabled)

```
sudo ufw allow 3000/tcp
```

Project: Monitor Your System with Grafana and Prometheus

🎯 Goal

Visualize your **CPU**, **memory**, and **disk usage** in real-time using **Prometheus** as the data source and **Grafana** as the visualization tool.

🔧 Tools Used

- **Prometheus** – collects and stores metrics
- **Node Exporter** – exposes Linux system metrics to Prometheus

- **Grafana** – displays metrics in dashboards

📋 Project Setup Steps

✅ Step 1: Install Prometheus

# Download Prometheus

```
wget https://github.com/prometheus/prometheus/releases/latest/download/prometheus-2.48.1.linux-amd64.tar.gz
```

# Extract and move

```
tar -xvzf prometheus-2.48.1.linux-amd64.tar.gz

cd prometheus-2.48.1.linux-amd64
```

✅ Step 2: Install Node Exporter

# Download Node Exporter

```
wget https://github.com/prometheus/node_exporter/releases/latest/download/node_exporter-1.8.0.linux-amd64.tar.gz
```

# Extract and move

```
tar -xvzf node_exporter-1.8.0.linux-amd64.tar.gz

cd node_exporter-1.8.0.linux-amd64
```

Run node exporter in a terminal:

```
./node_exporter
```

✅ Step 3: Configure Prometheus

Edit prometheus.yml:

yaml

```
scrape_configs:
  - job_name: 'node_exporter'
    static_configs:
      - targets: ['localhost:9100']
```

Then start Prometheus:

```
./prometheus --config.file=prometheus.yml
```

✅ Step 4: Install Grafana (if not done already)

Refer to earlier steps or run:

```
sudo apt install grafana -y
sudo systemctl enable grafana-server
sudo systemctl start grafana-server
```

✅ Step 5: Connect Prometheus to Grafana

- Go to: http://localhost:3000
- Login: admin / admin
- Add Data Source → Choose **Prometheus**
- URL: http://localhost:9090

✅ Step 6: Import Dashboard

- Click "+" → Import
- Use Dashboard ID: 1860 (for Node Exporter Full Dashboard)
- Select Prometheus as data source

📊 What You'll See

- CPU Usage over time
- Memory Utilization
- Disk I/O stats
- Network stats
- System load average

💡 Learning Outcomes

- Understand how Prometheus scrapes metrics
- Learn how Grafana visualizes metrics with charts and alerts
- Explore how dashboards can be customized
- Understand basic system metrics

📌 Use Cases of Grafana

🌐 1. Server & Infrastructure Monitoring

Use: Monitor CPU, memory, disk, and network usage of servers.
Data Source: Prometheus, Node Exporter
Example: A DevOps team uses Grafana to check if a server is under heavy load or if it's running out of memory.

📦 2. Application Performance Monitoring (APM)

Use: Track API response times, error rates, or request counts.
Data Source: Prometheus, Jaeger, Tempo, Loki
Example: A backend developer uses Grafana to find slow endpoints in their application.

🌩️ 3. Cloud Monitoring (AWS/GCP/Azure)

Use: Visualize cloud resource usage like EC2 instances, S3 buckets, billing, etc.
Data Source: AWS CloudWatch, GCP Stackdriver, Azure Monitor

Example: A cloud engineer monitors how much bandwidth is being used by an EC2 instance.

⚠️ 4. Alerting and Incident Response

Use: Set alerts when something goes wrong, e.g., CPU > 80% for 5 minutes.
Data Source: Prometheus + Grafana Alerting
Example: DevOps gets a Slack alert if disk space is critically low.

📊 5. Business & Analytics Dashboards

Use: Visualize user signups, sales, traffic, etc. from SQL databases.
Data Source: MySQL, PostgreSQL, Google Sheets
Example: A product team tracks daily active users and weekly signups on a Grafana dashboard.

📉 6. IoT and Sensor Data Visualization

Use: Monitor temperature, humidity, or other real-time sensor data.
Data Source: InfluxDB, MQTT
Example: A smart agriculture project tracks real-time temperature across fields using Grafana.

🧪 7. DevSecOps Monitoring

Use: Track security metrics, vulnerability scans, or compliance checks.
Data Source: Prometheus + tools like Trivy or SonarQube
Example: A DevSecOps engineer monitors the number of vulnerabilities detected during a CI/CD pipeline run.

🛰️ 8. Kubernetes Cluster Monitoring

Use: Observe pod CPU/memory usage, node health, service status.
Data Source: Prometheus + kube-state-metrics + Grafana
Example: A team monitors how many pods are running and if any nodes are NotReady.

🎮 9. Gaming or App Analytics

Use: Track concurrent users, matchmaking times, in-game events.
Data Source: PostgreSQL, Prometheus
Example: A game developer tracks how many players are online every hour.

🧠 10. Personal Projects / Home Labs

Use: Monitor home network, Raspberry Pi, weather station, etc.
Data Source: Node Exporter, InfluxDB
Example: A hobbyist visualizes power consumption at home using Grafana dashboards.

Loki – Log aggregation system by Grafana Labs.

Loki is an open-source **log aggregation system** created by **Grafana Labs**. It is used to **collect, store, and query logs** from different sources, especially in cloud-native environments like **Kubernetes**.

Loki is designed to work **just like Prometheus**, but for logs. Instead of indexing the full content of logs (which can be heavy and slow), Loki only indexes **labels** (like job, instance, container, etc.), making it **faster and cheaper**.

✅ Key Features:

- Integrates smoothly with **Grafana** for visualizing logs
- Works well with **Promtail**, **Fluentd**, or **Fluent Bit** as log shippers
- Especially useful in **Kubernetes** for managing logs from pods and containers
- Can correlate **logs with metrics** from Prometheus, giving a deeper view of your system

---

🔹 Key Features

- Seamless **Grafana integration** for log visualization
- Works with log shippers like **Promtail**, **Fluentd**, and **Fluent Bit**
- Designed for **Kubernetes**, efficiently collects pod/container logs
- Correlates logs with **Prometheus metrics** for deep observability

---

🔹 Installation (Using Docker Compose)

```
# Create a docker-compose.yml file with Loki, Promtail, and Grafana
mkdir loki-stack && cd loki-stack
nano docker-compose.yml
```

Paste this:

```
version: "3"

services:
  loki:
    image: grafana/loki:2.9.1
    ports:
      - "3100:3100"
    command: -config.file=/etc/loki/local-config.yaml

  promtail:
    image: grafana/promtail:2.9.1
    volumes:
      - /var/log:/var/log
      - /etc/promtail:/etc/promtail
    command: -config.file=/etc/promtail/promtail.yaml

  grafana:
    image: grafana/grafana:latest
    ports:
      - "3000:3000"
```

Create a basic Promtail config:

```
mkdir -p /etc/promtail

nano /etc/promtail/promtail.yaml
```

Paste this:

yaml

```
server:
  http_listen_port: 9080
  grpc_listen_port: 0

positions:
  filename: /tmp/positions.yaml

clients:
  - url: http://loki:3100/loki/api/v1/push

scrape_configs:
  - job_name: system
    static_configs:
      - targets:
```

```
      - localhost
    labels:
      job: varlogs
      __path__: /var/log/*.log
```

Then run:

```
docker-compose up -d
```

🚀 Loki Log Monitoring Stack (Beginner Project)

🔹 Introduction

Loki is an open-source log aggregation system created by **Grafana Labs**. It's used to **collect, store, and query logs**, especially in **cloud-native setups like Kubernetes**. Unlike traditional log systems, Loki only indexes metadata (labels), making it fast and cost-effective.

🔹 Key Features

- Works seamlessly with **Grafana** for visualizing logs
- Supports log shippers like **Promtail**, **Fluentd**, and **Fluent Bit**
- Great for **Kubernetes**: collects logs from containers & pods
- Pairs with **Prometheus** for metrics + log correlation

📁 Project: Loki + Promtail + Grafana using Docker Compose

✅ Goal

Create a full logging stack that:

- Collects logs from local files (like /var/log/*.log)
- Sends them to Loki via Promtail
- Visualizes logs in Grafana

📁 Project Structure

```
loki-log-monitoring/
├── docker-compose.yml
├── promtail/
│   └── promtail.yaml
└── README.md
```

⚙️ Setup Instructions

Step 1: Create Project Folder

```
mkdir -p loki-log-monitoring/promtail
cd loki-log-monitoring
```

Step 2: Create docker-compose.yml

```
version: "3"

services:

  loki:

    image: grafana/loki:2.9.1

    container_name: loki

    ports:

      - "3100:3100"

    command: -config.file=/etc/loki/local-config.yaml


  promtail:

    image: grafana/promtail:2.9.1

    container_name: promtail

    volumes:

      - /var/log:/var/log

      - ./promtail/promtail.yaml:/etc/promtail/promtail.yaml

    command: -config.file=/etc/promtail/promtail.yaml


  grafana:

    image: grafana/grafana:latest

    container_name: grafana
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=admin
```

Step 3: Create Promtail Config promtail/promtail.yaml

yaml

```
server:
  http_listen_port: 9080
  grpc_listen_port: 0

positions:
  filename: /tmp/positions.yaml

clients:
  - url: http://loki:3100/loki/api/v1/push

scrape_configs:
  - job_name: system
    static_configs:
      - targets:
          - localhost
        labels:
          job: varlogs
          __path__: /var/log/*.log
```

Step 4: Run the Stack

```
docker-compose up -d
```

Step 5: Access Grafana

- URL: http://localhost:3000
- Login: admin / admin

Step 6: Add Loki as Data Source in Grafana

1. Go to “Settings” → “Data Sources” → “Add data source”
2. Choose **Loki**
3. Set URL to: http://loki:3100

4. Click **Save & Test**

Step 7: Explore Logs

Go to **Explore** tab in Grafana and run:

logql

```
{job="varlogs"}
```

You’ll see real-time logs from /var/log/*.log.

📘 README.md (Optional)

```
# Loki Log Monitoring Project

This project sets up a log monitoring system using Loki, Promtail, and Grafana with Docker Compose.

## 🔧 Components

- **Loki**: Log database

- **Promtail**: Log shipper that reads /var/log/*.log

- **Grafana**: Visualization UI

## 🚀 Run the Stack
```

```
docker-compose up -d
```

🌐 Access

- Grafana: http://localhost:3000
  Login: admin / admin

📊 Sample Log Query

logql

```
{job="varlogs"}
```

yaml

## 💡 Use Cases

- Monitor logs from **local machine or Kubernetes pods**

- **Debug applications** by searching and filtering logs

- Build a centralized logging stack for **DevOps projects**

- Combine logs with **Prometheus metrics** for advanced observability

---

🔹 Use Cases for Loki

1. Kubernetes Log Aggregation

- **Scenario**: In a Kubernetes environment, multiple pods and containers generate logs.

- **Solution**: Use **Promtail** as a log shipper to collect logs from all containers, and store them in **Loki**. With **Grafana**, you can then query and visualize logs to monitor application behavior and troubleshoot issues across different containers and nodes.

- **Benefit**: Easy access to logs from all Kubernetes pods in a single place, making it easier to correlate events and troubleshoot issues.

---

2. Real-time Log Monitoring

- **Scenario**: A system administrator wants to monitor system logs in real-time.

- **Solution**: Set up **Loki** to collect logs from system files (e.g., /var/log/*.log) using **Promtail**, then use **Grafana** for real-time log visualization and query logs with specific labels (e.g., job="varlogs").

- **Benefit**: Instant visibility into the system's health, errors, and warnings as they happen.

---

3. Log Correlation with Metrics

- **Scenario**: You have **Prometheus** collecting system metrics (CPU, memory usage, etc.) and **Loki** collecting logs.
- **Solution**: Integrate **Loki** with **Prometheus** in **Grafana**. By correlating logs and metrics, you can easily diagnose issues like a spike in CPU usage that may correlate with a specific log event.
- **Benefit**: A deeper, more comprehensive view of your system's behavior and performance. You can link performance metrics to specific log events for faster diagnosis.

---

4. Centralized Logging for Microservices

- **Scenario**: A microservices-based application generates logs in multiple services across different environments (dev, staging, production).
- **Solution**: Use **Loki** to collect logs from all microservices and store them in a central location. **Grafana** can be used to create a unified dashboard where you can query and visualize logs from all services, making it easier to troubleshoot issues across multiple services.
- **Benefit**: One place to access logs from all microservices, helping DevOps teams quickly identify and resolve problems, even when they span multiple services.

---

5. Security Monitoring and Incident Detection

- **Scenario**: You want to monitor logs for suspicious activity like unauthorized access attempts or errors.

- **Solution**: Set up **Loki** to collect logs from security devices, firewalls, and application logs. Use **Grafana** to create dashboards that alert you to specific security events, such as multiple failed login attempts or unusual patterns in system activity.

- **Benefit**: Proactively detect security incidents and respond faster by correlating logs with potential security threats.

---

6. Log Storage for Compliance and Auditing

- **Scenario**: Your organization must retain logs for auditing and compliance purposes, such as meeting **GDPR** or **HIPAA** requirements.

- **Solution**: Store logs in **Loki** and use **Grafana** to maintain visibility. By indexing logs by labels like job, user, or service, you can efficiently query and analyze logs over long periods.

- **Benefit**: Secure and compliant log storage, combined with powerful querying capabilities for audits and regulatory checks.

---

7. Troubleshooting Production Issues

- **Scenario**: A production application is failing, but the logs are scattered across various systems and containers.

- **Solution**: Use **Loki** to aggregate logs from all containers, nodes, and systems involved in the application. With **Grafana**, you can quickly search for specific error patterns, correlate logs with application states, and diagnose issues faster.

- **Benefit**: Faster troubleshooting with access to centralized, easily searchable logs, reducing downtime and improving incident resolution times.

---

8. Distributed Tracing and Log Aggregation for Microservices

- **Scenario**: You're building a distributed system where services communicate with each other. Each service generates logs with traces that can be used to track a request across multiple services.

- **Solution**: Use **Loki** to collect and aggregate logs from each service, then correlate logs with **distributed tracing** information (using tools like **Jaeger** or **Zipkin**) to trace requests as they pass through different services.

- **Benefit**: Easier identification of bottlenecks or issues across services in a distributed environment, improving system reliability.

---

9. Cost-effective Log Management

- **Scenario**: You need a log aggregation solution that's cheaper than traditional systems like Elasticsearch.

- **Solution**: **Loki** is designed to be more cost-effective because it indexes **labels** instead of the full log content. This reduces storage and computational requirements.

- **Benefit**: A budget-friendly log aggregation solution for teams that need to manage large volumes of logs without breaking the bank.

---

10. Custom Dashboards for Business Insights

- **Scenario**: A business team needs to track system metrics that impact user experience, like response times, errors, or specific user actions.
- **Solution**: Use **Loki** and **Grafana** to create custom dashboards that pull in logs related to specific business KPIs, such as user sign-ups or purchase attempts, and visualize that data in real-time.
- **Benefit**: Deliver actionable insights for business teams by integrating system logs with business metrics, making it easier to track business performance.

Conclusion

Loki is a powerful and versatile log aggregation system that is well-suited for use cases across development, operations, security, and business intelligence. Its integration with **Grafana** and ability to work seamlessly with cloud-native environments like **Kubernetes** make it an essential tool for modern log management.

Jaeger – Distributed tracing, useful for microservices debugging.

Jaeger is an **open-source distributed tracing system** developed by **Uber Technologies** and now part of the **CNCF (Cloud Native Computing Foundation)**.

In a **microservices** architecture, a single user request may travel through multiple services. Jaeger helps you **track that request across all the services**, showing how long each step takes and where issues might occur.

✅ Why use Jaeger?

- Helps **debug and troubleshoot** performance issues

- Visualizes **latency** and **bottlenecks**
- Shows **trace details** – how requests flow through services
- Works well with tools like **Prometheus**, **Grafana**, and **OpenTelemetry**

🔍 Think of it like Google Maps for your microservices – it shows the full **journey** of each request!

⚙️ How to Install Jaeger (Quick & Easy)

🐳 Option 1: Run Jaeger using Docker All-in-One

This setup runs all Jaeger components in a single container. Ideal for **development** or **learning** purposes.

```
docker run -d --name jaeger \
  -e COLLECTOR_ZIPKIN_HOST_PORT=:9411 \
  -p 5775:5775/udp \
  -p 6831:6831/udp \
  -p 6832:6832/udp \
  -p 5778:5778 \
  -p 16686:16686 \
  -p 14268:14268 \
  -p 14250:14250 \
  -p 9411:9411 \
  jaegertracing/all-in-one:latest
```

🔗 Access the Jaeger UI at: **http://localhost:16686**

📦 Option 2: Install Jaeger using Docker Compose

You can use the official Jaeger Compose file:

```
curl -O https://raw.githubusercontent.com/jaegertracing/jaeger/master/docker-compose/docker-compose-all-in-one.yaml
docker-compose -f docker-compose-all-in-one.yaml up -d
```

Again, access the UI at: **http://localhost:16686**

☸️ Option 3: Install Jaeger in Kubernetes (via Helm)

If you're working with Kubernetes:

```
helm repo add jaegertracing https://jaegertracing.github.io/helm-charts
helm repo update
helm install jaeger jaegertracing/jaeger
```

This installs Jaeger in your cluster. Use kubectl get pods and kubectl port-forward to access the UI.

Project: Implement Jaeger Distributed Tracing in a Microservices Architecture

Objective:

The goal of this project is to demonstrate how Jaeger can help trace a request across multiple microservices and visualize performance bottlenecks or issues in a distributed system. This project will simulate a simple microservices-based application and integrate Jaeger for tracing.

Tech Stack:

- **Jaeger** for distributed tracing.
- **Docker** and **Docker Compose** to containerize the services.
- **Spring Boot** (or Flask) for building simple microservices.
- **Prometheus** and **Grafana** for metrics and visualization.
- **OpenTelemetry** for instrumenting the services.

Steps:

1. **Set up Microservices:**
   - Create 3-4 simple microservices, e.g., User Service, Order Service, Payment Service.
   - Each microservice will expose simple REST APIs (e.g., POST /createUser, GET /order/{id}, POST /processPayment).
2. **Integrate Jaeger with Microservices:**
   - Add Jaeger client libraries to each microservice (you can use OpenTelemetry to instrument the services).
   - Set up the Jaeger agent as a container or as a service to capture traces.
   - Ensure that each service sends trace data to the Jaeger agent for the requests it handles.
3. **Simulate Requests:**
   - Use tools like Postman or CURL to simulate user requests that flow across multiple services (e.g., createUser in User Service ->

createOrder in Order Service -> processPayment in Payment Service).

4. **Visualize Traces in Jaeger UI:**

   - After sending the requests, visit Jaeger's UI to visualize the traces and see the flow of requests across the microservices.

   - Identify latency, bottlenecks, or errors in the trace.

5. **Monitor with Prometheus & Grafana:**

   - Integrate Prometheus to collect metrics from the microservices (like response times, error rates, etc.).

   - Set up Grafana dashboards to visualize the metrics and correlate them with Jaeger's traces to understand system performance.

6. **Troubleshooting:**

   - Intentionally introduce performance issues or errors (e.g., delays or failures in the Payment Service) to see how Jaeger and Grafana help identify problems.

   - Use Jaeger's trace data to debug where the request is getting delayed or where errors are happening.

Expected Outcomes:

- Learn how Jaeger helps in tracing requests across microservices and visualizing performance bottlenecks.

- Understand how to use Jaeger, Prometheus, and Grafana to monitor and troubleshoot distributed systems.

- Gain insight into how Jaeger integrates with other observability tools (like OpenTelemetry) for end-to-end tracing.

Further Exploration:

- Use **Jaeger in Kubernetes** to handle scaling and distributed tracing in a cloud-native environment.
- Explore **Jaeger's Sampling Mechanism** to control the amount of trace data collected in a production environment.
- Integrate **Jaeger with other CNCF projects**, such as Istio for service mesh, to get even more powerful observability and tracing capabilities.

This project will give you hands-on experience with Jaeger and distributed tracing in a microservices environment, and it's a great starting point for anyone learning about observability in modern cloud-native applications.

---

1. Performance Optimization and Bottleneck Identification

Scenario:

A large e-commerce platform has multiple microservices handling different functionalities (user management, product catalog, order processing, payment, etc.). When a customer makes a purchase, their request travels through several services.

Use Case:

Jaeger helps to trace the request from the front-end to the back-end services. It visualizes the latency across each microservice, allowing the team to pinpoint which service is causing delays. For example, if the order processing service takes much longer than expected, the team can focus their optimization efforts on that service.

Outcome:

- Identify slow-performing services.
- Improve response time by optimizing bottlenecks.
- Provide detailed insight into service latencies to make informed performance optimizations.

---

2. Root Cause Analysis in Production Issues

Scenario:

After deploying a new feature, customers start reporting errors with orders not being processed correctly, but logs are inconclusive. The system is complex, with services spread across multiple regions, making troubleshooting difficult.

Use Case:

With Jaeger, you can trace a failing request from start to finish. By examining the trace data, you identify that an error occurs in the payment-service, specifically during a communication delay between services. You can also see if there are network issues or timeouts, helping isolate the root cause.

Outcome:

- Quickly identify the source of production issues.
- Use trace data to fix bugs or misconfigurations that affect service communication.

- Reduce mean time to resolution (MTTR) for production incidents.

---

3. Monitoring Distributed Transactions in Complex Workflows

Scenario:

A cloud-based banking application involves several steps when processing a loan application, including background checks, loan eligibility, approval, and disbursement. This involves interacting with multiple services, like background-check-service, loan-eligibility-service, and loan-approval-service.

Use Case:

Jaeger can be used to monitor the entire loan approval workflow, which involves different services and databases. Traces help visualize the path of each loan application, so the team can see how long each stage takes and if there are delays in any of the steps. It's especially useful for complex workflows where multiple services are involved, making it easier to understand how each service contributes to the overall process.

Outcome:

- Track long-running transactions that span across multiple services.
- Gain visibility into how services interact during complex workflows.
- Spot latency or error hotspots in multi-step processes.

---

4. Service Dependency Mapping

Scenario:

As a new service (user-profile-service) is added to a microservices ecosystem, the team needs to understand how it interacts with other existing services like user-service, order-service, and payment-service.

Use Case:

Jaeger provides a comprehensive view of all the requests that flow through the user-profile-service. By analyzing Jaeger's trace visualization, the team can quickly map how user-profile-service interacts with other services and identify potential impacts of changes in the new service. This mapping helps understand the service dependencies and their impact on the system.

Outcome:

- Map and visualize service dependencies.
- Understand the interaction between newly added services and existing services.
- Ensure smooth communication and prevent issues from new service integrations.

---

5. Error Tracking and Fault Isolation

Scenario:

An API in your microservices ecosystem returns HTTP 500 errors, but the logs do not provide enough context to understand why. The service call flows through multiple microservices, and pinpointing the failure in such a distributed setup is difficult.

Use Case:

With Jaeger, you can track the traces for failed requests and observe the trace path that includes every microservice call. If one service fails, you can immediately identify the exact service and endpoint where the failure occurred. This makes it easier to isolate the fault and fix it without needing to dive deep into logs from each service individually.

Outcome:

- Quickly identify and isolate errors.
- Get a clear view of where the fault occurred in the distributed system.
- Improve fault detection and response times.

---

6. Distributed Tracing in Cloud-Native Environments (Kubernetes)

Scenario:

A cloud-native company is using Kubernetes for orchestrating their microservices. They want visibility into their containerized applications and the network traffic between services to ensure the system operates optimally.

Use Case:

Jaeger can be integrated with Kubernetes to collect tracing data for services running within pods. By deploying Jaeger agents as sidecars in your Kubernetes pods, you get distributed tracing data from services deployed across multiple containers. This gives the team insight into latency, errors, and bottlenecks specific to the containerized environment.

Outcome:

- Monitor the performance of microservices running inside Kubernetes clusters.
- Understand service interactions and optimize container-based deployments.

- Simplify troubleshooting in cloud-native environments.

---

7. A/B Testing and Feature Monitoring

Scenario:

A company is testing a new feature (e.g., a new checkout process) with a subset of users while the rest continue to use the old flow. This test involves different service interactions and responses, and the team wants to monitor the impact of the new feature on overall performance and latency.

Use Case:

Jaeger can help compare traces between users using the old and new checkout processes. By monitoring the latency, request flow, and errors in both traces, the team can measure the performance impact of the new feature and determine whether any additional optimization is needed.

Outcome:

- Measure the impact of A/B tests on service performance.
- Monitor the new feature’s impact on latency and errors.
- Make data-driven decisions based on performance insights.

---

8. Cross-Team Collaboration and Debugging

Scenario:

Multiple teams manage different microservices in a large organization. When a bug is reported, it's hard for one team to know exactly where the issue lies in the whole service flow, especially if it's affecting cross-team services.

Use Case:

Jaeger enables cross-team collaboration by providing a shared trace view. When a bug is reported, the team can quickly share trace data with other teams to investigate how their service interacts with others. This improves troubleshooting and speeds up resolution by giving a holistic view of the entire system's behavior.

Outcome:

- Facilitate faster issue resolution across teams.
- Improve communication and collaboration between service teams.
- Enable a more efficient debugging process in complex, multi-team environments.

These **use cases** show how Jaeger plays a crucial role in monitoring and debugging distributed systems, especially in a microservices-based architecture. It provides visibility into the system's behavior, helps optimize performance, and improves fault detection and resolution.

Thanos – Open-source tool that extends Prometheus for high availability and scalable long-term storage.

🏗️ What is Thanos? – Introduction for Beginners

Thanos is an **open-source project** that extends **Prometheus** to make it more **scalable, highly available, and long-term storage-ready**.

While **Prometheus** is amazing for collecting and querying metrics, it has some **limitations**:

- It **stores data locally**, which can be lost if the server crashes
- It doesn't support **clustering** or **high availability** natively
- Long-term storage and **global querying** across multiple Prometheus servers isn't built-in

Thanos solves all of this by:

- **Storing Prometheus data** in cloud storage like S3 or GCS (long-term)
- Allowing **global queries** across multiple Prometheus instances
- Enabling **HA (High Availability)** setups with deduplication of data
- Using **sidecar**, **store**, **querier**, and **compactor** components to build a full system

🎯 **Use Case Example**: If you're running Prometheus in multiple Kubernetes clusters, **Thanos lets you query all of them together** from a single Grafana dashboard — and keeps the data safe for years.

To install **Thanos**, the open-source tool that extends Prometheus for high availability and scalable long-term storage, follow the steps below:

Prerequisites

- Prometheus is already installed and running in your environment.

- Kubernetes clusters, or a similar environment, where Prometheus is set up (optional, but useful for large-scale deployments).

- Cloud storage (e.g., S3, GCS, etc.) for long-term storage.

Step-by-Step Installation Guide

1. **Download Thanos**

    - You can download the latest release of Thanos from GitHub Releases.

For example, use the following command to download the latest version (replace latest with the actual version if needed):

```
wget https://github.com/thanos-io/thanos/releases/download/v0.25.0/thanos-0.25.0-linux-amd64.tar.gz
tar -xvzf thanos-0.25.0-linux-amd64.tar.gz
cd thanos-0.25.0-linux-amd64
```

2. **Thanos Components** Thanos consists of multiple components to extend Prometheus functionality:

    - **Sidecar**: Helps in uploading Prometheus data to cloud storage.

    - **Store**: Reads data from cloud storage for long-term storage.

    - **Querier**: Aggregates queries across multiple Prometheus instances.

    - **Compactor**: Compacts data stored in the cloud.

3. **Install Thanos in Kubernetes**

   If you're using Kubernetes, you can deploy Thanos using Helm or manually with Kubernetes manifests. For Helm:

Install Thanos using Helm Add the Thanos repository:

```
helm repo add thanos https://thanos-io.github.io/thanos
helm repo update
```

Install Thanos using Helm:

```
helm install thanos thanos/thanos
```

Manually Using Kubernetes Manifests You can also use Thanos manifests to deploy each component (sidecar, store, querier, compactor). A basic example for the sidecar component would look like this:

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: thanos-sidecar
spec:
  replicas: 1
  selector:
    matchLabels:
      app: thanos-sidecar
  template:
    metadata:
      labels:
        app: thanos-sidecar
    spec:
      containers:
      - name: thanos
```

```
        image: thanosio/thanos:v0.25.0
        args:
          - sidecar
          - --http-address=0.0.0.0:10902
          - --grpc-address=0.0.0.0:10901
          - --prometheus.url=http://<prometheus-server>:9090
          - --objstore.config-file=/etc/thanos/bucket.yaml
        volumeMounts:
          - mountPath: /etc/thanos
            name: bucket-config
      volumes:
      - name: bucket-config
        configMap:
          name: thanos-bucket-config
```

4. **Configure Long-Term Storage (S3, GCS, etc.)**

   - Thanos relies on external storage (such as AWS S3 or Google Cloud Storage) for long-term storage.

5. **Create a config file (e.g., bucket.yaml) for your cloud provider:**

S3 Example:

yaml

```
type: S3
config:
  bucket: <your-bucket-name>
  endpoint: s3.amazonaws.com
  access_key: <AWS_ACCESS_KEY>
  secret_key: <AWS_SECRET_KEY>
  secure: true
```

GCS Example:

yaml

```
type: GCS
config:
  bucket: <your-bucket-name>
  service_account: <your-service-account-json>
```

6. **Start Thanos Components**

   - **Sidecar**:

     - Prometheus with the Thanos sidecar component will automatically upload data to the long-term storage (e.g., S3, GCS).

   - **Store**: The Thanos store component fetches the data stored in cloud storage and makes it available for querying.

Querier: Start the Thanos querier component to allow global queries across Prometheus instances.

Example to start querier:

```
thanos query --http-address=0.0.0.0:9090 --grpc-address=0.0.0.0:9091
--query-frontend.enabled
```

7. **Integrate with Grafana for Querying**

   To visualize Thanos data on Grafana, add Thanos as a data source:

   - URL: http://<thanos-querier-service>:9090
   - Choose Prometheus as the data source type.

8. **Configure High Availability**

   With Thanos, you can deploy multiple Prometheus instances in a high-availability configuration. Thanos will automatically deduplicate data from multiple Prometheus instances.

9. **Monitor and Optimize**

   - Once Thanos is running, monitor the components via Grafana or Prometheus.
   - Optimize configurations based on your infrastructure and storage needs.

Project: Multi-Cluster Monitoring with Prometheus, Thanos, and Grafana

Project Overview:

In this project, you'll extend Prometheus monitoring across multiple Kubernetes clusters by setting up Thanos. You'll also integrate Grafana for visualizing metrics from all clusters in a centralized dashboard. The project will involve configuring Thanos components like Sidecar, Store, Querier, and Compactor, as well as integrating long-term storage (e.g., S3 or GCS) for metrics retention.

Project Components:

1. **Prometheus**: Collects metrics from multiple Kubernetes clusters.

2. **Thanos**: Extends Prometheus to provide high availability, scalability, and long-term storage.

   - **Sidecar**: For uploading Prometheus data to long-term storage.
   - **Store**: To access the data stored in long-term storage (e.g., S3, GCS).
   - **Querier**: Aggregates queries across all Prometheus instances.
   - **Compactor**: Compacts the data in long-term storage for efficiency.

3. **Grafana**: For querying and visualizing metrics from Prometheus and Thanos.

4. **Kubernetes**: The platform running multiple clusters of Prometheus instances.

Steps to Implement the Project:

1. Set Up Multiple Kubernetes Clusters

- Set up multiple Kubernetes clusters (you can use **kind** for local clusters or **GKE/EKS/AKS** for cloud-based clusters).

- In each cluster, install **Prometheus** to collect metrics from the applications running in the clusters.

2. Install Thanos Sidecar with Prometheus

- The Thanos Sidecar will be installed alongside Prometheus in each cluster. It will upload Prometheus metrics to cloud storage and enable Thanos features like high availability and long-term storage.

- Create a Kubernetes Deployment for the **Thanos Sidecar** and configure it to use cloud storage (S3, GCS, etc.).

Example thanos-sidecar.yaml:

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: thanos-sidecar
spec:
  replicas: 1
  selector:
    matchLabels:
      app: thanos-sidecar
  template:
    metadata:
      labels:
        app: thanos-sidecar
    spec:
      containers:
      - name: thanos
        image: thanosio/thanos:v0.25.0
        args:
          - sidecar
          - --http-address=0.0.0.0:10902
          - --grpc-address=0.0.0.0:10901
          - --prometheus.url=http://<prometheus-server>:9090
          - --objstore.config-file=/etc/thanos/bucket.yaml
        volumeMounts:
          - mountPath: /etc/thanos
            name: bucket-config
      volumes:
        - name: bucket-config
          configMap:
            name: thanos-bucket-config
```

3. Set Up Cloud Storage (S3/GCS)

- Choose cloud storage (e.g., S3 or GCS) to store your long-term metrics.
- Create a bucket on your preferred cloud provider and configure Thanos to upload data to this bucket.

Example of the bucket.yaml configuration file for S3:

yaml

```
type: S3
config:
  bucket: <your-bucket-name>
  endpoint: s3.amazonaws.com
  access_key: <AWS_ACCESS_KEY>
  secret_key: <AWS_SECRET_KEY>
  secure: true
```

4. Set Up Thanos Store Component

- The **Thanos Store** component fetches data from cloud storage (e.g., S3 or GCS).
- Set up the Store component in your Kubernetes environment and configure it to point to the cloud storage bucket.

Example thanos-store.yaml:

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: thanos-store
spec:
  replicas: 1
  selector:
    matchLabels:
      app: thanos-store
  template:
    metadata:
      labels:
        app: thanos-store
    spec:
      containers:
      - name: thanos
        image: thanosio/thanos:v0.25.0
        args:
          - store
          - --objstore.config-file=/etc/thanos/bucket.yaml
          - --http-address=0.0.0.0:10902
        volumeMounts:
          - mountPath: /etc/thanos
          name: bucket-config
      volumes:
        - name: bucket-config
          configMap:
            name: thanos-bucket-config
```

5. Set Up Thanos Querier

- The **Thanos Querier** aggregates queries from all Prometheus instances, allowing you to run global queries across multiple clusters.

Example thanos-querier.yaml:

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: thanos-querier
spec:
  replicas: 1
  selector:
    matchLabels:
      app: thanos-querier
  template:
    metadata:
      labels:
        app: thanos-querier
    spec:
      containers:
      - name: thanos
        image: thanosio/thanos:v0.25.0
        args:
          - query
          - --http-address=0.0.0.0:9090
          - --grpc-address=0.0.0.0:9091
```

6. Set Up Thanos Compactor (Optional)

- The **Thanos Compactor** will help optimize and compact data in long-term storage for efficient querying and storage.

Example thanos-compactor.yaml:

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: thanos-compactor
spec:
  replicas: 1
  selector:
    matchLabels:
      app: thanos-compactor
  template:
    metadata:
      labels:
        app: thanos-compactor
    spec:
      containers:
      - name: thanos
        image: thanosio/thanos:v0.25.0
        args:
          - compact
          - --objstore.config-file=/etc/thanos/bucket.yaml
          - --http-address=0.0.0.0:10902
        volumeMounts:
```

```
    - mountPath: /etc/thanos
      name: bucket-config
volumes:
  - name: bucket-config
    configMap:
      name: thanos-bucket-config
```

7. Integrate Grafana for Metrics Visualization

- Once Thanos is running, integrate **Grafana** to visualize the metrics.
- Add Thanos as a data source in Grafana:
  - URL: http://<thanos-querier-service>:9090
  - Select **Prometheus** as the data source type.

8. Monitor and Query Metrics

- Now you can query metrics from multiple Prometheus instances using Thanos and visualize them in a single Grafana dashboard.
- Set up alerts in Grafana or Prometheus to monitor important metrics like CPU usage, memory usage, and application performance.

9. Test and Optimize

- Test the setup by generating load across multiple clusters and ensure that Thanos is able to aggregate queries correctly.
- Optimize the configurations based on your infrastructure's needs and scale.

Project Deliverables:

- **Multiple Kubernetes clusters running Prometheus** instances.
- **Thanos Sidecar** running alongside Prometheus to upload data to cloud storage.
- **Thanos Store and Querier** components set up for global query support.
- **Grafana dashboard** integrated with Thanos for centralized monitoring.
- **Long-term storage** configured (e.g., AWS S3, GCS).
- A **complete set of Kubernetes YAML manifests** for the entire Thanos setup.

---

1. Multi-Cluster Metrics Aggregation

- **Scenario**: You are managing multiple Kubernetes clusters across different regions and need to aggregate monitoring data from all of them.
- **Solution with Thanos**: Thanos can be set up to aggregate metrics from different Prometheus instances running in each cluster. With Thanos's **Querier** component, you can perform **global queries** across all Prometheus instances, providing a unified view of metrics from multiple clusters on a single Grafana dashboard.
- **Benefit**: Simplifies monitoring across multiple regions and clusters, allowing for centralized dashboards and a complete view of system health.

2. High Availability for Prometheus

- **Scenario**: You rely on Prometheus for critical monitoring, but you cannot afford downtime or data loss if a Prometheus instance fails.

- **Solution with Thanos**: Thanos integrates a **Sidecar** with each Prometheus instance to provide high availability. In case one Prometheus instance goes down, another can take over, ensuring continuous data collection. Thanos also performs **deduplication** when querying across multiple Prometheus instances to avoid data duplication.

- **Benefit**: Continuous availability of Prometheus data without any downtime, enhancing the reliability of monitoring infrastructure.

3. Long-Term Metric Storage

- **Scenario**: You need to store time-series data for extended periods (e.g., years) for compliance, historical analysis, or debugging purposes.

- **Solution with Thanos**: Thanos allows Prometheus data to be stored in cloud object storage services such as **S3** or **Google Cloud Storage**, ensuring long-term retention. The **Compactor** component of Thanos optimizes and compresses data, making it cost-effective for long-term storage.

- **Benefit**: Cost-effective long-term storage of Prometheus metrics, allowing you to retain years of data while keeping infrastructure costs manageable.

4. Scalable Prometheus Monitoring for Large Environments

- **Scenario**: You have thousands of microservices running in multiple environments, and you need to scale Prometheus to handle massive amounts of time-series data.

- **Solution with Thanos**: Thanos enables Prometheus to scale horizontally by using **multiple Prometheus instances** in parallel, allowing for the efficient handling of billions of time-series data. Thanos uses cloud storage to store data, freeing local storage on Prometheus instances and providing better scalability.

- **Benefit**: Effortless scalability to handle large amounts of time-series data without performance degradation.

5. Disaster Recovery for Prometheus Data

- **Scenario**: You need to ensure that Prometheus data is recoverable in case of disaster, such as hardware failure or data corruption.

- **Solution with Thanos**: Thanos stores Prometheus data in **cloud storage** (e.g., S3 or GCS), which allows for disaster recovery. Even in the event of server failure, the data remains intact in the cloud, and Thanos's **Querier** component enables querying of the stored data.

- **Benefit**: Reduced risk of data loss and built-in disaster recovery for Prometheus metrics.

6. Cost Optimization with Cloud Storage

- **Scenario**: You have a growing amount of time-series data and want to optimize storage costs while retaining the ability to query historical data.

- **Solution with Thanos**: By offloading older Prometheus data to cloud object storage, Thanos allows you to take advantage of the lower cost of cloud storage for large datasets. Thanos's **Compactor** component optimizes storage usage, allowing for cheaper and efficient data storage.

- **Benefit**: Significant cost savings by using cloud storage for long-term retention while maintaining the ability to query historical data when needed.

7. Cross-Region Monitoring for Distributed Applications

- **Scenario**: Your application is distributed across multiple regions, and you need a way to monitor and query metrics from these geographically separated environments.

- **Solution with Thanos**: Thanos provides **global queries** across multiple Prometheus instances, even if they are in different regions. This is particularly useful for applications deployed in **multi-region cloud environments**, where you need to aggregate metrics and monitor system performance as a whole.

- **Benefit**: A unified view of system health, even across multiple regions or cloud providers, ensuring global observability.

8. SaaS Monitoring for Multiple Tenants

- **Scenario**: You operate a SaaS application and need to monitor metrics for each tenant separately, but also want to have a global overview of the health of your entire platform.

- **Solution with Thanos**: Thanos can aggregate metrics from multiple Prometheus instances, each dedicated to individual tenants. It allows for centralized global querying, but also maintains tenant isolation. Data deduplication ensures the accuracy of the data.

- **Benefit**: Efficient multi-tenant monitoring with the ability to query data across all tenants while ensuring data privacy and isolation.

9. Federated Prometheus Setup

- **Scenario**: You manage multiple isolated Prometheus instances, such as in different environments (production, staging, development), and need to query across them in a single dashboard.

- **Solution with Thanos**: Thanos allows for **federation** of Prometheus instances by providing global querying. You can run queries across Prometheus instances deployed in different environments and aggregate the results into a single Grafana dashboard.

- **Benefit**: Simplifies management of multiple Prometheus instances and allows for cross-environment monitoring.

10. Improved Query Performance Across Multiple Prometheus Instances

- **Scenario**: You are running multiple Prometheus instances, but querying across them is inefficient and slow.

- **Solution with Thanos**: Thanos improves query performance by using a **distributed query layer** that aggregates data from all Prometheus instances and optimizes queries.

- **Benefit**: Faster and more efficient queries across multiple Prometheus instances, leading to better performance in large-scale environments.

Prometheus Operator – Open-source operator to manage Prometheus deployments on Kubernetes.

🌟 Introduction to Prometheus Operator

Prometheus Operator is an **open-source Kubernetes Operator** that makes it easier to deploy and manage **Prometheus monitoring** components in a Kubernetes cluster.

🧠 What is Prometheus?

Prometheus is a powerful **monitoring and alerting system** used to collect metrics from your applications and infrastructure, especially in **cloud-native** environments like Kubernetes.

📌 Why Use Prometheus Operator?

Manually configuring Prometheus in Kubernetes can be complex. The Prometheus Operator:

- Automates setup and configuration of Prometheus, Alertmanager, and related resources.
- Uses **Custom Resource Definitions (CRDs)** like ServiceMonitor, Prometheus, and Alertmanager to simplify configurations.
- Keeps Prometheus running with the right setup — it's self-healing and declarative.
- Makes it easy to scale monitoring as your application grows.

⚙️ Installation of Prometheus Operator (Using kube-prometheus-stack via Helm)

Here's how you can install the **Prometheus Operator** using **Helm** (recommended way):

✅ Prerequisites

- A working Kubernetes cluster (e.g., Minikube, kind, or cloud-based)
- kubectl installed and configured
- Helm installed

🚀 Steps to Install:

1. Add the Prometheus Community Helm Repo:

```
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts
helm repo update
```

2. Create a namespace for monitoring (optional but recommended):

```
kubectl create namespace monitoring
```

3. Install kube-prometheus-stack (includes Prometheus Operator):

```
helm install prometheus-operator prometheus-community/kube-prometheus-stack
--namespace monitoring
```

4. Check the pods:

```
kubectl get pods -n monitoring
```

5. Access Prometheus UI (using port-forward):

```
kubectl port-forward svc/prometheus-operator-kube-prometheus-prometheus 9090
-n monitoring
```

Then open your browser at: http://localhost:9090

📚 What's Included in kube-prometheus-stack?

- Prometheus + Alertmanager

- Node Exporter
- Kube-state-metrics
- Grafana (for dashboards)

Project: Set Up Prometheus Monitoring on Kubernetes Using Prometheus Operator

---

Objective:

Learn how to deploy Prometheus Operator using Helm, configure it for monitoring, and observe your applications' metrics in a Kubernetes cluster.

Steps:

1. Set Up Kubernetes Cluster

Before starting with Prometheus, you'll need a running Kubernetes cluster. You can use **Minikube** or **Kind** for a local Kubernetes cluster, or use a cloud provider's Kubernetes service (e.g., AWS EKS, GCP GKE).

Install Minikube:

```
brew install minikube

minikube start
```

Install Kind (Kubernetes in Docker):

```
brew install kind
```

```
kind create cluster
```

2. Install Helm

Helm helps to manage Kubernetes applications, and you’ll use it to install Prometheus and related components.

Install Helm:

```
brew install helm
```

3. Install Prometheus Operator Using Helm

The best way to deploy Prometheus and Alertmanager is by using **Helm**. The Prometheus Operator makes it easy to manage Prometheus and its components in a Kubernetes cluster.

Add the Prometheus Helm Chart Repository:

```
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts

helm repo update
```

Install the Prometheus Operator: You can install the **kube-prometheus-stack**, which includes Prometheus, Alertmanager, and Grafana.

```
 helm install prometheus prometheus-community/kube-prometheus-stack
--namespace monitoring --create-namespace
```

Check if Prometheus is Running: After installation, you can verify that Prometheus is running by listing the pods in the monitoring namespace:

```
 kubectl get pods -n monitoring
```

4. Expose Prometheus UI

Prometheus has a web UI for querying and viewing metrics. You can access the Prometheus dashboard through port forwarding.

Port Forward Prometheus Service:

```
kubectl port-forward svc/prometheus-operated 9090:9090 -n monitoring
```

Now, you can open your browser and visit http://localhost:9090 to access the Prometheus web UI.

5. Monitor a Sample Application

To monitor your Kubernetes applications, you'll need to set up a **ServiceMonitor** to allow Prometheus to scrape metrics.

Deploy a Sample Application: Let's deploy a simple nginx application as an example:

```
kubectl create deployment nginx --image=nginx
```

```
kubectl expose deployment nginx --port=80 --target-port=80
```

Create a ServiceMonitor for the Application: Create a ServiceMonitor resource in a YAML file to instruct Prometheus to scrape the nginx metrics.

yaml

```
apiVersion: monitoring.coreos.com/v1

kind: ServiceMonitor

metadata:

  name: nginx-monitor
  namespace: monitoring
spec:
  selector:
    matchLabels:
      app: nginx
  endpoints:
    - port: http
      interval: 30s
```

Apply the ServiceMonitor:

```
kubectl apply -f service-monitor.yaml
```

6. Set Up Alertmanager for Alerts

Alertmanager is responsible for handling alerts from Prometheus.

Access Alertmanager: Expose the Alertmanager UI via port forwarding:

```
kubectl port-forward svc/prometheus-alertmanager 9093:9093 -n monitoring
```

Create Simple Alerts: To get alerts, you can configure alert rules. For example, create an alert rule if the CPU usage of your nginx app exceeds a certain threshold.

Example alert rule in alert-rules.yaml:

yaml

```
groups:
  - name: example-alerts
  rules:
    - alert: HighCPUUsage
      expr: avg(rate(container_cpu_usage_seconds_total{container!="",pod!="",namespace="default"}[5m])) by (pod) > 0.8
      for: 5m
      labels:
        severity: critical
      annotations:
        summary: "High CPU usage in pod {{ $labels.pod }}"
```

Apply the alert rule:

```
kubectl apply -f alert-rules.yaml
```

7. Monitor Metrics in Grafana

The kube-prometheus-stack includes **Grafana**, which allows you to visualize your metrics.

Access Grafana UI: Forward the Grafana service port to access the Grafana dashboard:

```
kubectl port-forward svc/prometheus-grafana 3000:80 -n monitoring
```

- Open your browser and go to http://localhost:3000. The default credentials are:
    - Username: admin

    - Password: prom-operator

- **Create Dashboards:** Once logged in, you can create dashboards to visualize your application's metrics. Grafana comes with pre-configured dashboards, including one for Kubernetes monitoring.

Key Learnings from this Project:

Kubernetes Basics: Learn how to set up and interact with a Kubernetes cluster.
Prometheus Operator: Install and manage Prometheus using the Prometheus Operator.
Monitoring & Alerts: Monitor applications and set up alerts for specific metrics.
Grafana Visualization: Use Grafana to visualize Prometheus metrics.

Conclusion:

By completing this project, you'll have set up Prometheus monitoring in a Kubernetes environment, integrated it with Grafana for visualization, and configured Alertmanager for alerting. This will give you a foundational understanding of monitoring in cloud-native environments, which is an essential skill for DevOps and site reliability engineering (SRE).

---

1. Application Performance Monitoring (APM)

Use Case:
Monitoring the performance of applications running in Kubernetes clusters, including tracking CPU, memory usage, and response times.

Description:
For cloud-native applications, especially microservices running in Kubernetes, it's critical to monitor how each service performs. Prometheus collects metrics such as:

- **CPU and Memory Usage:** Helps in identifying resource-intensive services and containers that may require scaling.

- **Response Times and Latencies:** Monitors how long your services are taking to respond and can alert you when response times exceed acceptable limits.

- **Request Rates and Error Rates:** Track the number of requests your service is receiving and the error rates to identify potential issues.

Example Metrics:

- http_requests_total{status="500", method="GET"} – total number of HTTP requests with a 500 error.

- container_cpu_usage_seconds_total – CPU usage per container.

---

2. Kubernetes Cluster Health Monitoring

Use Case:
Monitoring the health of Kubernetes nodes and the overall cluster to ensure that workloads are running smoothly.

Description:
Prometheus can help monitor the health and performance of your Kubernetes cluster components such as:

- **Node Health:** Track the status of nodes in the cluster (up, down, CPU, memory usage).
- **Pod Health:** Monitor pod statuses (whether they are running, pending, or failed).
- **API Server Performance:** Ensure the Kubernetes API server is responding efficiently to requests.

Example Metrics:

- kube_node_status_condition{condition="Ready", status="true"} – status of Kubernetes nodes.
- kube_pod_status_phase – status of pods in the cluster (running, failed, etc.).

---

3. Auto-scaling Based on Metrics

Use Case:
Automatically scaling applications based on metrics like CPU and memory usage.

Description:
Prometheus integrates well with the **Horizontal Pod Autoscaler (HPA)** in Kubernetes, which uses metrics collected by Prometheus to automatically scale workloads up or down. For example, if the CPU usage of a service exceeds a predefined threshold, Kubernetes can scale the pods running that service to ensure consistent performance.

Example Metrics:

- avg(rate(container_cpu_usage_seconds_total[5m])) by (pod) – average CPU usage for each pod.

- avg(container_memory_usage_bytes{container!="",pod!=""}) by (pod) – average memory usage by pod.

---

4. Alerts for System Anomalies

Use Case:
Setting up alerts to notify teams when specific thresholds are met, such as high CPU usage, memory consumption, or service downtime.

Description:
Prometheus allows you to define **alerting rules** that notify you when something goes wrong. Common alerts might include:

- **High CPU or Memory Usage:** Alert when a container's CPU usage exceeds a certain percentage.
- **Pod Failures:** Alert when a pod is in a failed state.
- **Service Downtime:** Alert when a service is unreachable or not responding.

Example Alert Rule:

yaml

```
alert: HighCPUUsage
expr:
avg(rate(container_cpu_usage_seconds_total{container!="",pod!="",namespace="default"}[5m])) by (pod) > 0.8
for: 5m
labels:
  severity: critical
annotations:
  summary: "High CPU usage in pod {{ $labels.pod }}"
```

5. Monitoring Databases in Kubernetes

Use Case:
Monitoring databases like MySQL, PostgreSQL, MongoDB, etc., running inside Kubernetes clusters to track performance and availability.

Description:
For databases running as part of microservices or cloud-native applications in Kubernetes, Prometheus can track metrics like:

- **Query Latency:** How long queries are taking to execute.
- **Connection Count:** Number of active database connections.
- **Disk Space Usage:** Ensuring there is enough disk space for database growth.

Example Metrics:

- mysql_global_status_threads_connected – current number of active MySQL connections.
- pg_stat_database_xact_commit – PostgreSQL transaction commit rate.

6. Continuous Integration (CI) and Continuous Deployment (CD) Monitoring

Use Case:
Monitor the performance and health of CI/CD pipelines running in Kubernetes clusters.

Description:
In CI/CD environments, Prometheus can monitor:

- **Job Execution Time:** Track how long each build or deployment takes.
- **Failure Rates:** Monitor the success/failure rates of build jobs or deployments.
- **Queue Length:** Monitor how many jobs are waiting in the queue, which may indicate bottlenecks.

Example Metrics:

- build_duration_seconds – duration of the build process.
- ci_cd_job_status – status of a CI/CD job (e.g., success, failure).

---

7. Monitoring Microservices in a Service Mesh

Use Case:
In a service mesh architecture (e.g., Istio, Linkerd), Prometheus is often used to monitor the performance of inter-service communication and the health of microservices.

Description:
Prometheus can be integrated with service meshes to collect metrics such as:

- **Request/Response Times:** Monitor the communication between services, including latency.
- **Success/Failure Rates:** Track the success or failure of service-to-service calls.

- **Traffic Flow:** Monitor the volume of traffic between services.

Example Metrics:

- istio_requests_total{response_code="200", destination_service="service-name"} – request count for a service.
- istio_request_duration_seconds – latency between services.

---

8. Tracking Resource Utilization of Containers

Use Case:
Prometheus can be used to track resource utilization by containers in a Kubernetes cluster, helping teams optimize resource allocation.

Description:
You can monitor:

- **CPU and Memory Usage per Container:** Track the resources consumed by containers.
- **Container Restarts:** Identify containers that are frequently restarting, which could indicate an issue.

Example Metrics:

- container_cpu_usage_seconds_total – CPU usage by container.
- container_memory_usage_bytes – Memory usage by container.

---

9. End-User Experience Monitoring (Synthetic Monitoring)

Use Case:
Simulate user traffic and monitor the application's performance from the user's perspective to track availability and response times.

Description:
By deploying synthetic monitoring scripts or jobs that simulate real user traffic (e.g., curl or HTTP requests to your application), Prometheus can track:

- **Response Time:** Time taken for requests to get a response.
- **Service Availability:** Whether the service is up and running.
- **Error Rate:** Rate at which errors occur when accessing the service.

Example Metrics:

- http_response_time_seconds – track response times for simulated user requests.
- http_requests_failed_total – count of failed requests.

---

10. Compliance and Security Monitoring

Use Case:
Use Prometheus to track security and compliance-related metrics within Kubernetes clusters.

Description:
Prometheus can be used to monitor:

- **Unauthorized Access Attempts:** Track failed login attempts or suspicious activity in the cluster.

- **Security Policy Violations:** Monitor Kubernetes security policies to ensure compliance.

- **Vulnerabilities in Containers:** Track the health of containers to ensure they are running secure versions of software.

Example Metrics:

- kube_pod_container_security_context – monitor security context of containers.

- audit_event_count – count of security-related audit events.

Conclusion:

Prometheus, along with the **Prometheus Operator**, is a powerful tool for monitoring and alerting in Kubernetes environments. Its flexibility makes it applicable for a variety of use cases, from monitoring infrastructure health to application performance, security, and resource utilization. As a new learner, working with these use cases will help you understand how to set up and use Prometheus effectively in different scenarios within Kubernetes.

Security & Compliance

Trivy – Vulnerability scanner for container images, file systems, and Git repos.

Trivy is an open-source security scanner used to detect vulnerabilities in container images, file systems, and Git repositories. It's a simple and powerful tool designed to help developers and DevOps teams ensure the security of their applications by identifying known vulnerabilities in the software dependencies and operating system packages.

Key Features of Trivy:

- **Comprehensive Scanning**: Trivy scans container images for both OS packages and application dependencies.

- **Fast and Easy**: Trivy performs quick vulnerability scans, making it easy to integrate into CI/CD pipelines.

- **Regularly Updated Databases**: The tool uses regularly updated vulnerability databases, so you are always protected against the latest threats.

- **Support for Multiple Artifact Types**: It can scan various artifact types such as container images, file systems, and Git repositories.

By using Trivy, you can quickly identify and fix vulnerabilities in your application before they become security risks, ensuring that your containers and the applications inside them are secure.

Installation of Trivy

Here’s a simple guide on how to install Trivy:

On Linux:
Install via Homebrew (if Homebrew is installed):

```
brew install aquasecurity/trivy/trivy
```

Install via script: You can download and install Trivy by running the following script:

```
curl -sfL
https://github.com/aquasecurity/trivy/releases/latest/download/trivy_$(uname
-s)-$(uname -m).tar.gz | tar -xz -C /usr/local/bin
```

Verify the Installation: After installation, verify it by checking the version:

```
trivy --version
```

On macOS:

Install via Homebrew:

```
brew install aquasecurity/trivy/trivy
```

Verify the Installation: Check the installation by running:

```
trivy --version
```

On Windows:

Install using Chocolatey: If you have Chocolatey installed, run the following command:

```
choco install trivy
```

1. **Manual Installation**: Download the latest release from the GitHub releases page and extract the binary to your system's path.

Running Trivy:

Once installed, you can run a basic scan on a container image using the following command:

```
trivy image <image_name>
```

Example:

```
trivy image nginx:latest
```

This will scan the nginx:latest image for vulnerabilities and provide you with a report.

That’s it! You now have Trivy installed and ready to use for scanning your container images and other artifacts for security vulnerabilities.

Project: Vulnerability Scanner for Docker Images using Trivy

Prerequisites

- Basic knowledge of Docker and how to create Docker images.
- Trivy installed (you can follow the installation steps I provided earlier).
- A simple web application (can be written in any language, but we will use **Node.js** for simplicity).
- Basic knowledge of Jenkins or any CI/CD pipeline tool (optional).

Project Overview

In this project, we will:

1. **Create a simple Node.js web application.**

2. **Create a Dockerfile to containerize the application.**
3. **Use Trivy to scan the Docker image for vulnerabilities.**
4. **Build a simple Jenkins pipeline to automate the process.**

Step 1: Create the Node.js Web Application

Let’s create a simple Node.js application.

Create a project folder:

```
mkdir trivy-vulnerability-scanner
cd trivy-vulnerability-scanner
```

Create a simple app.js file:

javascript

```
// app.js
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello, Trivy!');
});

app.listen(3000, () => {
  console.log('App running on port 3000');
});
```

Create a package.json file:

json

```
{
  "name": "trivy-vulnerability-scanner",
  "version": "1.0.0",
  "description": "A simple app to demonstrate Trivy",
  "main": "app.js",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

Install the required dependencies:

```
npm install
```

Now you have a basic Node.js app that runs on port 3000 and displays “Hello, Trivy!” when accessed via a web browser.

Step 2: Create the Dockerfile

Now, let's create a Dockerfile to containerize the Node.js app.

Create a Dockerfile in the project root directory:

Dockerfile

```
# Use the official Node.js image as a base
FROM node:14

# Set the working directory inside the container
WORKDIR /app
```

```
# Copy package.json and install dependencies
COPY package*.json ./
RUN npm install

# Copy the rest of the application code
COPY . .

# Expose the port the app will run on
EXPOSE 3000

# Run the app
CMD ["node", "app.js"]
```

Step 3: Build the Docker Image

Build the Docker image:

```
docker build -t trivy-vuln-app .
```

Run the Docker image (optional, to test the app):

```
docker run -p 3000:3000 trivy-vuln-app
```

You should be able to visit http://localhost:3000 and see “Hello, Trivy!”

Step 4: Scan the Docker Image with Trivy

Now, let’s use **Trivy** to scan the Docker image we just created for vulnerabilities.

Run Trivy scan:

```
trivy image trivy-vuln-app
```

Trivy will scan the image for any vulnerabilities and output a list of any vulnerabilities found, including their severity level (e.g., CRITICAL, HIGH, etc.).

Example output:

```
2025-04-20T12:33:45.123Z        8 vulnerabilities (2 critical, 6 high)
```

1. If no vulnerabilities are found, you’ll see a message indicating that the image is safe.

Step 5: Automate the Process with Jenkins Pipeline (Optional)

To make the process automated, let's integrate Trivy into a **Jenkins pipeline**.

Create a Jenkinsfile in the project root:

groovy

```
pipeline {
    agent any

    environment {
        DOCKER_IMAGE = 'trivy-vuln-app:latest'
    }

    stages {
        stage('Checkout Code') {
            steps {
                git 'https://github.com/yourusername/trivy-vuln-app.git'  // replace with your repo URL
            }
        }

        stage('Build Docker Image') {
            steps {
                script {
                    sh 'docker build -t $DOCKER_IMAGE .'
                }
            }
        }

        stage('Scan with Trivy') {
            steps {
                script {
                    sh 'trivy image --exit-code 1 --severity HIGH,CRITICAL $DOCKER_IMAGE'
                }
            }
        }

        stage('Deploy') {
            steps {
                script {
                    sh 'docker run -d -p 3000:3000 $DOCKER_IMAGE'
                }
            }
        }
    }

    post {
        always {
            cleanWs()
        }
    }
}
```

1. **Explanation of Jenkinsfile Stages**:

    - **Checkout Code**: Fetches the code from your GitHub repository.

- **Build Docker Image**: Builds the Docker image using the Dockerfile.
- **Scan with Trivy**: Runs the Trivy scan on the Docker image and fails the build if critical or high vulnerabilities are found (--exit-code 1).
- **Deploy**: If no vulnerabilities are found, deploy the container.

2. **Run the Jenkins Pipeline**:
   - Push the code to your GitHub repository.
   - Create a Jenkins job for this repository and add the Jenkinsfile.
   - Run the Jenkins pipeline, and it will automatically build, scan, and deploy your Docker image.

Conclusion

This simple project walks you through:

1. **Creating a Node.js application**.
2. **Containerizing it with Docker**.
3. **Scanning the Docker image for vulnerabilities using Trivy**.
4. **Automating the process with Jenkins** (optional).

---

1. CI/CD Pipeline Integration for Vulnerability Scanning

Use Case: Automatically scan Docker images for vulnerabilities before deployment in a CI/CD pipeline.

Problem: Without vulnerability scanning, Docker images with known security flaws might be deployed into production, risking security breaches and data leaks.

Solution: Integrate **Trivy** into your CI/CD pipeline (e.g., Jenkins, GitHub Actions, GitLab CI) to scan Docker images for vulnerabilities automatically at the end of the build process. If vulnerabilities are found (especially high or critical severity), the pipeline will fail, preventing the deployment of unsafe images.

Steps:

- During the Docker image build process, after building the image, run a **Trivy** scan as part of the pipeline.
- If vulnerabilities are found (critical or high severity), the pipeline is halted and developers are notified.
- If no vulnerabilities are found, proceed with the deployment.

Tools:

- Jenkins
- GitHub Actions
- GitLab CI
- Trivy

---

2. Scanning Images Before Pushing to Docker Registry

Use Case: Scan Docker images for vulnerabilities before pushing them to a public/private registry like Docker Hub or AWS ECR.

Problem: Docker images in public or private registries may contain vulnerabilities that could be exploited by attackers.

Solution: Use **Trivy** to scan images locally before pushing them to any registry, ensuring that only secure images are uploaded. This reduces the chances of malicious code entering the registry.

Steps:

- Build the Docker image locally.
- Run **Trivy** to scan the image for known vulnerabilities.
- If vulnerabilities are found, address them before pushing the image to the registry.

Tools:

- Docker
- Trivy
- Docker Hub / AWS ECR / Google Container Registry

---

3. Git Repository Vulnerability Scanning

Use Case: Scan application dependencies (in package.json, pom.xml, requirements.txt, etc.) in Git repositories for known vulnerabilities.

Problem: Code repositories might include insecure third-party libraries or dependencies that contain vulnerabilities, leading to potential security risks.

Solution: Use **Trivy** to scan the Git repository and its dependency files for vulnerabilities. This can be set up as part of the build process or as a scheduled task in a DevSecOps pipeline.

Steps:

- Run **Trivy** on the Git repository to scan dependency files (e.g., package.json, pom.xml, requirements.txt).
- If vulnerabilities are detected in any dependencies, alert the development team to update the vulnerable packages.

Tools:

- Git
- Trivy
- Jenkins/GitHub Actions/GitLab CI

---

4. File System Vulnerability Scanning

Use Case: Scan server or container file systems for vulnerabilities, misconfigurations, or outdated packages.

Problem: Servers or containers might have outdated or vulnerable OS packages, which can lead to exploits if not patched regularly.

Solution: Use **Trivy** to scan file systems, including all installed packages, for known vulnerabilities and misconfigurations. This can help secure the underlying operating system of containers or virtual machines.

Steps:

- Mount the file system or directory into a container.
- Run **Trivy** to scan the mounted file system for vulnerabilities.
- If any vulnerabilities are detected, patch the affected packages or update the system.

Tools:

- Trivy
- Docker

---

5. Monitoring and Auditing Docker Images in Production

Use Case: Regularly monitor Docker images in production for new vulnerabilities as they are discovered.

Problem: As new vulnerabilities are discovered in containerized applications, production images might become insecure over time, exposing the system to threats.

Solution: Set up regular scheduled scans of Docker images running in production, using **Trivy** to ensure that any new vulnerabilities are identified and mitigated.

Steps:

- Schedule regular scans of Docker images in production using **Trivy**.
- If vulnerabilities are found, assess the risk and plan for patching or rebuilding the image.
- Implement an automated update mechanism if necessary.

Tools:

- Trivy
- Cron jobs or CI/CD pipelines

---

6. Vulnerability Management in Multi-Cloud/Hybrid Cloud Environments

Use Case: Use **Trivy** to scan images and configurations across multiple cloud environments (e.g., AWS, Azure, Google Cloud) for vulnerabilities.

Problem: Organizations using multi-cloud or hybrid cloud environments need to ensure that their images are secure across different cloud platforms.

Solution: Integrate **Trivy** into the security monitoring tools in each cloud platform (e.g., AWS, Azure, Google Cloud). Regularly scan images and configurations for vulnerabilities to maintain security across all cloud environments.

Steps:

- Use **Trivy** to scan images on Docker Hub, AWS ECR, or Google Container Registry for vulnerabilities.
- Integrate vulnerability data into your centralized cloud security monitoring systems.
- Respond to critical vulnerabilities by redeploying patched images across the cloud platforms.

Tools:

- Trivy

- AWS / Azure / Google Cloud
- Security Monitoring Tools

---

7. Automated Patch Management in DevOps

Use Case: Automatically patch Docker images with outdated or vulnerable dependencies as part of the CI/CD process.

Problem: Vulnerable third-party packages in dependencies are often the root cause of security breaches, and keeping them up to date manually can be cumbersome.

Solution: Integrate **Trivy** with automated patch management systems to detect outdated or vulnerable packages. When vulnerabilities are identified, automatically trigger a rebuild of the Docker image with updated dependencies.

Steps:

- Use **Trivy** to scan Docker images during the CI/CD pipeline.
- If vulnerabilities are detected, use dependency management tools to update and patch the affected libraries.
- Rebuild the Docker image and redeploy the updated, secure version.

Tools:

- Trivy
- Docker
- Dependency Management Tools (e.g., npm, Maven, pip)

- CI/CD Tools (e.g., Jenkins, GitHub Actions, GitLab CI)

---

8. Compliance and Security Auditing for Containers

Use Case: Use Trivy to ensure compliance with industry standards (e.g., PCI-DSS, HIPAA) by scanning Docker images for vulnerabilities that may impact compliance.

Problem: Compliance regulations require organizations to ensure that their containerized applications meet specific security standards. Failure to meet these standards can result in penalties or data breaches.

Solution: Implement **Trivy** in your compliance checks, automatically scanning Docker images for vulnerabilities that could impact compliance requirements. Integrate the results into compliance audits and generate reports for security review.

Steps:

- Set up **Trivy** as part of the compliance pipeline.
- Run vulnerability scans on Docker images to identify issues that may breach compliance standards.
- Generate audit reports and use them to review compliance.

Tools:

- Trivy
- Compliance Tools
- CI/CD Tools (Jenkins, GitLab CI, etc.)

Conclusion

Trivy is an extremely versatile tool for vulnerability scanning, and the above use cases show how it can be applied in different contexts to improve security in DevOps, containerized environments, and CI/CD pipelines. By implementing Trivy in your workflows, you can ensure that vulnerabilities are caught early, reducing the risk of security breaches and maintaining secure deployments throughout the development lifecycle.

Kube-bench – Checks your clusters against CIS Kubernetes security benchmarks.

Kube-bench Introduction:

Kube-bench is an open-source tool designed to assess the security configuration of Kubernetes clusters. It checks if the cluster adheres to the security best practices outlined in the Center for Internet Security (CIS) Kubernetes Benchmark. This benchmark is a widely accepted set of security guidelines and standards to help organizations secure their Kubernetes environments. Kube-bench runs a series of checks that align with these guidelines and evaluates whether the Kubernetes configuration meets the benchmark's recommendations.

By using Kube-bench, Kubernetes administrators can:

- Verify the security compliance of their clusters.
- Identify potential misconfigurations or security risks.
- Improve the overall security posture of their Kubernetes environment.
- Ensure that the cluster complies with industry standards.

Installation of Kube-bench:

You can install Kube-bench on your system in different ways, including via Docker, Go, or a precompiled binary. Below are steps for installation via **Docker** and **Go**.

1. **Using Docker**:
    - Ensure Docker is installed on your machine.

Pull the latest Kube-bench Docker image:

```
docker pull aquasec/kube-bench:latest
```

To run Kube-bench using Docker, execute:

```
docker run --rm -v /etc:/etc -v /var/run:/var/run aquasec/kube-bench:latest
```

2. This command will run Kube-bench and bind necessary volumes to check the configuration files.
3. **Using Go**:
    - Install Go if it's not installed already. Follow the instructions here: https://golang.org/doc/install.

Once Go is installed, download the Kube-bench source code:

```
git clone https://github.com/aquasecurity/kube-bench.git
cd kube-bench
```

Install Kube-bench:

You can now run Kube-bench locally by executing the binary file:

```
./kube-bench
```

4. **Using Precompiled Binary**:

   - Visit the Kube-bench GitHub releases page: https://github.com/aquasecurity/kube-bench/releases.

   - Download the appropriate binary for your operating system.

   - Extract the binary and move it to a directory in your PATH (e.g., /usr/local/bin).

Now you can run Kube-bench from the command line:
kube-bench

✅ Project: Kubernetes Cluster Security Audit using Kube-bench

🎯 Goal:

Check your Kubernetes cluster's security posture using kube-bench and ensure it follows the **CIS Kubernetes Benchmark**.

📁 Project Structure:

arduino

kube-bench-security-audit/

│

├── README.md

├── run-kube-bench.sh

├── kube-bench-report.txt

├── Jenkinsfile (optional for CI/CD)

```
└── screenshots/ (optional folder to keep images for blog/demo)
```

📘 Step-by-Step Project Guide

1️⃣Prerequisites

- Docker installed and running
- A Kubernetes cluster (can use kind, minikube, or a cloud-managed one)
- Basic Linux/terminal knowledge

2️⃣Project Initialization

Create a directory:

```
mkdir kube-bench-security-audit && cd kube-bench-security-audit
```

3️⃣🔍 Create Shell Script to Run Kube-bench

Create a file named run-kube-bench.sh:

```
#!/bin/

echo "🚀 Running kube-bench against your Kubernetes cluster..."
docker pull aquasec/kube-bench:latest
docker run --rm \
  -v /etc:/etc:ro \
  -v /var/run:/var/run:ro \
  -v $(pwd):/output \
  aquasec/kube-bench:latest > ./kube-bench-report.txt

echo "✅ Report generated: kube-bench-report.txt"
```

Make it executable:

```
chmod +x run-kube-bench.sh
```

4️⃣ 📄 Create README.md

Here's a sample README.md:

markdown

```
# 🔐 Kubernetes Cluster Security Audit with Kube-bench

This project checks your Kubernetes cluster against CIS benchmarks using the open-source tool `kube-bench`.

## 🛠️ Requirements
```

- Docker
- Kubernetes cluster (minikube/kind/production)
- terminal

## 🚀 How to Run

```
chmod +x run-kube-bench.sh

./run-kube-bench.sh
```

📁 **Output**

- kube-bench-report.txt will contain the full security audit report.

📌 **Reference**

- Kube-bench GitHub
- CIS Kubernetes Benchmark

yaml

### 5️⃣ (Optional) CI/CD Integration with Jenkins

Create a `Jenkinsfile`:

```groovy
pipeline {
    agent any
    stages {
        stage('Pull Kube-bench Image') {
            steps {
                sh 'docker pull aquasec/kube-bench:latest'
            }
        }
        stage('Run Kube-bench') {
            steps {
                sh 'docker run --rm -v /etc:/etc:ro -v /var/run:/var/run:ro -v $(pwd):/output aquasec/kube-bench:latest > kube-bench-report.txt'
            }
        }
        stage('Archive Report') {
            steps {
                archiveArtifacts artifacts: 'kube-bench-report.txt', fingerprint: true
            }
        }
    }
}
```

6️⃣📸 (Optional) Add Screenshots

Create a folder screenshots/ and save images if you want to demonstrate your output for a LinkedIn/blog post.

7️⃣✅ Final Test

Run the tool:

```
./run-kube-bench.sh
```

Check the output:

```
cat kube-bench-report.txt
```

You’ll see lines like:

```
pgsql

[INFO] 1.1 - Master Node Configuration
```

```
[PASS] 1.1.1 Ensure that the API server only allows authorized users

[FAIL] 1.1.2 Ensure that anonymous-auth is disabled
```

🧠 What You Learn

- Kube-bench setup and execution
- Docker and volume mounts
- Kubernetes security checks
- Automating security audits in CI/CD

🗂️ Project Zip (Manual)

You can zip it up to share or keep:

```
zip -r kube-bench-security-audit.zip kube-bench-security-audit/
```

🔍 Use Cases of Kube-bench

🔐 1. Kubernetes Security Compliance Audit

Goal: Ensure your Kubernetes cluster complies with CIS Kubernetes Security Benchmarks.

Use Case:
An organization wants to follow security best practices defined by the Center for Internet Security (CIS). They use kube-bench to automatically check if all cluster components (like API server, scheduler, kubelet, etc.) meet the recommended configuration.

✅ *Ensures industry-standard security compliance.*

🧪 2. Automated Security Testing in CI/CD Pipelines

Goal: Catch misconfigurations before deploying to production.

Use Case:
A DevSecOps team integrates kube-bench into their Jenkins/GitHub Actions pipeline. Before deploying a new Kubernetes cluster or app, the pipeline runs kube-bench to generate a security report.

✅ *Prevents insecure clusters from being deployed.*

🧰 3. Cluster Hardening During Provisioning

Goal: Harden the Kubernetes cluster at the time of setup.

Use Case:
During cluster provisioning (using Terraform, Kubeadm, etc.), engineers use kube-bench to validate configuration before moving to the application deployment stage.

✅ *Enforces secure defaults from Day 1.*

📊 4. Security Posture Reporting

Goal: Generate reports for security audits and management.

Use Case:
Security teams run kube-bench weekly or monthly and store the reports in dashboards or logs. These reports help in tracking improvements over time and preparing for security audits.

✅ *Improves transparency and traceability of security.*

🛠️ 5. Misconfiguration Detection in Development Clusters

Goal: Detect weak security settings in dev environments.

Use Case:
Developers often run Minikube or Kind clusters locally. Using kube-bench, they can scan those environments to learn about real-world security issues (e.g., anonymous access, privileged containers, missing audit logging).

✅ *Promotes security awareness among developers.*

🧑‍🏫 6. Learning and Training

Goal: Teach Kubernetes security best practices.

Use Case:
Learners and trainers use kube-bench as a tool to understand which settings are secure/insecure in Kubernetes. Each check explains what it does and how to fix it.

✅ *Hands-on way to learn Kubernetes security.*

🌐 7. Cloud Migration Readiness

Goal: Ensure clusters are secure before migrating workloads to the cloud.

Use Case:
A company moving workloads to GKE, EKS, or AKS runs kube-bench to validate security and compliance before transitioning to managed Kubernetes services.

✅ *Reduces risk during cloud adoption.*

🔁 8. Regular Security Scans with CronJobs

Goal: Run periodic scans automatically.

Use Case:
You schedule a Kubernetes CronJob or system-level cron to run kube-bench daily/weekly and send results to an S3 bucket or security dashboard.

✅ *Automates continuous cluster monitoring.*

Kubesec – Static analysis for Kubernetes manifests.

Kubesec is a lightweight **static analysis tool** designed to help you **identify security issues in your Kubernetes YAML manifests** (like Deployment, Service, StatefulSet, etc.).

🔍 What does it do?

Kubesec scans your Kubernetes YAML files and checks them against **predefined security rules**. It helps you:

- Detect insecure configurations
- Identify missing security best practices
- Avoid risky deployments before they reach production

✅ Why should you use it?

- Helps **shift security left** in the CI/CD pipeline
- Prevents misconfigurations like missing readOnlyRootFilesystem, running as root, or lack of resource limits
- Useful for **DevSecOps**, especially in automated pipelines

---

⚙️ Kubesec Installation Steps

You can use **Kubesec CLI** or access it via **API**.

Option 1: Using Docker (recommended for quick use)

```
docker run -i kubesec/kubesec scan < your-k8s-manifest.yaml
```

Option 2: Install Kubesec CLI locally

Step 1: Download the binary

Visit the Kubesec GitHub Releases page and download the appropriate binary for your OS.

Example for Linux:

```
curl -LO https://github.com/controlplaneio/kubesec/releases/latest/download/kubesec_linux_amd64
chmod +x kubesec_linux_amd64
sudo mv kubesec_linux_amd64 /usr/local/bin/kubesec
```

Step 2: Verify installation

```
kubesec --help
```

🧪 Example Usage

```
kubesec scan nginx-deployment.yaml
```

It will return a **score** and **list of findings** with suggestions to improve security.

🛡️ Project: Securing Kubernetes Manifests with Kubesec

📁 Project Structure

```
kubesec-project/
├── manifests/
│   ├── insecure-deployment.yaml
│   └── secure-deployment.yaml
├── Dockerfile
├── .github/
│   └── workflows/
│       └── kubesec.yml
└── README.md
```

📜 1. README.md

This is the project's documentation file. It provides an overview of the project, its purpose, and instructions on how to set it up.

Contents of README.md:

markdown

# Securing Kubernetes Manifests with Kubesec

## Introduction
This project demonstrates how to integrate **Kubesec** into a **CI/CD pipeline** to **automatically check the security of Kubernetes YAML manifests** before deployment. We use Kubesec, a tool that performs static analysis to ensure your Kubernetes configurations follow best security practices.

## Project Files
- **manifests/insecure-deployment.yaml**: An insecure Kubernetes deployment manifest.
- **manifests/secure-deployment.yaml**: A secure Kubernetes deployment manifest.
- **Dockerfile**: A simple Dockerfile to run Kubesec.
- **.github/workflows/kubesec.yml**: A GitHub Actions workflow to automate Kubesec scans.

## Setup

1. **Clone the repository**:
   ```
   git clone https://github.com/your-username/kubesec-project.git
   cd kubesec-project
   ```

Run Kubesec using Docker:

```
docker run --rm -v $(pwd)/manifests:/manifests kubesec/kubesec scan /manifests/insecure-deployment.yaml
```

2. **Automate Kubesec using GitHub Actions**: This repository contains a GitHub Actions workflow to run Kubesec automatically when a YAML file is pushed to the repository. See .github/workflows/kubesec.yml.

Results

You’ll see a security score and suggestions for fixing vulnerabilities in your YAML manifests. If the score is too low, the deployment should be improved before proceeding to production.

yaml

### 📂 **2. `manifests/insecure-deployment.yaml`**
This file represents a Kubernetes manifest with **insecure configurations**. It has a simple deployment of Nginx without security practices like non-root users, read-only root filesystem, etc.

Contents of `insecure-deployment.yaml`:
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: insecure-nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
```

Explanation:

- **Missing security context**: The container runs as the root user, which is a security risk.

- **Lacks resource limits**: This can lead to performance issues or resource exhaustion.

- **No security restrictions**: The filesystem is not read-only, allowing containers to modify it.

📂 3. manifests/secure-deployment.yaml

This file represents a **secure version** of the Kubernetes manifest. It includes security best practices such as running the container as a non-root user and using a read-only root filesystem.

Contents of secure-deployment.yaml:

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secure-nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
```

```
  securityContext:
    runAsNonRoot: true
  containers:
  - name: nginx
    image: nginx:stable
    securityContext:
      readOnlyRootFilesystem: true
      allowPrivilegeEscalation: false
```

Explanation:

- **Non-root user**: The runAsNonRoot: true ensures the container runs as a non-root user, reducing the attack surface.
- **Read-only filesystem**: The readOnlyRootFilesystem: true prevents modifications to the container's filesystem, which is a security best practice.
- **Disallow privilege escalation**: The allowPrivilegeEscalation: false ensures that the container cannot escalate its privileges.

📂 4. Dockerfile

This file creates a Docker image that includes the **Kubesec** tool. This is useful for running Kubesec in a local environment or CI/CD pipeline.

Contents of Dockerfile:

Dockerfile

```
# Use an official Ubuntu as a parent image
FROM ubuntu:20.04

# Install Kubesec CLI
```

```
RUN apt-get update && apt-get install -y \
  curl \
  && curl -LO https://github.com/controlplaneio/kubesec/releases/latest/download/kubesec_linux_amd64 \
  && chmod +x kubesec_linux_amd64 \
  && mv kubesec_linux_amd64 /usr/local/bin/kubesec

# Set the entrypoint
ENTRYPOINT ["kubesec"]
```

Explanation:

- **Ubuntu base image**: The Docker image is based on Ubuntu 20.04.
- **Kubesec installation**: It installs the latest version of the Kubesec CLI.
- **ENTRYPOINT**: When the container runs, it will invoke the kubesec command by default.

📂 5. .github/workflows/kubesec.yml

This is a **GitHub Actions workflow** to automate the scanning of Kubernetes manifests using Kubesec every time a change is pushed to the manifests directory.

Contents of .github/workflows/kubesec.yml:

yaml

```
name: Kubesec Scan

on:
  push:
    paths:
      - 'manifests/*.yaml'

jobs:
  scan:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v2

      - name: Run Kubesec on manifests
        run: |
          docker run --rm -v ${{ github.workspace }}/manifests:/manifests \
            kubesec/kubesec scan /manifests/insecure-deployment.yaml \
            > kubesec-report.json
          cat kubesec-report.json
```

Explanation:

- **Trigger on push**: This workflow is triggered whenever a change is pushed to any YAML file in the manifests folder.
- **Run Kubesec using Docker**: It runs the Kubesec Docker container and scans the insecure-deployment.yaml file for security issues.
- **Output report**: The results are saved in kubesec-report.json, which contains security findings and recommendations.

🚀 How to Run the Project

Step 1: Clone the repository

```
git clone https://github.com/your-username/kubesec-project.git
cd kubesec-project
```

Step 2: Run Kubesec Locally

You can run Kubesec manually on your YAML files with Docker:

```
docker run --rm -v $(pwd)/manifests:/manifests kubesec/kubesec scan
/manifests/insecure-deployment.yaml
```

Step 3: GitHub Actions

- Push your changes to GitHub.
- The GitHub Actions workflow will automatically run and show the results in the GitHub Actions tab.

📊 Expected Output

After running Kubesec, you will see a security report, such as:

json

```
{
  "score": 35,
  "findings": [
    {
      "severity": "high",
      "message": "Container is running as root. Consider using non-root user."
    },
    {
      "severity": "medium",
    "message": "Missing resource limits. Set CPU and memory limits."
   }
 ]
}
```

This is a basic **Kubesec-based DevSecOps project** that checks Kubernetes manifests for vulnerabilities before deployment. It introduces basic concepts of security in Kubernetes and how to automate security checks using **Docker** and **GitHub Actions**.

OPA (Open Policy Agent) – Enforce policies on Kubernetes resources.

OPA (Open Policy Agent) – Enforce Policies on Kubernetes Resources

OPA (Open Policy Agent) is a free and open-source tool that helps you write and apply rules (called **policies**) to control how things work in your system.

When you use **OPA with Kubernetes**, you can make sure that certain rules are followed automatically. For example:

- Only **secure container images** (safe versions of software) should be allowed to run.

- **Namespaces** (a way to organize Kubernetes resources) must follow certain access rules.

OPA works closely with **Kubernetes' API server**. So, when someone tries to create or update a Kubernetes resource (like a Pod, Deployment, or Namespace), OPA checks your rules **right away** and **allows or denies** the request based on those rules.

This helps keep your Kubernetes environment **safe, secure, and under control**—without needing to check everything manually.

🛠️ Step-by-Step: Install OPA Gatekeeper on Kubernetes

📌 Prerequisites

- A running **Kubernetes cluster** (minikube, kind, or cloud-based)
- **kubectl** installed and configured

✅ Step 1: Install Gatekeeper

Run the following command to install OPA Gatekeeper using the official manifest:

```
kubectl apply -f
https://raw.githubusercontent.com/open-policy-agent/gatekeeper/release-3.14/deploy/gatekeeper.yaml
```

This command will:

- Create Gatekeeper pods
- Install the controller and necessary Custom Resource Definitions (CRDs)

🔍 Step 2: Verify the Installation

Check that the Gatekeeper pods are running:

```
kubectl get pods -n gatekeeper-system
```

You should see pods like:

```
gatekeeper-audit-xxxx

gatekeeper-controller-manager-xxxx
```

🧪 Step 3: Test with a Sample Constraint

Let’s try a **sample policy**: Disallow containers running as root.

First, create a **ConstraintTemplate**:

yaml

```
# template.yaml
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8srequiredlabels
spec:
  crd:
    spec:
      names:
        kind: K8sRequiredLabels
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8srequiredlabels

        violation[{"msg": msg}] {
          missing := {label | input.review.object.metadata.labels[label] == ""}
          count(missing) > 0
          msg := sprintf("Missing required labels: %v", [missing])
        }
```

Apply it:

```
kubectl apply -f template.yaml
```

Now define the constraint using the template:

yaml

```
# constraint.yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
```

```
metadata:
  name: ns-must-have-team
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Namespace"]
  parameters:
    labels: ["team"]
```

Apply it:

```
kubectl apply -f constraint.yaml
```

Now try creating a namespace **without the team label**, and you’ll see it gets blocked.

🧪 OPA Gatekeeper Project for Beginners: Enforce Namespace Label Policy

📁 Project Structure

pgsql

```
opa-k8s-policy-project/
```

```
├── 1-install-gatekeeper.yaml
├── 2-constraint-template.yaml
├── 3-constraint.yaml
├── README.md
```

📦 1. 1-install-gatekeeper.yaml

You don't need to write this manually. Use the following command to apply it:

```
kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/release-3.14/deploy/gatekeeper.yaml
```

This sets up OPA Gatekeeper on your Kubernetes cluster.

📄 2. 2-constraint-template.yaml

This file creates a **template** for a policy that requires specific labels on namespaces.

yaml

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8srequiredlabels
spec:
  crd:
    spec:
      names:
        kind: K8sRequiredLabels
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8srequiredlabels

        violation[{"msg": msg}] {
          required := input.parameters.labels
          provided := input.review.object.metadata.labels
          missing := {label | label := required[_]; not provided[label]}
          count(missing) > 0
          msg := sprintf("Missing required labels: %v", [missing])
        }
```

📄 3. 3-constraint.yaml

This file creates the actual **policy** using the template. It enforces that every namespace must have a team label.

yaml

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: ns-must-have-team-label
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Namespace"]
  parameters:
    labels: ["team"]
```

✅ How to Run the Project

Create a new folder and move into it:

```
mkdir opa-k8s-policy-project && cd opa-k8s-policy-project
```

1. Save the above 2 YAML files as 2-constraint-template.yaml and 3-constraint.yaml.

Install Gatekeeper:

```
kubectl apply -f
```

```
https://raw.githubusercontent.com/open-policy-agent/gatekeeper/release-3.14/deploy/gatekeeper.yaml
```

Wait for pods to come up:

```
kubectl get pods -n gatekeeper-system
```

Apply the constraint template:

```
kubectl apply -f 2-constraint-template.yaml
```

Apply the constraint:

```
kubectl apply -f 3-constraint.yaml
```

Test by trying to create a namespace **without a team label**:

```
kubectl create ns test-ns
```

You should see an error like:

swift

```
admission webhook "validation.gatekeeper.sh" denied the request:
[ns-must-have-team-label] Missing required labels: {"team"}
```

🧠 Use Case: Enforce Image Registry

🎯 Goal

Only allow container images from trusted registries like docker.io/your-org/, ghcr.io/, etc.

📘 Why?

To avoid using images from unverified or public sources that may be insecure.

✅ ConstraintTemplate: k8strustedimageregistry

yaml

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8strustedimageregistry
spec:
  crd:
    spec:
      names:
        kind: K8sTrustedImageRegistry
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8strustedimageregistry

        violation[{"msg": msg}] {
          image := input.review.object.spec.containers[_].image
          not startswith(image, "docker.io/your-org")
          not startswith(image, "ghcr.io/")
          msg := sprintf("Image %s is not from a trusted registry", [image])
        }
```

✅ Constraint

yaml

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sTrustedImageRegistry
metadata:
  name: only-trusted-registries
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
```

🧠 Use Case: Deny HostPath Volumes

🎯 Goal

Block all usage of hostPath volumes to prevent access to the host filesystem.

📘 Why?

Mounting host paths can lead to privilege escalation or data leakage.

✅ ConstraintTemplate: k8snohostpath

yaml

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8snohostpath
spec:
  crd:
    spec:
      names:
        kind: K8sNoHostPath
  targets:
    - target: admission.k8s.gatekeeper.sh
```

```
      rego: |
        package k8snohostpath

        violation[{"msg": msg}] {
          hostPath := input.review.object.spec.volumes[_].hostPath
          msg := "Using hostPath volumes is not allowed."
        }
```

✅ Constraint

yaml

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sNoHostPath
metadata:
  name: deny-hostpath
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
```

🧠 Use Case: Enforce Resource Limits

🎯 Goal

Ensure all containers define CPU and memory requests & limits.

📘 Why?

Helps with scheduling and prevents a single container from consuming all node resources.

✅ ConstraintTemplate: k8sresourcelimits

yaml

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8sresourcelimits
spec:
  crd:
    spec:
      names:
        kind: K8sResourceLimits
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8sresourcelimits

        violation[{"msg": msg}] {
          container := input.review.object.spec.containers[_]
          not container.resources.limits.cpu
          msg := "CPU limit not set."
        }

        violation[{"msg": msg}] {
          container := input.review.object.spec.containers[_]
          not container.resources.requests.memory
          msg := "Memory request not set."
        }
```

✅ **Constraint**

yaml

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sResourceLimits
metadata:
  name: enforce-limits
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
```

🧠 Use Case: Enforce Read-Only Root Filesystem

🎯 Goal

Containers should run with a read-only root filesystem.

📘 Why?

Helps reduce the risk of file tampering or malware persistence.

✅ ConstraintTemplate: k8sreadonlyrootfs

yaml

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8sreadonlyrootfs
spec:
  crd:
    spec:
      names:
        kind: K8sReadOnlyRootFilesystem
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8sreadonlyrootfs
        violation[{"msg": msg}] {
          container := input.review.object.spec.containers[_]
          not container.securityContext.readOnlyRootFilesystem
          msg := "readOnlyRootFilesystem is not enabled."
        }
```

✅ Constraint

yaml

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sReadOnlyRootFilesystem
metadata:
  name: enforce-readonly-rootfs
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
```

🧠 Use Case: Disallow Host Networking

🎯 Goal

Prevent Pods from using host networking.

📘 Why?

Host networking bypasses Kubernetes' network isolation.

✅ ConstraintTemplate: k8snohostnetwork

yaml

```
apiVersion: templates.gatekeeper.sh/v1beta1
```

```
kind: ConstraintTemplate
metadata:
  name: k8snohostnetwork
spec:
  crd:
    spec:
      names:
        kind: K8sNoHostNetwork
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8snohostnetwork

        violation[{"msg": msg}] {
          input.review.object.spec.hostNetwork == true
          msg := "Using hostNetwork is not allowed."
        }
```

✅ **Constraint**

yaml

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sNoHostNetwork
metadata:
  name: deny-host-network
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
```

Falco – An open-source runtime security monitoring tool for detecting abnormal behavior in Kubernetes and containers.

🔐 What is Falco?

Falco is an **open-source runtime security tool** designed to monitor and detect abnormal behavior in **Kubernetes clusters** and **containers**. It acts like a **security camera** for your containerized applications, watching what they do in real-time and alerting you if something suspicious happens.

🚀 Why Use Falco?

Containers and Kubernetes are powerful, but they also introduce new security risks. Traditional security tools may not see what's happening **inside containers**. Falco helps fill that gap by:

- Watching for **unusual system activity**
- Alerting when it detects **unexpected behaviors**
- Helping you **respond to threats quickly**

🔎 What Can Falco Detect?

Falco uses **rules** to define what "normal" behavior looks like. It can detect things like:

- A container trying to read sensitive files (e.g., /etc/shadow)
- Unexpected network connections

- Shells being launched inside containers
- Changes to configuration files
- Unauthorized access attempts

🛠️ How Does Falco Work?

- **Syscalls Monitoring**: Falco listens to system calls made by containers or host processes. This means it sees everything happening in real-time at the OS level.
- **Rules Engine**: It uses a flexible set of rules to decide what’s suspicious.
- **Alerts**: When something matches a rule, Falco sends an alert (e.g., logs, Slack, email, etc.).

⚙️ Where is Falco Used?

Falco is especially useful in environments where **Kubernetes** and **Docker** are used. It integrates well with:

- Kubernetes clusters (to secure pods)
- CI/CD pipelines (for runtime security)
- Cloud-native platforms (like AWS, GCP, Azure)

🧠 Example Rule (Human-Friendly)

yaml

```yaml
- rule: Write below etc
  desc: >
    Detect any process that tries to write below /etc
  condition: >
    evt.type = write and fd.name startswith /etc
  output: >
    File below /etc opened for writing (user=%user.name command=%proc.cmdline)
```

This rule alerts you if **any process writes to the /etc directory**, which could indicate a config tampering attack.

✅ Key Benefits

- Real-time threat detection
- Works well with Kubernetes & containers
- Customizable rules
- Lightweight and open-source
- Backed by CNCF (Cloud Native Computing Foundation)

📌 Option 1: Install Falco on Kubernetes (using Helm – recommended way)

✅ Prerequisites:

- Kubernetes cluster (e.g., Minikube, Kind, or a cloud provider)
- kubectl installed and configured
- helm installed (Helm 3+)

📥 Step-by-step:

```
# Add the Falco Helm repo
helm repo add falcosecurity https://falcosecurity.github.io/charts

# Update repo to get the latest charts
helm repo update

# Install Falco
helm install falco falcosecurity/falco
```

> Falco will now run as a **DaemonSet**, meaning one pod per node, monitoring system calls.

📌 Option 2: Install Falco on a Linux Host (outside Kubernetes)

✅ Prerequisites:

- Linux system (Ubuntu, CentOS, etc.)
- Kernel headers must be installed

📥 Step-by-step:

```
# Download and install Falco
curl -s https://falco.org/install.sh | sudo
```

> This script auto-detects your OS and sets up everything needed to run Falco.

📌 Option 3: Run Falco with Docker (for testing or local dev)

✅ Prerequisites:

- Docker installed

📥 Step-by-step:

```
docker run -i -t --name falco \
  --privileged \
  -v /var/run/docker.sock:/host/var/run/docker.sock \
  -v /dev:/host/dev \
  -v /proc:/host/proc:ro \
  -v /boot:/host/boot:ro \
  -v /lib/modules:/host/lib/modules:ro \
  -v /usr:/host/usr:ro \
  falcosecurity/falco:latest
```

> ⚠️ --privileged and mounting /proc, /dev, etc., are necessary so Falco can monitor the host system.

✅ Check Falco is Running

```
kubectl get pods -n default
```

Or if using Docker:

```
docker ps
```

Then check logs:

```
docker logs falco
```

🧪 Test It (Optional)

To simulate suspicious activity (e.g., writing to /etc/passwd):

```
echo "test" | sudo tee -a /etc/passwd
```

Falco should immediately detect this and log an alert 🎯.

🔐 Falco Security Project: Detect Suspicious Activity in Kubernetes

🎯 Goal

Set up Falco in a Kubernetes cluster and trigger a security alert when a container tries to access or modify sensitive files like /etc/passwd.

✅ Project Overview

Tools Used:

- Kubernetes (Minikube, Kind, or any cluster)
- Falco (installed via Helm)
- A test pod (to simulate abnormal behavior)
- Custom Falco rule (optional, to detect specific activities)

🔧 Step 1: Setup Kubernetes Cluster

If you don’t have one already, use kind:

```
kind create cluster --name falco-demo
```

Or use Minikube:

```
minikube start
```

🐸 Step 2: Install Falco using Helm

```
helm repo add falcosecurity https://falcosecurity.github.io/charts
helm repo update
helm install falco falcosecurity/falco
```

Check that it’s running:

```
kubectl get pods
```

🧪 Step 3: Deploy a Test Pod

Create a simple test pod that will simulate suspicious behavior.

test-pod.yaml

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
  containers:
  - name: busybox
    image: busybox
    command: ["sleep", "3600"]
    securityContext:
      privileged: true
```

Apply it:

```
kubectl apply -f test-pod.yaml
```

🚨 Step 4: Simulate a Security Threat

Open a shell into the container:

```
kubectl exec -it busybox -- sh
```

Then run:

```
echo "hacked!" >> /etc/passwd
```

This mimics someone modifying a critical file.

📥 Step 5: Check Falco Alerts

Falco logs can be viewed with:

```
kubectl logs -l app=falco
```

You should see an alert like:

```
File below /etc opened for writing (user=root command=echo hacked!)
```

✍️ (Optional) Step 6: Add a Custom Falco Rule

Let’s create a simple rule that alerts if anyone opens /secret-data.txt.

Add to the falco_rules.local.yaml file:

yaml

```
- rule: Access secret data file
  desc: Detect access to /secret-data.txt
  condition: open.file.name = "/secret-data.txt"
  output: "Secret file accessed (user=%user.name command=%proc.cmdline)"
  priority: WARNING
  tags: [security]
```

You can mount this file into Falco as a ConfigMap for real-world usage.

📌 What You Learned

- How to set up Falco in Kubernetes
- How to simulate and detect suspicious behavior
- Basics of Falco rule customization

🔍 Falco Use Cases in DevOps & Security

🛡️ 1. Detect Shells in Containers

Use Case: Prevent attackers or insiders from opening a shell inside containers.

Example Detection:

> Falco can alert if someone runs , sh, or zsh inside a container.

Why it matters: Shell access may indicate someone is trying to explore or tamper with the container.

🔒 2. Unauthorized File Access

Use Case: Detect access to sensitive files like /etc/shadow, /etc/passwd, or app secrets.

Example Detection:

> Any attempt to read or write to protected files.

Why it matters: Critical for detecting privilege escalation or credential theft attempts.

🧨 3. Suspicious Network Connections

Use Case: Detect unexpected outbound connections to strange IPs or ports.

Example Detection:

> A container tries to connect to a known malicious IP or opens a reverse shell.

Why it matters: Indicates data exfiltration or command & control communication.

🛠️ 4. Changes to Binary or Configuration Files

Use Case: Alert when important system files or app configs are modified.

Example Detection:

> Unexpected edits to /usr/bin, /etc/kubernetes/, or configmaps.

Why it matters: Helps detect tampering or configuration drift in critical components.

🔐 5. Privilege Escalation Attempts

Use Case: Catch when containers or users try to escalate privileges (e.g., using sudo or changing user IDs).

Example Detection:

> Processes running as root or attempting to change UID.

Why it matters: Early signs of a breach or misconfigured security context.

🔄 6. Abnormal Process Activity

Use Case: Detect new or unexpected processes running inside containers or nodes.

Example Detection:

> A Node.js container suddenly runs curl or python — tools not expected in the app.

Why it matters: Could be signs of malware or misbehavior.

🔁 7. Monitoring Kubernetes API Access

Use Case: Falco can alert on suspicious kubectl commands or changes to cluster resources.

Example Detection:

> Creating a new privileged pod or deleting a namespace unexpectedly.

Why it matters: Helps catch insider threats or automation gone rogue.

📂 8. Reading Docker or Kubernetes Secrets

Use Case: Alert when a process tries to read files in /var/run/secrets/ (Kubernetes secrets).

Why it matters: Secrets should only be accessed by intended applications — this could be a sign of compromise.

📉 9. Unexpected Volume Mounts

Use Case: Detect if a container mounts host directories like /proc, /root, /etc.

Why it matters: Mounting host paths could let containers escape their isolation.

🧪 10. CI/CD Runtime Security

Use Case: Add Falco to CI/CD pipelines (like Jenkins or GitHub Actions) to detect runtime risks during testing.

Why it matters: Helps test security posture before moving to production.

🛡️ Detailed Falco Use Cases

---

1. Detect Shells Inside Containers

🔍 What happens:

A user (developer, attacker, or automated tool) opens a shell (, sh) in a running container.

🎯 Why it matters:

Containers should run applications, not be accessed interactively. Shell access could lead to data leaks, container escapes, or tampering.

🛑 Falco detects:

- Execution of shell commands inside containers
- Tools like , sh, zsh, ksh, dash

📘 Example rule:

yaml

```
- rule: Terminal shell in container
  desc: A shell was run inside a container
  condition: container and proc.name in (, sh, zsh)
  output: "Shell detected inside container (user=%user.name command=%proc.cmdline)"
  priority: WARNING
```

---

2. Unauthorized File Access (e.g., /etc/shadow or /etc/passwd)

🔍 **What happens:**

Someone tries to read/write sensitive files — these store user credentials and system-level info.

🎯 **Why it matters:**

Accessing these files is typically linked to privilege escalation or credential theft.

🛑 **Falco detects:**

- Write attempts to /etc/passwd, /etc/shadow
- Reads of /run/secrets/kubernetes.io/

📘 **Example rule:**
yaml

```
- rule: Write below etc
  desc: Detect writing to files under /etc
  condition: open.write and fd.name startswith /etc
  output: "File below /etc opened for writing (user=%user.name
command=%proc.cmdline)"
  priority: CRITICAL
```

---

3. Unexpected Outbound Network Connections

🔍 **What happens:**

A container suddenly starts sending traffic to an external IP or a port it normally wouldn’t use.

🎯 **Why it matters:**

Could indicate malware is trying to connect to a command-and-control server (C2).

🛑 **Falco detects:**

- Outbound traffic to blacklisted IPs
- Use of non-standard ports like 4444, 6666, or 1337

📘 **Example rule:**

yaml

```
- rule: Unexpected outbound connection
  desc: Container is making an outbound network connection
  condition: container and evt.type=connect and fd.sip != <expected_internal_IPs>
  output: "Suspicious network activity (container=%container.name ip=%fd.sip)"
  priority: WARNING
```

---

4. Changes to Binaries or Config Files

🔍 **What happens:**

Someone changes a binary (e.g., nginx) or modifies a config file.

🎯 **Why it matters:**

These changes can allow attackers to introduce backdoors, disable security, or alter app behavior.

🛑 **Falco detects:**

- Write operations in /usr/bin, /bin, or /etc
- Unexpected creation of .conf or .yaml files

📘 **Example rule:**

yaml

```
- rule: Write binary directory
  desc: A binary directory was written to
  condition: open.write and fd.name startswith (/usr/bin or /bin)
  output: "Binary directory modified (file=%fd.name user=%user.name)"
  priority: ERROR
```

---

5. Privilege Escalation Attempts

🔍 **What happens:**

Processes try to elevate their privileges — by using sudo, changing user IDs, or manipulating security contexts.

🎯 **Why it matters:**

Privilege escalation can turn a container compromise into a node-level threat.

🛑 **Falco detects:**

- sudo execution
- setuid, setgid system calls
- Containers running as root

📘 **Example rule:**

yaml

```
- rule: Privileged container started
  desc: Container started with privileged access
  condition: container and container.privileged=true
  output: "Privileged container detected (container=%container.name)"
  priority: CRITICAL
```

---

6. Suspicious Process Execution

🔍 **What happens:**

A container spawns unusual processes like nmap, curl, telnet, or python.

🎯 **Why it matters:**

Legitimate app containers rarely need these tools. Their appearance may signal scanning or exfiltration activity.

🛑 **Falco detects:**

- Processes running curl, wget, scp, ftp, python, nmap

📘 **Example rule:**

yaml

```
- rule: Unexpected process in container
  desc: Detect tools often used for recon or exfiltration
  condition: container and proc.name in (nmap, curl, wget, scp, python)
  output: "Suspicious process started in container (proc=%proc.name
command=%proc.cmdline)"
  priority: WARNING
```

---

7. Abnormal Kubernetes Activity

🔍 **What happens:**

Someone uses kubectl or API to create/delete namespaces, deploy privileged pods, etc.

🎯 **Why it matters:**

Unexpected Kubernetes control plane activity may indicate insider misuse or breach.

🛑 **Falco detects:**

- kubectl exec, kubectl delete, kubectl create commands
- Changes to role bindings or service accounts

📘 **Example rule:**

yaml

```
- rule: Kubernetes exec into container
  desc: Detect kubectl exec
  condition: proc.name = kubectl and proc.args contains exec
  output: "kubectl exec detected (user=%user.name command=%proc.cmdline)"
  priority: WARNING
```

---

8. Secrets Access

🔍 **What happens:**

Apps or users try to read files like Kubernetes secrets or environment files.

🎯 **Why it matters:**

Secrets should only be accessible to the pod that owns them. Unexpected access can leak credentials or tokens.

🛑 **Falco detects:**

- Reads in /var/run/secrets/
- Reads of .env, .git-credentials

📘 **Example rule:**

yaml

```
- rule: Read Kubernetes secrets
  desc: Access to mounted Kubernetes secrets
  condition: open.read and fd.name startswith /var/run/secrets/
  output: "Secret access attempt (file=%fd.name user=%user.name)"
  priority: CRITICAL
```

---

9. Host Filesystem Access by Containers

🔍 **What happens:**

A container mounts host paths like /proc, /etc, or /root.

🎯 **Why it matters:**

Host path mounts can let containers escape isolation and tamper with the host OS.

🛑 **Falco detects:**

- Mounting of host files
- Writes to host paths by containers

📘 **Example rule:**

yaml

```
- rule: Container accessing host files
  desc: Container tries to access host filesystem
  condition: container and fd.name startswith (/host/etc or /host/proc)
  output: "Container accessing host files (file=%fd.name container=%container.name)"
  priority: CRITICAL
```

---

10. Runtime Security in CI/CD Pipelines

🔍 **What happens:**

Use Falco in your staging environment to test what kind of runtime behavior your app generates.

🎯 **Why it matters:**

Better to catch misconfigurations, over-permissive containers, or unknown behaviors **before** production.

🛑 **Falco detects:**

- Test containers that behave abnormally
- Security regression in newly pushed containers

Integrate alerts into Slack, email, or SIEM for proactive responses.

Namespace & Resource Management

Kustomize – Customize Kubernetes YAMLs without forking.

Kustomize: Customizing Kubernetes YAMLs without Forking

Kustomize is a tool used in Kubernetes to modify configuration files (YAMLs) without changing the original files. In Kubernetes, YAML files define the configurations for things like pods, deployments, and services. When you have different environments (like development, staging, and production), you often need to modify these files to suit each environment.

Instead of copying and modifying the same YAML file for each environment (which can be hard to manage), Kustomize helps you make these customizations in a cleaner way. It does this using a concept called **overlays**.

Key Concepts of Kustomize:

1. **Base Files**: These are your original YAML files that define the common settings for your Kubernetes objects, like a deployment or a service.

2. **Overlays**: These are environment-specific customizations that are applied on top of the base files. For example, you could have different overlays for development, staging, and production environments.

3. **No Need for Forking**: You don't have to copy or fork the base YAML files for each environment. Kustomize allows you to maintain one set of base files, and apply different overlays (modifications) for different environments.

Why Use Kustomize?

- **Simplifies management**: You only need to manage one version of your YAML files, while applying environment-specific changes through overlays.

- **Keeps your YAMLs DRY**: Instead of duplicating YAML files, you keep them DRY (Don’t Repeat Yourself) by reusing the same base files.

- **Easily integrated with kubectl**: Kustomize is built into kubectl, the Kubernetes command-line tool, making it easy to use.

So, instead of manually changing YAML files for each environment, Kustomize streamlines the process and helps you customize your Kubernetes configurations efficiently.

Installation on Linux:

1. **Download the latest release**:

You can download Kustomize from its GitHub releases page. First, get the latest release URL:

```
curl -s https://api.github.com/repos/kubernetes-sigs/kustomize/releases/latest | jq -r .assets[0].browser_download_url
```

2. **Install Kustomize**:

Alternatively, you can download and install it directly from the terminal using curl:

```
curl -sLO https://github.com/kubernetes-sigs/kustomize/releases/download/v5.0.0/kustomize_v5.0.0_linux_amd64.tar.gz
```

Extract the downloaded file:

```
tar -zxvf kustomize_v5.0.0_linux_amd64.tar.gz
```

Move the binary to a directory included in your PATH:

```
sudo mv kustomize /usr/local/bin
```

3. **Verify Installation**:

Check that Kustomize is correctly installed by running:

```
kustomize version
```

Installation on Mac:

If you're using Homebrew, you can install Kustomize with a single command:

```
brew install kustomize
```

Alternatively, you can manually install it:

Download the latest release:

```
curl -sLO https://github.com/kubernetes-sigs/kustomize/releases/download/v5.0.0/kustomize_v5.0.0_darwin_amd64.tar.gz
```

Extract the archive:

```
tar -zxvf kustomize_v5.0.0_darwin_amd64.tar.gz
```

Move the binary to /usr/local/bin:

```
sudo mv kustomize /usr/local/bin
```

Verify the installation:

```
kustomize version
```

Installation on Windows:

For Windows, you can install Kustomize using the Windows Subsystem for Linux (WSL) or download the binary directly.

Using choco (if you have Chocolatey installed):

```
choco install kustomize
```

1. **Manual Installation**:

Download the latest Windows release from GitHub:

```
curl -LO
https://github.com/kubernetes-sigs/kustomize/releases/download/v5.0.0/kustomize_v5.0.0_windows_amd64.tar.gz
```

Extract the tar file:

```
tar -zxvf kustomize_v5.0.0_windows_amd64.tar.gz
```

- Move the kustomize.exe to a directory in your PATH for easy access.

Verify Installation:

After installation, check the version of Kustomize to ensure it's working:

```
kustomize version
```

This should display the installed version of Kustomize. If you see the version details, the installation was successful!

Project Overview: Kubernetes Application with Multiple Environments Using Kustomize

In this project, you'll create a simple Kubernetes application and then customize it for multiple environments (development, staging, and production) using Kustomize.

---

1. Create the Base Kubernetes Configuration

Create a directory structure like this:

```
my-kustomize-project/
├── base/
│   ├── deployment.yaml
│   ├── service.yaml
│   └── kustomization.yaml
├── overlays/
│   ├── development/
│   │   ├── kustomization.yaml
│   │   ├── configmap.yaml
│   ├── staging/
│   │   ├── kustomization.yaml
│   │   ├── configmap.yaml
│   └── production/
│      ├── kustomization.yaml
│      ├── configmap.yaml
└── README.md
```

2. Define the Base Kubernetes Files

Inside the base/ directory, create the main configuration files that are shared across all environments.

deployment.yaml (Base Deployment)

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: my-app-image:latest
          ports:
            - containerPort: 80
```

service.yaml (Base Service)

yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

kustomization.yaml (Base Kustomization)

yaml

```
resources:
  - deployment.yaml
  - service.yaml
```

3. Create Environment-Specific Overlays

Now, create environment-specific customizations for each environment (development, staging, and production) using overlays.

Development Overlay:

kustomization.yaml (Development)

yaml

```
resources:
  - ../../base

configMapGenerator:
  - name: app-config
    literals:
      - ENV=development
```

configmap.yaml (Development)

yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  ENV: development
```

Staging Overlay:

kustomization.yaml (Staging)

yaml

```
resources:
  - ../../base

configMapGenerator:
  - name: app-config
    literals:
      - ENV=staging
```

configmap.yaml (Staging)

yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  ENV: staging
```

Production Overlay:

kustomization.yaml (Production)

yaml

```
resources:
  - ../../base

configMapGenerator:
  - name: app-config
    literals:
      - ENV=production
```

configmap.yaml (Production)

yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  ENV: production
```

4. Apply the Overlays Using Kustomize

After setting up the base and overlay configurations, you can apply the environment-specific configurations using Kustomize.

For example:

To deploy to Development:

```
kubectl apply -k ./overlays/development
```

To deploy to Staging:

```
kubectl apply -k ./overlays/staging
```

To deploy to Production:

```
kubectl apply -k ./overlays/production
```

5. Verify Your Deployments

You can check the deployed resources in your Kubernetes cluster:

```
kubectl get deployments
kubectl get services
kubectl get configmaps
```

6. Conclusion

This project demonstrates how Kustomize allows you to manage different configurations for different environments without duplicating YAML files. By using a **base** set of YAML files and **overlays**, you can customize your Kubernetes resources for each environment (development, staging, and production).

Project Extension Ideas:

- **Add Secrets Management**: Use Kustomize to manage environment-specific secrets.

- **Versioning Configurations**: Use Kustomize in conjunction with version control to manage different configurations across versions of your app.

- **Use Helm with Kustomize**: Combine Kustomize with Helm charts for even more powerful configuration management.

Project Overview: Kubernetes Application with Multiple Environments Using Kustomize

In this project, you'll create a simple Kubernetes application and then customize it for multiple environments (development, staging, and production) using Kustomize.

---

1. Create the Base Kubernetes Configuration

Create a directory structure like this:

```
my-kustomize-project/
├── base/
│   ├── deployment.yaml
│   ├── service.yaml
│   └── kustomization.yaml
├── overlays/
│   ├── development/
│   │   ├── kustomization.yaml
│   │   ├── configmap.yaml
│   ├── staging/
│   │   ├── kustomization.yaml
│   │   ├── configmap.yaml
│   └── production/
│      ├── kustomization.yaml
│      ├── configmap.yaml
└── README.md
```

2. Define the Base Kubernetes Files

Inside the base/ directory, create the main configuration files that are shared across all environments.

deployment.yaml (Base Deployment)

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: my-app-image:latest
          ports:
            - containerPort: 80
```

service.yaml (Base Service)

yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

kustomization.yaml (Base Kustomization)

yaml

```
resources:
  - deployment.yaml
  - service.yaml
```

3. Create Environment-Specific Overlays

Now, create environment-specific customizations for each environment (development, staging, and production) using overlays.

Development Overlay:

kustomization.yaml (Development)

yaml

```
resources:
  - ../../base

configMapGenerator:
  - name: app-config
    literals:
      - ENV=development
```

configmap.yaml (Development)

yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  ENV: development
```

Staging Overlay:

kustomization.yaml (Staging)

yaml

```yaml
resources:
  - ../../base

configMapGenerator:
  - name: app-config
    literals:
      - ENV=staging
```

configmap.yaml (Staging)

yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  ENV: staging
```

Production Overlay:

kustomization.yaml (Production)

yaml

```
resources:
  - ../../base

configMapGenerator:
  - name: app-config
    literals:
      - ENV=production
```

configmap.yaml (Production)

yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  ENV: production
```

4. Apply the Overlays Using Kustomize

After setting up the base and overlay configurations, you can apply the environment-specific configurations using Kustomize.

For example:

To deploy to Development:

```
kubectl apply -k ./overlays/development
```

To deploy to Staging:

```
kubectl apply -k ./overlays/staging
```

To deploy to Production:

```
kubectl apply -k ./overlays/production
```

5. Verify Your Deployments

You can check the deployed resources in your Kubernetes cluster:

```
kubectl get deployments
kubectl get services
kubectl get configmaps
```

6. Conclusion

This project demonstrates how Kustomize allows you to manage different configurations for different environments without duplicating YAML files. By using a **base** set of YAML files and **overlays**, you can customize your Kubernetes resources for each environment (development, staging, and production).

Project Extension Ideas:

- **Add Secrets Management**: Use Kustomize to manage environment-specific secrets.
- **Versioning Configurations**: Use Kustomize in conjunction with version control to manage different configurations across versions of your app.
- **Use Helm with Kustomize**: Combine Kustomize with Helm charts for even more powerful configuration management.

Use cases for Kustomize that will help a new learner understand how it can be applied in different scenarios. These use cases highlight how Kustomize makes managing Kubernetes resources more efficient, especially in multi-environment deployments.

1. Managing Different Environments (Development, Staging, Production)

Use Case: Customizing Configurations for Multiple Environments

In Kubernetes, it's common to have the same application deployed in multiple environments (e.g., development, staging, and production), but with different configurations like resource limits, replica counts, and environment variables.

How Kustomize Helps:

- **Base Resources**: Define a base set of Kubernetes YAML files (e.g., Deployment, Service) that apply to all environments.

- **Overlays**: Use Kustomize overlays to customize configurations for each environment.

    - In the **development** environment, you might want fewer replicas (e.g., 1 replica).

    - In **production**, you might want more replicas (e.g., 3 replicas) and higher resource limits (e.g., CPU and memory).

Instead of manually editing the YAML files for each environment, you can create overlays like development/, staging/, and production/ with environment-specific customizations, while still maintaining a common base.

---

2. Avoiding Duplication of YAML Files

Use Case: DRY (Don’t Repeat Yourself) Principle for Kubernetes YAMLs

You might need to deploy the same application across different clusters or environments. Without Kustomize, you'd have to copy the same YAML files and edit them each time, leading to code duplication and possible mistakes in synchronization.

How Kustomize Helps:

- With Kustomize, you only need to maintain **one base set of YAML files** for your application. Any changes needed for a specific environment (like changing the image version, replica count, or environment variables) can be applied through **overlays**.

- This means fewer files to manage and a cleaner configuration.

---

3. Customizing Application Settings Using ConfigMaps and Secrets

Use Case: Managing Application Configuration via ConfigMaps and Secrets

Different environments (e.g., development vs. production) often require different application settings. For example, you might want to use a mock database in development and a real database in production. The database URL, API keys, or other environment-specific settings can be stored in **ConfigMaps** or **Secrets**.

How Kustomize Helps:

- Kustomize can manage environment-specific **ConfigMaps** and **Secrets** in overlays.

- In the **development** environment, the configmap.yaml might include a mock database URL, whereas in **production**, the database URL will be different.

Instead of modifying the same ConfigMap or Secret file manually for each environment, Kustomize allows you to define these differences in overlays while keeping the base ConfigMap file the same.

---

4. Managing Versioned Deployments

Use Case: Version Control for Kubernetes Configurations

When you release new versions of your application, you often need to update the Kubernetes configurations (e.g., updating the container image version or changing resource allocations).

How Kustomize Helps:

- Kustomize allows you to use **overlays** to handle different application versions.
- You can specify the application's version (for example, my-app:v1.0.0) in the base configuration, and for each release, use overlays to change version-specific configurations without altering the base YAML files.
- By maintaining versioned overlays, you can easily deploy the same application version across different clusters without modifying the original YAML files.

---

5. Multi-Cluster Management

Use Case: Deploying to Multiple Clusters with Different Configurations

You might need to deploy your application to multiple Kubernetes clusters (e.g., one in AWS, another in GCP, and another on-premises). Each cluster may have different resource configurations, but the application itself is the same.

How Kustomize Helps:

- You can create **overlays** for each cluster.
    - The base configuration can stay the same, but each cluster overlay can specify differences in settings like CPU, memory, and replica counts that are specific to each environment.
- This allows you to manage Kubernetes resources in different clusters without duplicating YAML files, making the management more efficient and less error-prone.

---

6. Simplifying Helm Chart Customization

Use Case: Customizing Helm Charts with Kustomize

When deploying applications using Helm, you often need to customize settings like replica counts, image versions, or configurations specific to the environment.

How Kustomize Helps:

- Kustomize can be used to customize **Helm charts** by layering on top of the existing Helm-generated YAML files.
- You can use Kustomize to patch Helm chart values for specific environments or Kubernetes clusters, without directly modifying the Helm chart itself.

This way, you can leverage Helm for its package management features while using Kustomize to manage environment-specific configurations.

---

7. Managing Namespaces and Resource Limits

Use Case: Configuring Kubernetes Namespaces and Resource Quotas

Kubernetes namespaces allow you to organize resources logically, and resource quotas control the maximum amount of resources a namespace can consume.

How Kustomize Helps:

- You can create a **base namespace configuration** and then create overlays for different environments that apply specific resource quotas, limits, or even specific namespace names.

- For example, in the **staging** environment, you may apply a stricter resource quota (e.g., memory and CPU limits) than in **development**.

---

8. Handling Continuous Integration/Continuous Deployment (CI/CD)

Use Case: Automating Kubernetes Deployment with CI/CD Pipelines

In a CI/CD pipeline, you might want to deploy the same application in different stages (e.g., testing, staging, and production) with slight modifications (e.g., image tags, resource allocation).

How Kustomize Helps:

- You can integrate **Kustomize** into your CI/CD pipeline (e.g., Jenkins, GitLab CI) to automatically apply environment-specific configurations using overlays before deploying to different clusters.

- The same set of YAML files can be reused across environments, reducing duplication and making the pipeline simpler to maintain.

---

9. Managing Horizontal Pod Autoscaling

Use Case: Horizontal Pod Autoscaling Across Environments

When deploying applications with **Horizontal Pod Autoscalers (HPA)**, different environments may need different configurations for the autoscaler, like different CPU/memory thresholds or scaling policies.

How Kustomize Helps:

- Use **overlays** to manage environment-specific configurations for HPA.
- For **production**, you might want stricter scaling policies, whereas in **development**, the autoscaler might have looser scaling policies to save resources.

This allows you to manage scaling policies separately for each environment without modifying the base configuration.

Conclusion

These use cases show how Kustomize can make managing Kubernetes configurations easier, especially when working with multiple environments or when you want to avoid code duplication. Whether you're managing different environments (development, staging, production), customizing application configurations, or deploying across multiple clusters, Kustomize simplifies the process by enabling environment-specific customizations while keeping the base configuration intact.

Helm – Package manager for Kubernetes (charts).

Helm is a tool that helps you easily install and manage applications on Kubernetes. It works like a package manager (similar to how you use npm for JavaScript), but for Kubernetes. Helm uses something called "charts," which are like pre-configured templates that bundle all the necessary settings and resources needed to deploy an application on Kubernetes.

Instead of writing long YAML files for every application, Helm allows you to use these charts to install, update, or manage applications with a single command. This makes it easier to manage complex applications, handle dependencies (like making sure certain parts of your app are installed in the right order), and track different versions of your application.

In short, Helm simplifies managing applications on Kubernetes, especially when there are multiple parts and configurations involved.

Step 1: Install Helm

You can install Helm on Linux, macOS, or Windows. Below are the instructions for each.

For Linux:
Open a terminal and run the following command to download the Helm installation script:

```
curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 |
```

This command will download and install the latest version of Helm.

Verify the installation by checking the Helm version:

```
helm version
```

For macOS:
If you use **Homebrew**, you can install Helm by running:

```
brew install helm
```

Verify the installation by checking the Helm version:

```
helm version
```

For Windows:

You can install Helm using **Chocolatey**:

```
choco install kubernetes-helm
```

> Alternatively, you can download the Helm binary from the official Helm releases page and add it to your system's PATH.

Verify the installation by checking the Helm version:

```
helm version
```

Step 2: Set up Helm (Optional)

Once Helm is installed, you'll need to configure it to work with a Kubernetes cluster.

Add a Helm repository (for example, the official Helm charts repository):

```
helm repo add stable https://charts.helm.sh/stable
```

Update the Helm repository to fetch the latest charts:

```
helm repo update
```

Step 3: Verify your Setup

To ensure everything is working correctly, you can run a simple Helm command to check:

```
helm list
```

This should show a list of Helm releases (which would be empty if you haven't installed anything yet).

Now you're ready to use Helm to manage your Kubernetes applications!

Project: Deploy a Web Application with Helm on Kubernetes

In this project, you will use Helm to deploy a simple web application (e.g., a Node.js app) on a Kubernetes cluster. This will help you understand how to package an application using Helm charts and deploy it in a standardized way.

Steps for the Project:

Step 1: Set up Kubernetes Cluster

If you don't already have a Kubernetes cluster, you can set one up using **Minikube** (for local development) or use a cloud provider like AWS, GCP, or Azure.

To create a Kubernetes cluster with **Minikube**:

1. Install Minikube and kubectl (if you haven't already).

Start a Minikube cluster:

```
minikube start
```

Step 2: Install Helm

Follow the installation instructions provided earlier to install **Helm** on your system.

Step 3: Create a Helm Chart

Create a new Helm chart for your web application (e.g., a simple Node.js app):
helm create my-nodejs-app

This will create a directory structure with the necessary files.

1. **Customize the Chart**:

In the values.yaml file, define your application settings like Docker image, replicas, ports, etc. Example:

```
 yaml

image:
  repository: my-nodejs-app
  tag: latest
service:
  port: 80
```

  - Customize templates/deployment.yaml and templates/service.yaml if needed for your app's requirements.

Step 4: Build and Push Docker Image
Build the Docker image for your Node.js app (if you don't have a Docker image already). Example Dockerfile for a Node.js app:

```
 dockerfile

FROM node:14

WORKDIR /usr/src/app
COPY . .

RUN npm install
```

```
CMD ["node", "app.js"]

EXPOSE 8080
```

Build the image and push it to a container registry (like Docker Hub or a private registry):

```
docker build -t myusername/my-nodejs-app .
docker push myusername/my-nodejs-app
```

Step 5: Deploy the Application with Helm

Install the Helm Chart to deploy your web app on Kubernetes:

```
helm install my-nodejs-app ./my-nodejs-app
```

This will create Kubernetes resources like Pods, Services, and Deployments based on your Helm chart.

Step 6: Verify Deployment

Check if your application is running in the Kubernetes cluster:

```
kubectl get pods
kubectl get services
```

You can also access your application by getting the external IP of the service:

```
kubectl get svc my-nodejs-app
```

If using Minikube, you can use:

```
minikube service my-nodejs-app
```

Step 7: Upgrade and Rollback

To practice upgrading your application, modify your values.yaml file (e.g., change the Docker image version) and upgrade the release:

Modify the values.yaml to update the image tag:

yaml

```
image:
  tag: v2
```

Upgrade the release:

```
helm upgrade my-nodejs-app ./my-nodejs-app
```

If needed, you can also rollback to a previous version:

```
helm rollback my-nodejs-app 1
```

Step 8: Clean Up

Once you're done with the project, you can uninstall the Helm release:

```
helm uninstall my-nodejs-app
```

Project: Deploying a Node.js Web Application Using Helm on Kubernetes

---

Step 1: Set up Kubernetes Cluster

If you don’t have Kubernetes set up already, you can use **Minikube** for local development or any cloud Kubernetes provider.

For Minikube:

Install Minikube (if you don't have it installed):

```
brew install minikube  # For macOS
```

```
sudo apt install minikube  # For Linux
```

Start Minikube:

```
minikube start
```

Install kubectl (Kubernetes CLI) if you don’t have it installed:

```
brew install kubectl  # For macOS
```

```
sudo apt install kubectl  # For Linux
```

Check if the cluster is running:

```
kubectl cluster-info
```

Step 2: Install Helm

Install **Helm** using the appropriate method for your OS (follow the instructions provided earlier).

Step 3: Create a Node.js Web Application

Create the Node.js app

Create a new directory for your project:

```
mkdir node-helm-app
```

```
cd node-helm-app
```

Initialize a Node.js app:

```
npm init -y
```

Install Express (a lightweight web framework for Node.js):

```
npm install express
```

Create app.js as the entry point for the application:

js

```
// app.js
const express = require('express');
const app = express();
const port = 8080;

app.get('/', (req, res) => {
  res.send('Hello, Kubernetes with Helm!');
});

app.listen(port, () => {
  console.log(`App listening at http://localhost:${port}`);
});
```

Create a Dockerfile to containerize the application:

dockerfile

```
# Dockerfile
FROM node:14


WORKDIR /app
COPY . .


RUN npm install


EXPOSE 8080


CMD ["node", "app.js"]
```

1.

Build and test the Docker image:

```
docker build -t node-helm-app .
docker run -p 8080:8080 node-helm-app
```

2. Visit http://localhost:8080 to ensure the app is running.

Push the Docker image to Docker Hub:

```
docker login

docker tag node-helm-app <your-dockerhub-username>/node-helm-app:v1

docker push <your-dockerhub-username>/node-helm-app:v1
```

Step 4: Create Helm Chart

Create a new Helm chart:

```
helm create nodejs-app
```

1. **Modify the values.yaml** in the chart to define the image details:
   - Open nodejs-app/values.yaml.
   - Set the following values (replace <your-dockerhub-username>):

```
yaml

image:

  repository: <your-dockerhub-username>/node-helm-app

  tag: v1

service:

  port: 80
```

Update the Deployment template (nodejs-app/templates/deployment.yaml) to ensure it works with the Docker image:

yaml

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: {{ include "nodejs-app.fullname" . }}

  labels:

    {{- include "nodejs-app.labels" . | nindent 4 }}

spec:

  replicas: 1

  selector:

    matchLabels:

      app.kubernetes.io/name: {{ include "nodejs-app.name" . }}

  template:

    metadata:

      labels:

        app.kubernetes.io/name: {{ include "nodejs-app.name" . }}

    spec:

      containers:

        - name: nodejs-app
```

```
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          ports:
            - containerPort: 8080
```

Update the Service template (nodejs-app/templates/service.yaml):

yaml

```
apiVersion: v1
kind: Service
metadata:
  name: {{ include "nodejs-app.fullname" . }}
spec:
  selector:
    app.kubernetes.io/name: {{ include "nodejs-app.name" . }}
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: LoadBalancer
```

Step 5: Deploy the Application Using Helm

Package the Helm chart:

```
helm package nodejs-app
```

Deploy the application to your Kubernetes cluster:

```
 helm install nodejs-app ./nodejs-app
```

Check if your application is running:

```
kubectl get pods
```

```
kubectl get svc
```

If you're using Minikube, you can expose the service:

```
minikube service nodejs-app
```

Visit the exposed URL to see your application live.

Step 6: Upgrade and Rollback

Update the application:
Modify values.yaml to change the image tag (e.g., v2).

Run the following to upgrade the app:

```
helm upgrade nodejs-app ./nodejs-app
```

Rollback to a previous version:

```
helm rollback nodejs-app 1
```

Step 7: Clean Up

Uninstall the Helm release:

```
helm uninstall nodejs-app
```

Stop Minikube (if you're using it):

```
minikube stop
```

Additional Enhancements:

1. **Use Helm Secrets** for managing sensitive information such as API keys or credentials.
2. **Create a multi-container deployment** (e.g., a backend service with a database).
3. **Monitor with Prometheus** or **Grafana** for better observability.
4. **Helm Charts in a GitOps workflow**: Use ArgoCD or Flux to automate Helm chart deployments.

Conclusion:

This project walks you through creating a simple Node.js app, containerizing it with Docker, deploying it on Kubernetes using Helm, and managing updates and rollbacks with Helm commands. You can expand this project further by adding more complexity (e.g., adding a database, CI/CD, monitoring).

---

1. Deploying Web Applications

Scenario: You need to deploy a multi-component web application (e.g., a front-end React app and a back-end Node.js API) across different environments (development, staging, production).

Use Case:

- **Helm helps** you create a reusable and configurable package for the app (a Helm chart).
- You can use **Helm values** to define environment-specific configurations (like database URLs, API keys, etc.).
- With **Helm templates**, you can generate Kubernetes YAML configurations that are used to create deployments, services, and other resources in a repeatable and automated way.

Benefit: Simplifies the deployment and maintenance of multi-service applications, enabling a smooth rollout in multiple environments with consistent configurations.

2. Managing Microservices Architectures

Scenario: You are running multiple microservices (e.g., payment, user management, and notification services) on Kubernetes. Each microservice has its own set of configurations and dependencies.

Use Case:

- With **Helm**, you can create a Helm chart for each microservice, encapsulating its configuration and Kubernetes resources.
- You can also manage **dependencies** between microservices (e.g., a payment service may depend on a user service).
- You can define each microservice's version, deploy them independently, and update them without affecting other services.

Benefit: Helm helps manage complexity in microservices architectures by enabling repeatable deployments, updates, and rollbacks.

3. Automating Kubernetes Application Updates

Scenario: You need to update your application with bug fixes or new features, and you want the update process to be consistent and repeatable.

Use Case:

- **Helm** allows you to package and upgrade applications with ease. Once the Helm chart is defined, you can upgrade your application by simply changing version numbers or configuration values and running a single command (helm upgrade).
- You can use Helm to apply **rolling updates**, ensuring minimal downtime by gradually replacing older versions with new ones.

Benefit: Helm automates and simplifies application updates, reducing the risk of errors during deployment and ensuring that your application is always running the desired version.

4. Managing Application Dependencies

Scenario: Your application requires several services such as databases, caches, and message queues (e.g., MySQL, Redis, Kafka) to work.

Use Case:

- **Helm charts** allow you to define not just the application itself, but also its dependencies (such as databases, caches, or other microservices).
- Helm can pull in pre-configured charts from **Helm repositories** (e.g., Bitnami, official Kubernetes charts) for these services, allowing you to manage their versions and configurations as part of your overall application.
- For example, if you're deploying a WordPress site, you can use Helm to install both WordPress and MySQL as dependencies in a single Helm release.

Benefit: Helm makes it easy to manage complex, multi-component applications and their dependencies, reducing manual configuration and setup.

5. Configuring Environment-Specific Settings

Scenario: You are deploying an application that needs different configurations for development, staging, and production environments.

Use Case:

- **Helm values files** allow you to manage environment-specific configurations. For instance, you can have different values files (values-dev.yaml, values-prod.yaml) for development and production environments.
- When deploying, you can pass the appropriate values file to Helm to adjust settings like resource limits, replica counts, or API keys, ensuring that each

environment is configured correctly.

Benefit: Helm simplifies managing multiple environments by allowing you to specify different configurations per environment while using the same chart.

6. Managing Legacy Applications

Scenario: You are migrating a legacy application to Kubernetes, and you need a consistent way to deploy and manage it in the cluster.

Use Case:

- You can create a **Helm chart** for the legacy application, encapsulating all the Kubernetes resources it needs (e.g., Deployments, Services, ConfigMaps).
- With Helm, you can easily version and upgrade the application without manually managing YAML files for each component.
- Helm enables you to adopt **CI/CD practices**, making it easier to deploy and manage the legacy application in Kubernetes over time.

Benefit: Helm helps streamline the migration of legacy applications to Kubernetes by providing a standardized way to package, deploy, and manage them.

7. Managing Configurations and Secrets

Scenario: Your application needs to access sensitive configurations (e.g., database credentials, API keys) that should not be hardcoded in the Kubernetes manifests.

Use Case:

- **Helm Secrets** integrates with **Helm** to securely manage sensitive configurations. You can store your secrets in **encrypted files** and use Helm to decrypt and inject them into the application during deployment.

- Helm's **values** can include references to external secret management systems (e.g., AWS Secrets Manager, HashiCorp Vault), keeping sensitive data out of version-controlled charts.

Benefit: Helm provides a secure way to manage and inject sensitive configurations, helping you adhere to best practices in security and configuration management.

8. Self-Hosted SaaS Applications

Scenario: You need to provide a self-hosted version of your SaaS application to customers. Each customer may have different configurations or requirements (e.g., custom domain names, resource limits).

Use Case:

- You can create a **Helm chart** that is highly configurable, allowing customers to deploy their own instances of your SaaS application with their preferred settings.

- Customers can set up their instances using Helm, specifying values like the number of replicas, database connections, and custom environment variables.

- You can provide Helm charts as a way to offer a **turnkey solution** for your customers, simplifying their deployment and management process.

Benefit: Helm allows you to provide customers with a straightforward, standardized way to deploy and manage your SaaS product in their own Kubernetes clusters.

9. Multi-Cluster Deployments

Scenario: You need to deploy the same application across multiple Kubernetes clusters, either for high availability, disaster recovery, or geographic distribution.

Use Case:

- **Helm** enables you to deploy the same chart to multiple clusters by defining different configurations for each cluster in values files.
- You can also use **Helm charts** to deploy to different cloud providers, enabling multi-cloud deployments for high availability or disaster recovery.

Benefit: Helm simplifies the process of managing applications in multi-cluster and multi-cloud environments by allowing consistent deployment practices across clusters.

10. Continuous Integration and Continuous Deployment (CI/CD)

Scenario: You need to automate the process of deploying applications to Kubernetes using a CI/CD pipeline.

Use Case:

- **Helm** integrates seamlessly with CI/CD tools like **Jenkins**, **GitLab CI**, **GitHub Actions**, and **CircleCI**.
- In a CI/CD pipeline, Helm can be used to deploy your application, upgrade it, or roll back to previous versions automatically as part of the deployment

process.

- You can use Helm in combination with **helm test** and **helm lint** to validate your Kubernetes resources during the CI/CD pipeline execution.

Benefit: Helm helps streamline your CI/CD workflows by allowing you to automate the deployment and management of applications, making it easy to implement continuous delivery.

Conclusion

Helm is a powerful tool for Kubernetes that simplifies deployment, versioning, and management of applications, and it is useful across a wide variety of use cases, including:

- Deploying web applications and microservices
- Managing environment-specific configurations
- Automating application updates
- Handling application dependencies
- Managing legacy applications and secrets
- Facilitating CI/CD pipelines

Using Helm allows you to reduce complexity, ensure consistency, and streamline application management on Kubernetes.

Skaffold – An open-source tool designed for continuous development in Kubernetes environments, automating the build, deployment, and testing of applications.

What is Skaffold? Skaffold is an open-source tool that helps developers automate the process of building, deploying, and testing applications in Kubernetes environments. It makes it easier to work with Kubernetes by handling repetitive tasks like:

- **Building Docker images** for your application
- **Deploying** these images to your Kubernetes cluster
- **Running tests** to ensure everything works properly

With Skaffold, you don’t need to manually run each of these tasks every time you make a change to your application. Skaffold handles them for you, so you can focus on writing your code and making improvements.

Why is Skaffold useful for new learners?

- **Speeds up Development**: Skaffold automates the repetitive tasks in your workflow, allowing you to see changes faster.
- **Works well with Kubernetes**: It integrates perfectly with Kubernetes, which is commonly used for managing and deploying applications in the cloud.
- **Supports Continuous Development**: With Skaffold, you can quickly reflect changes to your application on your Kubernetes cluster without having to rebuild everything manually every time.

In summary, Skaffold makes it easier to build, deploy, and test applications on Kubernetes by automating common tasks, which is especially helpful when you're developing and iterating quickly. It's a tool that integrates well into CI/CD pipelines, ensuring that the process remains smooth and efficient.

---

1. Installing Skaffold on macOS

You can install Skaffold using **Homebrew**, a package manager for macOS.

1. Open your terminal.

Run the following command:

```
brew install skaffold
```

2. Installing Skaffold on Linux

For Linux, you can download the latest Skaffold release from the official GitHub releases page or install it using a package manager.

Option 1: Using curl (recommended)

1. Open your terminal.

Run the following commands to install Skaffold using curl:

```
curl -Lo skaffold
https://storage.googleapis.com/skaffold/releases/latest/skaffold-linux-amd64

chmod +x skaffold

sudo mv skaffold /usr/local/bin
```

Option 2: Using apt-get (for Ubuntu/Debian)

Add the Skaffold package repository to your system:

```
curl -fsSL
https://storage.googleapis.com/downloads.skaffold.dev/latest/skaffold-linux-amd64
> skaffold
```

```
sudo install skaffold /usr/local/bin/
```

3. Installing Skaffold on Windows

For Windows, you can install Skaffold via **Chocolatey** or download the Windows binary directly.

Option 1: Using Chocolatey

1. Open your PowerShell as Administrator.

Run the following command:

```
choco install skaffold
```

Option 2: Using the direct binary

1. Download the latest release from the Skaffold GitHub releases page.
2. Extract the binary and move it to a folder included in your system's PATH.

4. Verifying Installation

After installation, verify that Skaffold is installed correctly by running the following command in your terminal:

```
skaffold version
```

This should display the installed version of Skaffold.

Project Overview

We will create a **simple web application** using **Docker**, and we'll set up **Skaffold** to automate the build, deployment, and testing process in Kubernetes.

Steps to Create a Skaffold Project

1. Prerequisites

- **Kubernetes Cluster**: Ensure you have a running Kubernetes cluster. You can use **Minikube** or **kind** (Kubernetes in Docker) if you need a local cluster.

- **Docker**: Make sure Docker is installed and running.

- **Skaffold**: Ensure you have Skaffold installed (refer to the previous installation guide).

2. Initialize a New Project

Start by setting up a new project directory and creating a simple web application.

Create a new directory for your project:

```
mkdir skaffold-demo

cd skaffold-demo
```

Create a simple **Node.js** web app (or use your favorite language). Here's an example with Node.js.

index.js (Create this file in the project directory):

javascript

```
const express = require('express');
const app = express();
const port = 8080;

app.get('/', (req, res) => {
  res.send('Hello from Skaffold & Kubernetes!');
});

app.listen(port, () => {
  console.log(`App listening at http://localhost:${port}`);
});
```

package.json (Create this file):

json

```
{
  "name": "skaffold-demo",
  "version": "1.0.0",
  "main": "index.js",
  "dependencies": {
    "express": "^4.17.1"
  },
  "scripts": {
    "start": "node index.js"
  }
}
```

Install the dependencies:

```
npm install
```

3. Dockerize the Application

Next, we'll create a Dockerfile to containerize the application.

Dockerfile (Create this file):

Dockerfile

```
# Use official Node.js image as the base
FROM node:14
# Create and set the working directory
WORKDIR /app
# Copy the package.json and install dependencies
COPY package.json /app
```

```
RUN npm install

# Copy the rest of the app's code

COPY . /app

# Expose the port the app runs on

EXPOSE 8080

# Start the app

CMD ["npm", "start"]
```

4. Create Skaffold Configuration

Now we will set up the Skaffold configuration file to automate the build and deployment process.

skaffold.yaml (Create this file in the project root):

yaml

```yaml
apiVersion: skaffold/v2beta29

kind: Config

metadata:

  name: skaffold-demo

build:

  artifacts:

    - image: skaffold-demo

      context: .
      docker:
        dockerfile: Dockerfile
deploy:
  kubectl:
    manifests:
      - k8s/*
```

This file tells Skaffold to:

- Build the Docker image for the app.
- Deploy the application using **kubectl** with Kubernetes manifests.

5. Create Kubernetes Deployment Manifests

Create the Kubernetes manifest files that define the deployment of your application.

Create a directory called k8s for Kubernetes configurations:

```
mkdir k8s
```

k8s/deployment.yaml (Create this file):

yaml

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: skaffold-demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: skaffold-demo
  template:
    metadata:
      labels:
        app: skaffold-demo
    spec:
      containers:
        - name: skaffold-demo
          image: skaffold-demo
          ports:
            - containerPort: 8080
```

k8s/service.yaml (Create this file):

yaml

```
apiVersion: v1
```

```
kind: Service
metadata:
  name: skaffold-demo
spec:
  selector:
    app: skaffold-demo
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: LoadBalancer
```

6. Running Skaffold

Now you are ready to run your application using Skaffold.

1. Open your terminal in the project directory.

Run the following command to start the Skaffold development cycle:

```
skaffold dev
```

Skaffold will:

- Build the Docker image for the application.
- Deploy the application to your Kubernetes cluster.

   - Monitor the project for changes, automatically rebuilding and redeploying when files are modified.

2. You can view the logs in your terminal, and your application will be available in your Kubernetes cluster.

7. Access the Application

If you're using a local Kubernetes setup like Minikube or kind, you can access the service through the cluster's external IP. If it's a cloud-based Kubernetes cluster (e.g., GKE, AKS), it will be accessible via the external IP assigned to the LoadBalancer service.

To get the external IP (if it's Minikube or kind), you can use:

```
minikube service skaffold-demo
```

8. Testing and Iterating

With Skaffold running in **dev mode**, it will automatically detect changes to your code and re-build, re-deploy, and run your application in the Kubernetes cluster. This is great for fast feedback during development!

Summary

- You've created a simple **Node.js** application.
- You've Dockerized the application.
- You've set up Skaffold to automate the build, deployment, and testing in Kubernetes.
- You've created Kubernetes manifests for deployment and exposed your application via a service.

This project demonstrates how to use Skaffold for continuous development in Kubernetes, and it can easily be extended to more complex applications.

Use Cases

1. Continuous Development for Kubernetes Applications

Use Case: Automatically rebuild, deploy, and test applications during development in a Kubernetes environment.

Description:

- Developers often make frequent changes to their applications, especially in agile development environments. With Skaffold, developers can focus on writing code while Skaffold automatically handles the build, deployment, and testing.

- Every time there’s a code change, Skaffold will rebuild the Docker image, deploy the new version to the Kubernetes cluster, and optionally run tests or other checks.

Example:

- A developer works on a microservice-based app running in a Kubernetes cluster. As they make changes to the code, Skaffold ensures the application is rebuilt and redeployed in real time with minimal effort.

---

2. Integration with CI/CD Pipelines

Use Case: Automating the continuous integration and continuous deployment (CI/CD) of applications with Kubernetes clusters.

Description:

- Skaffold fits into a CI/CD pipeline and automates the build and deployment of applications.

- It integrates well with tools like **Jenkins**, **GitLab CI/CD**, **GitHub Actions**, or **CircleCI**, making it ideal for continuous integration and deployment workflows.

Example:

- A team uses GitLab CI/CD pipelines to deploy their application to a Kubernetes cluster. Skaffold is used to automate the build process within the pipeline, allowing the team to automatically deploy changes to their staging or production environment whenever new code is pushed to the repository.

---

3. Multi-Environment Development

Use Case: Easily manage deployments to multiple Kubernetes clusters (e.g., staging, production) with different configurations.

Description:

- When working with multiple environments, Skaffold allows developers to easily switch between environments and deploy the same codebase with different configurations (e.g., different databases or API endpoints).

- Skaffold supports multiple profiles, enabling different configurations for various environments like local, staging, or production.

Example:

- A developer can use a specific profile to deploy the app to a staging environment with mock data, and another profile for deploying to production with real data.

---

4. Local Development with Kubernetes (K8s)

Use Case: Simplifying local development with Kubernetes (via Minikube or kind) for rapid prototyping and testing.

Description:

- Developers can spin up a local Kubernetes environment (using **Minikube** or **kind**) and use Skaffold to manage the build, deploy, and test cycles.

- This is especially useful for testing new features or bug fixes without needing to wait for deployment in the cloud or on a remote Kubernetes cluster.

Example:

- A developer is working on a feature in a microservices architecture and uses **kind** (Kubernetes in Docker) to create a local Kubernetes cluster. Skaffold automates the build and deployment to the local cluster, enabling rapid testing of the new feature before it is merged into the main codebase.

---

5. Managing Kubernetes Deployments with Helm Charts

Use Case: Integrating Helm charts with Skaffold for managing application deployments on Kubernetes.

Description:

- Helm is a package manager for Kubernetes, allowing users to define, install, and upgrade complex Kubernetes applications.

- Skaffold can be used in conjunction with Helm to automate deployments of applications defined by Helm charts.

Example:

- A developer is building a complex application with multiple microservices. They use **Helm** charts to define the Kubernetes resources (deployments, services, etc.), and Skaffold automates the process of building the Docker images, deploying them with Helm, and managing the whole workflow.

---

6. Building and Deploying Multiple Services (Multi-Container Apps)

Use Case: Managing multi-container applications in Kubernetes, where each service is in a separate Docker container.

Description:

- Skaffold can handle multi-container applications by managing the build and deployment of each container in a Kubernetes pod.

- This is especially useful for microservices architectures where different services (databases, backend, frontend) need to be developed and deployed together.

Example:

- A team works on a microservices-based application with a frontend, backend, and database. Skaffold manages the continuous build, deployment, and testing of all services in the Kubernetes cluster, ensuring they work together in the correct configuration.

---

7. Managing Test and Dev Builds Separately

Use Case: Running different build and deploy processes for development and test environments.

Description:

- You can configure Skaffold to use different profiles for dev and test environments, with each profile having different configurations, image names, deployment strategies, or testing processes.
- This allows you to isolate your test and development environments to avoid unnecessary resource usage or complications when testing.

Example:

- In the dev profile, Skaffold deploys the application with a "debug" version of the image, while in the test profile, it deploys a fully-tested version to run integration or unit tests against the Kubernetes environment.

---

8. Continuous Delivery for Cloud-Native Applications

Use Case: Deploying cloud-native applications continuously with minimal downtime using Skaffold.

Description:

- Skaffold is great for teams building cloud-native applications that run on Kubernetes and require frequent, seamless updates with minimal downtime.

- It allows teams to implement blue-green or rolling deployments with Kubernetes, ensuring that updates to the application don't cause service disruptions.

Example:

- A company runs an e-commerce platform and needs to continuously deploy new features and updates with zero downtime. Skaffold is configured to handle blue-green deployments, ensuring that new versions of the application are deployed smoothly to Kubernetes clusters with no interruption to the live service.

---

9. Automating the Local Development Setup

Use Case: Simplifying the local setup for new developers joining a project by providing a one-click development environment.

Description:

- Skaffold can be configured to work seamlessly with developers' local machines, allowing them to quickly set up the entire development environment with minimal manual configuration.

- New developers only need to install Skaffold and Kubernetes (via **Minikube** or **kind**), and they can use a single command to start working on the project.

Example:

- A new developer joins a project and wants to quickly set up a local Kubernetes environment with all the necessary services running. With

Skaffold, the developer simply runs skaffold dev, and the entire local development setup is built and deployed automatically, with all dependencies in place.

---

10. Simplifying Deployment to Remote Kubernetes Clusters

Use Case: Deploying to remote Kubernetes clusters without manually managing Docker images and kubectl configurations.

Description:

- Skaffold automates the process of building Docker images, pushing them to a container registry, and deploying to a remote Kubernetes cluster.
- This simplifies the workflow for deploying applications from local environments to remote Kubernetes clusters, particularly in cloud environments like **Google Kubernetes Engine (GKE)** or **Amazon EKS**.

Example:

- A team is working on a project deployed to **Amazon EKS**. They use Skaffold to automatically build Docker images, push them to **Amazon ECR**, and deploy the updates to the **EKS** cluster without having to manually handle the kubectl commands or image pushing.

Conclusion

Skaffold is a powerful tool that streamlines Kubernetes workflows, making it an excellent choice for a variety of use cases in modern development. Whether you're working on a simple application, managing multiple environments, or deploying cloud-native microservices, Skaffold helps automate repetitive tasks, enabling faster development cycles and smoother CI/CD processes.