

Automated Deployment of Python Flask App Using Docker, Helm, Kubernetes, and Jenkins

Overview:

This project involves automating the deployment of a Python Flask application using a combination of Docker for containerization, Helm and Kubernetes for orchestration, and Jenkins for CI/CD. The application is deployed on a local Kubernetes cluster managed by Minikube, with Jenkins running locally to automate the build and deployment processes. The setup also includes Helm to manage Kubernetes releases and ngrok to tunnel local services for GitHub webhook integration.

Technologies Used:

- 1- Python 3.13.0: For building the Flask app.
- 2- Docker: For containerizing the Python Flask application.
- 3- Helm: Kubernetes package manager to manage the deployment.
- 4- Kubernetes (Minikube): Local Kubernetes environment for testing and running the application.
- 5- Jenkins: For CI/CD automation, managing builds, and deployment.
- 6- ngrok: Used for tunneling traffic between local Jenkins and GitHub webhooks.
- 7- GitHub: For version control and automated integration with Jenkins.
- 8- Prometheus: Open-source systems monitoring and alerting toolkit designed specifically for reliability and scalability in cloud-native environments.
- 9- Grafana: Visualization and analytics software that lets you query, visualize, and alert on metrics collected by Prometheus.

Goals:

The goal of this project is to automate the deployment of a Python-based Flask web application. The process involves creating a Docker image for the application, deploying it on a Kubernetes cluster using Helm, and automating these steps through a Jenkins pipeline. Traffic from GitHub to Jenkins is tunneled using ngrok, enabling webhook-based CI/CD integration.

Workflow and Process:

1- Python Application Setup:

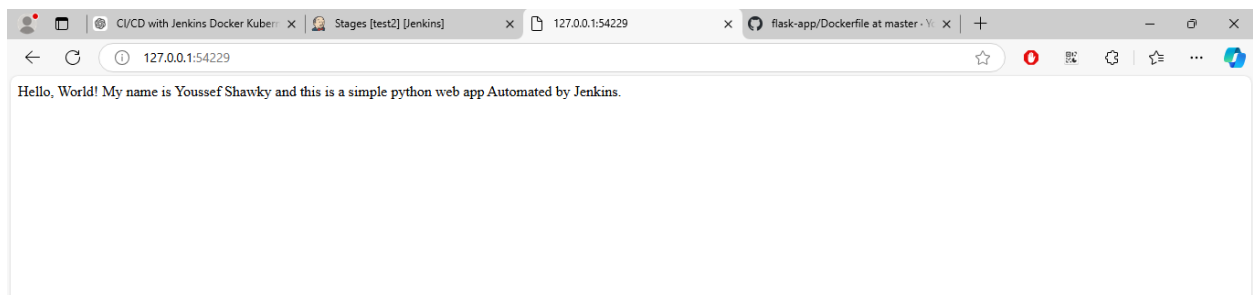
- Python Flask App Code:
 - A simple Flask app is developed in Python 3.13.0.
 - The app exposes endpoints for basic functionalities, such as a home page.

```
app.py x
C: > Users > Fr3on > Desktop > Project-2 > app.py > ...
1  from flask import Flask
2
3  #This creates an instance of the Flask class, representing your web application.
4  app = Flask(__name__)
5
6  #This is a route decorator that defines the URL path. In this case, it's the homepage (/).
7  @app.route('/')
8
9  # This function is executed when the homepage is accessed and returns a simple message.
10 def home():
11     return "Hello, World! My name is Youssef Shawky and this is a simple python web app."
12
13
14 if __name__ == '__main__': #a common Python idiom used to control the behavior of a script when i
15     app.run(host='0.0.0.0', port=5000) #This starts the web server and makes the app accessible,
```

Explanation:

- Flask(__name__): This creates an instance of the Flask class, representing your web application.
- @app.route('/'): This is a route decorator that defines the URL path. In this case, it's the homepage (/).
- home(): This function is executed when the homepage is accessed and returns a simple message.
- app.run(): This starts the web server and makes the app accessible. host='0.0.0.0' ensures it can be accessed externally, and port=5000 sets the port.

Output:



2- Dockerizing the Application:

- Dockerfile:

- A multi-stage Dockerfile is used to optimize the size of the final image.
- Stage 1 (Builder): Python 3.13.0 is used to install dependencies.
- Stage 2 (Slim Image): Python 3.8-slim is used to reduce image size.
- Files are copied from the builder stage to the final image, ensuring a lightweight production image.

```
app.py Dockerfile X
C: > Users > Fr3on > Desktop > Project-2 > Dockerfile > ...

4 # Set the working directory in the container
5 WORKDIR /app
6
7 # Copy only the requirements file first for caching purposes
8 COPY requirements.txt .
9
10 #Install the dependencies (Flask)
11 RUN pip install --no-cache-dir -r requirements.txt
12
13 # Copy the rest of the application files
14 COPY . .
15
16 # Use a smaller image for the final stage
17 FROM python:3.8.0-slim
18
19 #Set the working directory in the container
20 WORKDIR /app
21
22 # Copy installed dependencies from the builder stage
23 COPY --from=builder /usr/local/lib/python3.13/site-packages /usr/local/lib/python3.8/site-packages
24 COPY --from=builder /app /app
25
26 #Expose port 5000 to the outside world
27 EXPOSE 5000
28
29 #Define the command to run the Flask app
30 CMD ["python", "app.py"]
```

Before & after Multi-stage builds image size:

Windows PowerShell

```
PS C:\Users\Fr3on\Desktop\Project-2> docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
youssefshawky/my-flask-app	35	3f0db45d038d	8 hours ago	208MB
youssefshawky/my-flask-app	29	8ab0f203d4ee	9 hours ago	1.03GB
youssefshawky/my-flask-app	28	1f2eaa01c31f	9 hours ago	1.03GB

3- Kubernetes and Helm Setup:

- Helm Chart Creation:

- A Helm chart is created to manage the deployment of the Flask application on Kubernetes.
- The chart defines the Kubernetes resources (deployments, services) needed to run the app.
- Helm allows easy upgrades by setting the image version using Jenkins.

```
apiVersion: v2
name: flask-app-chart
description: A Helm chart for Kubernetes
```

```
# A chart can be either an 'application' or a 'library' chart.
```

```
type: application
```

```
# This is the chart version. This version number should be incremented each time you make changes
# to the chart and its templates, including the app version.
```

```
# Versions are expected to follow Semantic Versioning (https://semver.org/)
```

```
version: 0.1.0
```

```
# This is the version number of the application being deployed. This version number should be
# incremented each time you make changes to the application. Versions are not expected to
# follow Semantic Versioning. They should reflect the version the application is using.
# It is recommended to use it with quotes.
```

```
appVersion: "1.16.0"
```

- Kubernetes Deployment:

- Minikube is used to simulate a Kubernetes environment locally.
- The Flask application is deployed with a NodePort service type.
- The service allows external access to the application.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "flask-app-chart.fullname" . }}
  labels:
    {{- include "flask-app-chart.labels" . | nindent 4 }}
```

```

containers:
  - name: {{ .Chart.Name }}
    securityContext:
      {{- toYaml .Values.securityContext | nindent 12 }}
    image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default .Chart.AppVersion }}"
    imagePullPolicy: {{ .Values.image.pullPolicy }}
    ports:
      - name: http
        containerPort: 5000
        protocol: TCP

```

4- Jenkins Integration:

- Jenkins on Localhost:

- Jenkins is installed and run on the local Windows machine, not in a Docker container, to avoid potential networking issues with Minikube and Docker container interaction.
- The Jenkins pipeline pulls the latest code from GitHub, builds the Docker image, and deploys it to Minikube using Helm.

■ Jenkins Pipeline Script includes:

- Verifying Docker and Kubernetes setup.
- Deploying the new image using Helm.
- Sleep commands for handling delays between deployment stages.

Code:

```

pipeline {
    agent any // Use any available Jenkins agent

    environment {
        KUBECONFIG = 'C:/Users/Fr3on/.kube/config'
        DOCKER_IMAGE = "youssefshawky/my-flask-app" // Replace with your
        Docker Hub username
        KUBE_CONTEXT = "minikube" // Minikube Kubernetes context
    }

    stages {
        stage('Clone Repository') {
            steps {
                echo "Starting Clone Repository Stage..."
                // Clone the GitHub repository containing the Flask app
                git branch: 'master', url: 'https://github.com/Youssefshawky969/flask-
app.git'
            }
        }
    }
}

```

```

        echo "Repository cloned successfully."
    }
}

stage('Build Docker Image') {
    steps {
        echo "Starting Build Docker Image Stage..."
        script {
            // Build the Docker image for the Flask app
            def flaskAppImage =
docker.build("${DOCKER_IMAGE}:${env.BUILD_ID}")
        }
        echo "Docker image built successfully."
    }
}

stage('Test Application') {
    steps {
        echo "Starting Test Application Stage..."

        echo "Application tests completed."
    }
}

stage('Push Docker Image') {
    steps {
        echo "Starting Push Docker Image Stage..."
        script {
            // Log in to Docker Hub and push the built image
            docker.withRegistry('https://index.docker.io/v1/', '645b25b9-48c7-
4f77-a64f-8c24f9685066') {
                docker.image("${DOCKER_IMAGE}:${env.BUILD_ID}").push()
            }
        }
        echo "Docker image pushed to Docker Hub."
    }
}

stage('Deploy to Kubernetes') {

```

```

steps {
  echo "Starting Deploy to Kubernetes Stage..."
  script {
    // Set Minikube Kubernetes context
    bat "kubectl config use-context ${KUBE_CONTEXT}"

    // Deploy the application using Helm
    bat "helm upgrade --install flask-app ./flask-app-chart --set
image.tag=${env.BUILD_ID} --set service.type=NodePort"
  }
  echo "Deployment to Kubernetes completed."
}

stage('Sleep') {
  steps {
    echo "Sleeping for 30 seconds..."
    sleep 240
  }
}

post {
  success {
    echo 'Pipeline completed successfully!'
  }
  failure {
    echo 'Pipeline failed. Please check the logs for errors.'
  }
}
}

```

- Explanation:

1- Environment:

KUBECONFIG: Specifies the path to the Kubernetes configuration file used for accessing the Kubernetes cluster.

DOCKER_IMAGE: Defines the Docker image name for the Flask app.

KUBE_CONTEXT: Specifies the Kubernetes context (minikube), ensuring all kubectl commands target the Minikube cluster.

2- Clone Repository:

Jenkins pulls the Flask app's code from the GitHub repository using the Git plugin. It fetches from the master branch of the specified repository.

3- Build Docker Image:

This step builds the Docker image for the Flask app using the `docker.build()` function. `env.BUILD_ID` ensures that each image is tagged with the unique build ID to differentiate different builds.

4- Test Application:

In this stage, any unit or integration tests for the Flask application could be run. Currently, it's just a placeholder (echo) indicating that tests would be executed here.

5- Push Docker Image:

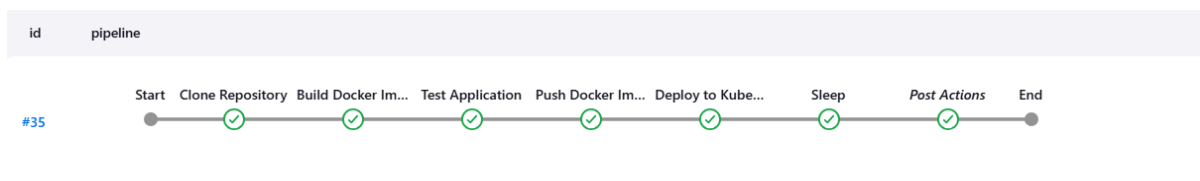
This stage logs in to Docker Hub using stored credentials.

6- Deploy to Kubernetes:

This stage deploys the Flask app to a Kubernetes cluster (Minikube). It first switches the Kubernetes context to Minikube using `kubectl config use-context`. Then, it uses Helm to either install or upgrade the deployment with the new Docker image (`env.BUILD_ID` ensures the latest version is deployed).

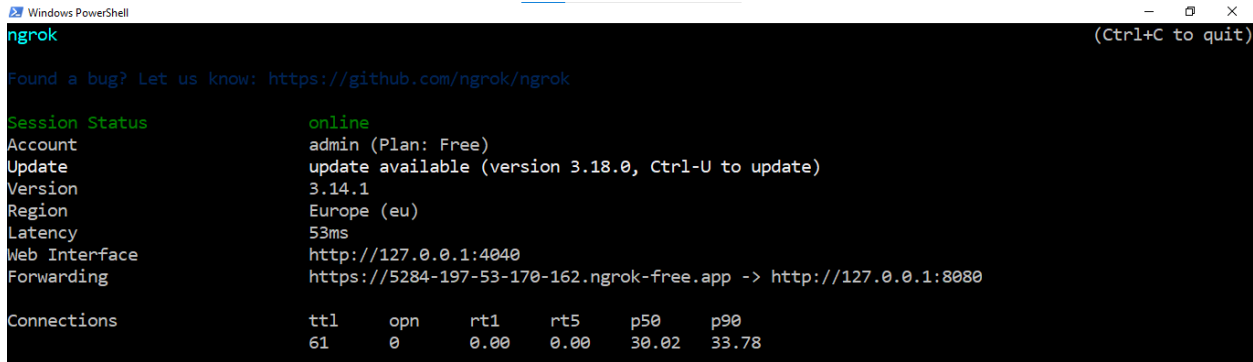
7- Sleep (Wait for Application to be Ready):

Adds a delay (240 seconds) after deployment to allow time for the Kubernetes pods and services to be fully operational.



5- Tunneling with ngrok:

- ngrok Setup:
 - ngrok is used to tunnel traffic from GitHub to Jenkins, as GitHub cannot access services running on localhost (127.0.0.1).
 - The ngrok tunnel is automated to be always active for continuous integration.



```
Windows PowerShell
ngrok
Found a bug? Let us know: https://github.com/ngrok/ngrok

Session Status      online
Account             admin (Plan: Free)
Update              update available (version 3.18.0, Ctrl-U to update)
Version             3.14.1
Region              Europe (eu)
Latency              53ms
Web Interface        http://127.0.0.1:4040
Forwarding           https://5284-197-53-170-162.ngrok-free.app -> http://127.0.0.1:8080

Connections          ttl    opn    rt1    rt5    p50    p90
                    61     0      0.00   0.00   30.02   33.78
```

Issues Faced and Solutions:

1- Jenkins Running on Localhost:

Problem: GitHub webhooks failed because Jenkins was running on localhost (127.0.0.1), which GitHub cannot access.

Solution: ngrok was used to tunnel traffic between the local Jenkins instance and GitHub, allowing successful webhook integration.

2- Multi-Stage Docker Build:

Problem: Multi-stage Docker build initially failed when copying files between different stages.

Solution: Ensured correct paths were used in the COPY directive to ensure dependencies were copied from the builder stage to the final image.

Contribution Work:

Kubernetes Monitoring with Prometheus and Grafana:

The project demonstrates setting up Kubernetes monitoring using Prometheus and Grafana. The setup provides detailed insights into resource usage, pod activity, and cluster health metrics through a customizable Grafana dashboard.

1- Prometheus Installation:

Add the Prometheus Helm repository:

```
Windows PowerShell
PS C:\Users\Fr3on\Desktop\Project-2\flask-app-chart> helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
```

```
Windows PowerShell
PS C:\Users\Fr3on\Desktop\Project-2\flask-app-chart> helm repo update
```

Install Prometheus in the <your-custom namespace>:

```
Windows PowerShell
PS C:\Users\Fr3on\Desktop\Project-2\flask-app-chart> helm install prometheus prometheus-community/prometheus --namespace general
```

2- Grafana Installation:

Add the Grafana Helm repository:

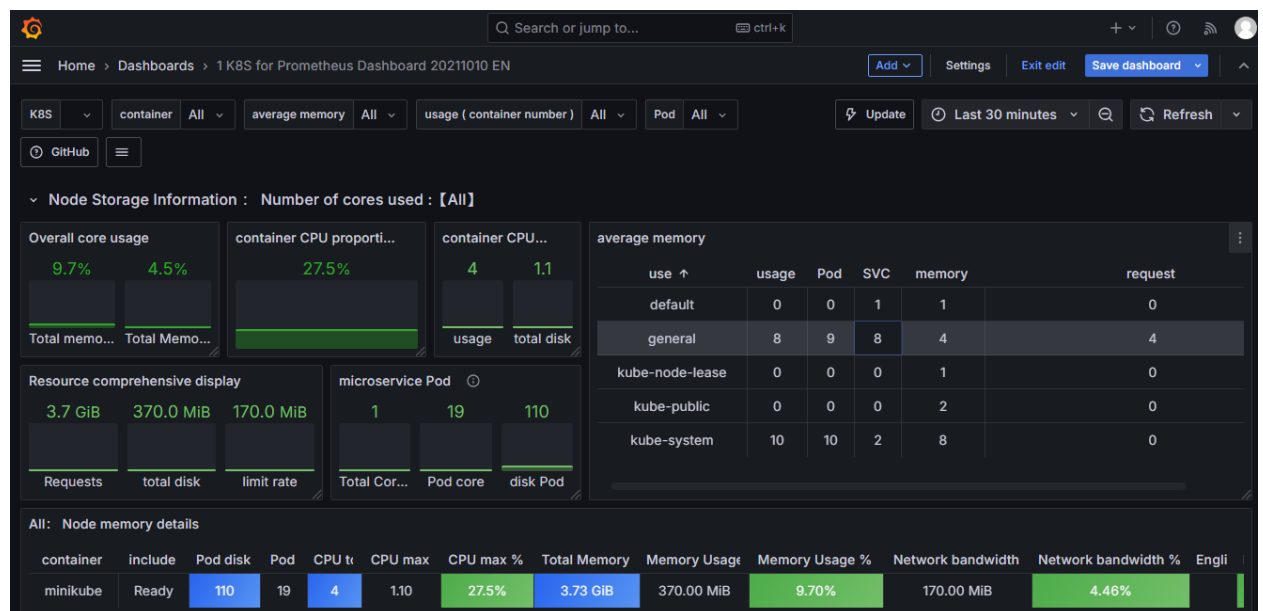
```
Windows PowerShell
PS C:\Users\Fr3on\Desktop\Project-2\flask-app-chart> helm repo add grafana https://grafana.github.io/helm-charts
```

```
Windows PowerShell
PS C:\Users\Fr3on\Desktop\Project-2\flask-app-chart> helm repo update
```

Install Prometheus in the <your-custom namespace>:

```
Windows PowerShell
PS C:\Users\Fr3on\Desktop\Project-2\flask-app-chart> helm install grafana grafana/grafana --namespace g
eneral
```

- 3- Expose Grafana as a NodePort service to access the web UI.
- 4- Connecting Grafana to Prometheus
 - Access Grafana by forwarding its port or accessing it via the exposed NodePort.
 - In Grafana, add Prometheus as a data source by pointing it to Prometheus's internal ClusterIP.
- 5- Configuring Grafana Dashboard:
 - Import the dashboard titled *1 K8S for Prometheus Dashboard 20211010 EN*.
 - This dashboard provides key metrics on Kubernetes cluster resource usage, pod counts, CPU and memory capacity, and other critical insights.



Conclusion:

This project demonstrates the effective use of a variety of DevOps tools and practices to automate the deployment of a Python Flask application. By utilizing Docker, Helm, and Kubernetes, the application is efficiently containerized, managed, and deployed in a local Minikube cluster. The integration of Jenkins automates the CI/CD pipeline, ensuring that new versions of the application are seamlessly built, tested, and deployed.

Additionally, the use of ngrok solves the challenge of local-hosted services interacting with public GitHub repositories for webhook-based automation. While the project faced several challenges, such as Kubernetes pod accessibility and Docker image cleanup issues, each problem was resolved with careful adjustments to the workflow. Moreover, we have successfully configured monitoring for Kubernetes clusters using Prometheus and Grafana. This solution provides a reliable and scalable approach to gain insights into cluster performance, enabling proactive management of resources and early detection of issues.

Overall, this setup provides a robust foundation for scaling up to production environments, and the inclusion of Helm charts and Jenkins pipelines ensures that the project can be maintained and upgraded with ease. The integration of all these tools creates a scalable, automated system that can be further extended for future development, testing, and continuous delivery needs.