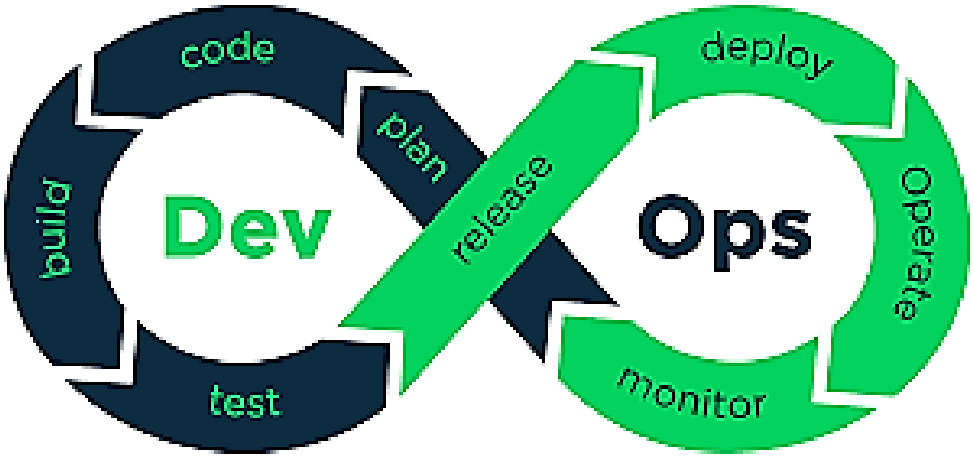
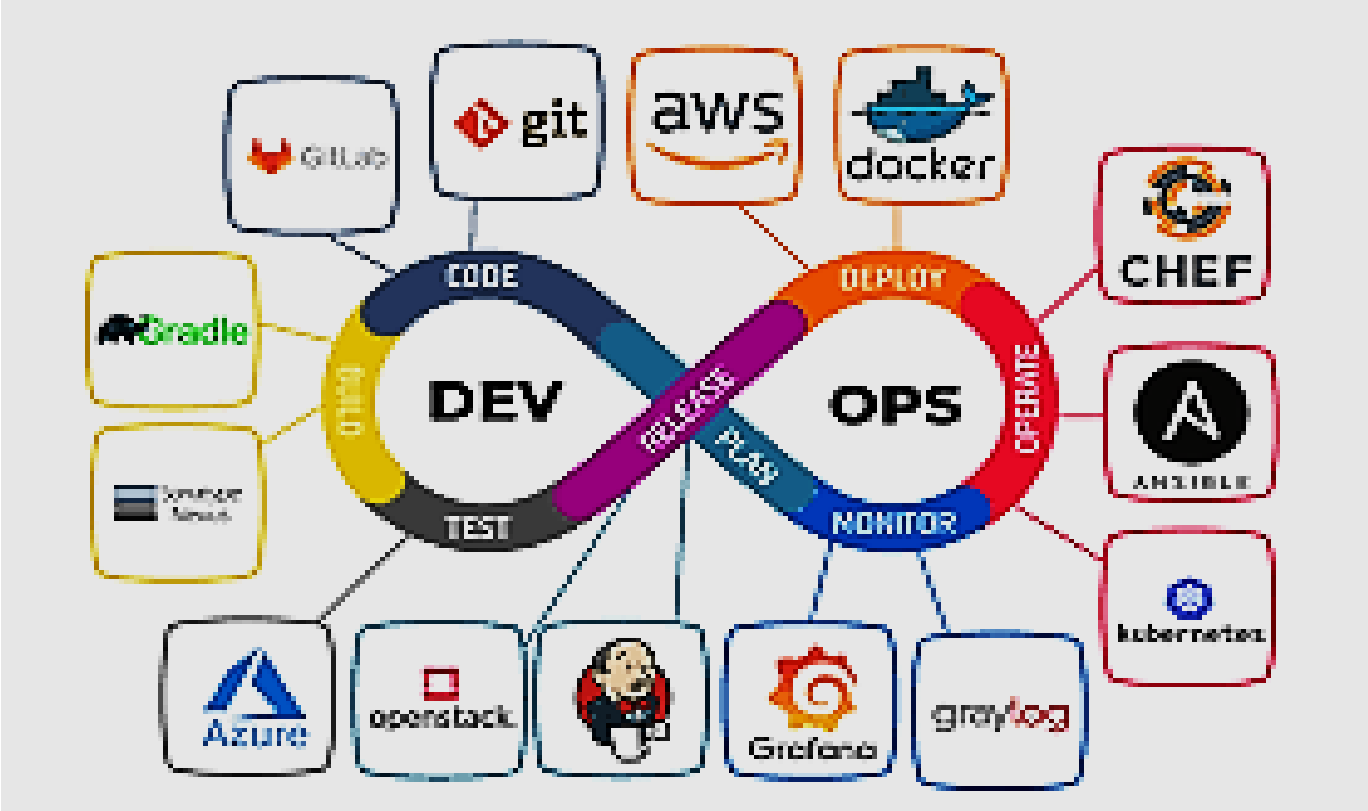
2024



DevOps in Action

TOOLS, PRACTICES, AND CLOUD INTEGRATION

Contents



Introduction to DevOps and Continuous Integration (CI)

What is DevOps?

DevOps is a set of practices, cultural philosophies, and tools designed to increase an organization's ability to deliver applications and services at high velocity. It combines software development (Dev) and IT operations (Ops) to shorten the development lifecycle and deliver high-quality software continuously.

Key Characteristics:

- Collaboration: Breaking down silos between development, operations, quality assurance, and security teams.
- Automation: Using tools to automate testing, integration, deployment, and infrastructure management.
- Continuous Integration/Continuous Delivery (CI/CD): Frequently integrating code changes and delivering them quickly.
- Monitoring and Feedback: Constant monitoring of applications and infrastructure to quickly detect and resolve issues.

Example: A Simple CI/CD Pipeline

Code Commit:

- Step 1: A developer writes code and commits it to a version control system (e.g., Git).
- Example: git commit -m "Add new feature X"

Automated Build:

- Step 2: A CI tool (like Jenkins, GitLab CI, or CircleCI) detects the commit and triggers an automated build.
- Example: A Jenkins pipeline script compiles the code and runs unit tests.

Automated Testing:

- Step 3: Automated tests (unit tests, integration tests) are executed.
- Example: The build fails if any tests do not pass, prompting the developer to fix issues.

Deployment:

- Step 4: Once tests pass, the code is automatically deployed to a staging or production environment using CD tools (e.g., Kubernetes, Docker).
- Example: A deployment script runs commands like kubectl apply -f deployment.yaml to update a Kubernetes cluster.

Monitoring:

- Step 5: Post-deployment, monitoring tools (such as Prometheus, Grafana) track application performance.
- Example: Alerts are set up to notify teams if the application response time exceeds a threshold.

The History and Evolution of DevOps

Early Beginnings:

- Agile Development: In the early 2000s, Agile methodologies began transforming software development by emphasizing iterative progress, collaboration, and customer feedback.
- Traditional Silos: Traditionally, development teams wrote code and operations teams managed infrastructure, leading to delays and miscommunication.

Emergence of DevOps:

- 2007-2008: The term "DevOps" started emerging as a response to the challenges faced in agile environments, particularly the disconnect between development and operations.
- Pioneers: Early adopters and thought leaders like Patrick Debois, Gene Kim, and Jez Humble began advocating for integrated practices.

Evolution and Adoption:

- Tooling Advances: The rise of tools such as Git for version control, Jenkins for CI/CD, and configuration management tools like Puppet and Chef facilitated the adoption of DevOps.
- Cloud Computing: The growth of cloud platforms (AWS, Azure, Google Cloud) accelerated the shift as they provided scalable, automated infrastructure.
- Microservices Architecture: DevOps practices further evolved with the advent of microservices, which require independent, scalable, and rapidly deployable services.

DevOps Culture and Practices

Cultural Elements:

- ✓ Collaboration and Communication: Breaking down barriers between traditionally separate teams.
- ✓ Shared Responsibility: Everyone, from developers to operations staff, is responsible for the system's performance.
- ✓ Continuous Improvement: Encouraging regular retrospectives and iterative enhancements.
- ✓ Learning from Failure: Implementing blameless post-mortems to learn from incidents without finger-pointing.

Common Practices:

Continuous Integration (CI):

- Practice: Regularly merging code changes into a shared repository.
- Benefit: Early detection of issues, which reduces integration problems.

Continuous Delivery (CD):

- Practice: Automating the release process so that deployments are quick and predictable.
- Benefit: Reduces time-to-market and minimizes manual errors.

Infrastructure as Code (IaC):

- Practice: Managing infrastructure using machine-readable configuration files.
- Example: Tools like Terraform, CloudFormation.
- Benefit: Version-controlled, repeatable, and scalable infrastructure setups.

Monitoring and Logging:

- Practice: Implementing comprehensive logging and monitoring to quickly detect anomalies.
- Benefit: Faster troubleshooting and proactive problem resolution.

Benefits of DevOps

Accelerated Delivery:

- ✓ Faster Releases: Automation of the build, test, and deployment processes leads to rapid and frequent releases.
- ✓ Example: With a CI/CD pipeline, a new feature can be deployed to production within hours rather than weeks or months.

Improved Collaboration:

- ✓ Cross-Functional Teams: Enhanced communication and shared goals lead to fewer misunderstandings and smoother workflows.
- ✓ Example: Regular stand-up meetings and integrated communication platforms (like Slack) foster collaboration between developers and operations.

Enhanced Quality and Reliability:

- ✓ Continuous Testing: Automated tests help identify and fix issues early in the development cycle.
- ✓ Monitoring: Continuous monitoring ensures quick detection of issues in production.
- ✓ Example: Automated unit, integration, and performance tests can significantly reduce the likelihood of production failures.

Cost Efficiency:

- ✓ Resource Optimization: Automated infrastructure provisioning and scaling reduce manual effort and optimize resource usage.
- ✓ Reduced Downtime: Faster detection and resolution of issues minimize downtime, saving costs associated with service interruptions.

Innovation and Agility:

- ✓ Rapid Experimentation: Developers can quickly test new ideas and roll back if needed.
- ✓ Scalability: The practices allow organizations to scale their systems efficiently in response to demand.
- ✓ Example: The ability to quickly spin up new environments (using tools like Docker and Kubernetes) empowers teams to experiment without risking the stability of production systems.

DevOps Tools Overview

DevOps Workflow Overview

1. Code Development: Developers write code and use Git for version control.
2. Continuous Integration (CI): Jenkins automates building and testing the code.
3. Infrastructure as Code (IaC): Terraform provisions cloud infrastructure.
4. Configuration Management: Ansible configures servers and deploys applications.
5. Containerization: Docker packages the application into containers.
6. Orchestration: Kubernetes manages containerized applications.
7. Monitoring: Prometheus, ELK, and Grafana monitor the application and infrastructure.

8. Testing: Postman is used for API testing.
9. Development Environment: Eclipse and JDK are used for coding and debugging.

Step-by-Step Workflow

1. Code Development
Tools: Git, Eclipse, JDK
Steps:
    a. Developers write code using Eclipse with JDK installed for Java development.
    b. Code is committed to a Git repository (e.g., GitHub, GitLab, or Bitbucket).
    c. Branches are created for features, bugs, or hotfixes.
    d. Pull requests are used for code reviews before merging into the main branch.

2. Continuous Integration (CI)
Tools: Jenkins, Git
Steps:
    a. Jenkins is configured to monitor the Git repository for changes.
    b. When a change is pushed to the repository, Jenkins triggers a build.
    c. Jenkins runs unit tests, static code analysis, and builds the application.
    d. If the build and tests pass, Jenkins generates a build artifact (e.g., a JAR file).

3. Infrastructure as Code (IaC)
Tools: Terraform
Steps:
    a. Use Terraform to define cloud infrastructure (e.g., AWS, Azure, GCP) in code.
    b. Write Terraform configuration files (main.tf, variables.tf, etc.) to provision resources like VMs, networks, and storage.
    c. Run terraform init, terraform plan, and terraform apply to create the infrastructure.

4. Configuration Management
Tools: Ansible
Steps:
    a. Use Ansible to configure servers and deploy the application.
    b. Write Ansible playbooks to install dependencies (e.g., JDK, Docker) and deploy the build artifact.
    c. Run the playbook to configure the servers provisioned by Terraform.

5. Containerization
Tools: Docker
Steps:
    a. Create a Dockerfile to define the application's container image.
    b. Build the Docker image: docker build -t my-app:latest .
    c. Push the Docker image to a container registry (e.g., Docker Hub): docker push my-app:latest

6. Orchestration
Tools: Kubernetes
Steps:

a. Write Kubernetes manifest files (deployment.yaml, service.yaml) to define how the application should run.
b. Deploy the application to a Kubernetes cluster: kubectl apply -f deployment.yaml
c. Kubernetes manages the application's lifecycle, scaling, and load balancing.

7. Monitoring
Tools: Prometheus, ELK, Grafana
Steps:

a. Prometheus collects metrics from the application and infrastructure.
b. ELK Stack (Elasticsearch, Logstash, Kibana) is used for log aggregation and analysis.
c. Grafana visualizes metrics and logs using dashboards.
d. Set up alerts in Grafana or Prometheus to notify the team of issues.

8. Testing
Tools: Postman
Steps:

a. Use Postman to create and run API tests against the deployed application.
b. Automate API testing by integrating Postman collections into the CI/CD pipeline.

9. Development Environment
Tools: Eclipse, JDK
Steps:

a. Developers continue to use Eclipse and JDK for coding and debugging.
b. They pull the latest code from Git, make changes, and push updates to trigger the CI/CD pipeline.

Summary of Tools and Their Roles

Tool	Purpose
Git	Version control and collaboration
Jenkins	Continuous Integration (CI) and automation
Terraform	Infrastructure as Code (IaC) for provisioning cloud resources
Ansible	Configuration management and application deployment
Docker	Containerization of the application
Kubernetes	Orchestration and management of containerized applications
Prometheus	Monitoring and alerting for metrics
ELK Stack	Log aggregation and analysis (Elasticsearch, Logstash, Kibana)
Grafana	Visualization of metrics and logs
Postman	API testing
Eclipse	Integrated Development Environment (IDE) for coding
JDK	Java Development Kit for building and running Java applications

Example Workflow in Action

1) A developer writes code in Eclipse and pushes it to Git.
2) Jenkins detects the change, builds the code, and runs tests.
3) Terraform provisions the infrastructure, and Ansible configures it.
4) Docker packages the application, and Kubernetes deploys it.
5) Prometheus and ELK monitor the application, while Grafana visualizes the data.
6) Postman tests the APIs to ensure everything works as expected.

Summary Table of Alternative Tools

Category	Primary Tool	Popular Alternatives
Version Control	Git	Mercurial, Subversion (SVN), Perforce
Continuous Integration	Jenkins	GitLab CI/CD, CircleCI, Travis CI, Azure DevOps Pipelines, GitHub Actions
Infrastructure as Code	Terraform	AWS CloudFormation, Pulumi, Ansible, Crossplane
Configuration Management	Ansible	Puppet, Chef, SaltStack, CFEngine
Containerization	Docker	Podman, Containerd, CRI-O, LXC/LXD
Container Orchestration	Kubernetes	Docker Swarm, Nomad, Apache Mesos, OpenShift
Monitoring & Observability	Prometheus, ELK	Datadog, New Relic, Splunk, Zabbix, Dynatrace
API Testing	Postman	Insomnia, SoapUI, Swagger, Katalon Studio
IDE	Eclipse	IntelliJ IDEA, Visual Studio Code, NetBeans, PyCharm
Java Development Kit	JDK	OpenJDK, Amazon Corretto, Adoptium, Zulu

Activity 1: Identify DevOps Server and Developer Machine Setup
Activity 2: Prepare platform (physical local machine, VM, Cloud VM)
Activity 3: Connect to platform using RDP or SSH
Activity 4: Setup Java JDK and Eclipse (or Python and VSCode) for simulating development work
Activity 5: Create a sample project

Version Control with Git

Using Git (Local)

1. Download and install Git for Windows

```
https://git-scm.com/downloads
```

If you're on Fedora (or any closely-related RPM-based distribution, such as RHEL or CentOS), you can use dnf:

```
$ sudo dnf install git-all
```

If you're on a Debian-based distribution, such as Ubuntu, try apt:

```
$ sudo apt install git-all
```

2. Using Git (Local Repository Version Control)
   On your project folder:
   (use Git Bash to easily access some linux commands like touch and ls)

   $ git init
   $ git config user.name "john"
   $ git config user.email "john@gmail.com"
   $ touch .gitignore

   Gitignore files are items that will **NOT** be added to the repository.

   Sample content of gitignore file:

```
Myreports/
*.txt
Otherfiles/
```

Check files for staging: ***$ git status***
Stage files for tracking: ***$ git add <filename>***
Or ***$ git add ****
Then check: ***$ git status***

Unstage files
$ git rm --cached <filename>
$ git rm --cached *

Perform a local commit ***$ git commit -m "my first commit"***
Show commit history ***$ git log***

Do some project code changes for a second commit.

Check changes in code ***$ git diff***
Try committing a second time ***$ git commit -m "my second commit"***

Jump to an old commit
$ git checkout <hash>

From here, you can make changes and perform a new commit
$ git commit -m "new commit"

Or discard changes and go back to previous master head
$ git switch -

Keep changes and create a branch from there
$ git switch -c <branch name>

Create a new branch from master and jump to it
$ git checkout -b <branch name>

Check which branch you are on
$ git branch

Jump to a branch
$ git branch <branchname>

Activity 6: Demonstrate and perform basic Git commands

Using GitHub

Upload local project to github repo

Create a github repo and clone it locally
$ git clone <github repo url>

Explain windows credential manager where github account login is cached in local computer

Note: you may also programmatically add and set origin github repo url
$ git remote add origin <github repo url>
$ git remote -v

Push your local repo (with correct origin url)
$ git push origin –all
$ git push origin master
Note: ignored files set in .gitignore will not be uploaded to remote github repo

Pull updates from github remote repo to your system
$ git pull <github repo>

Activity 7: Demonstrate and perform basic GitHub Operations

Git and Github Activity

1. Open source case (owner and contributor)
   a. Owner creates a repo
   b. Contributor visits the owner's repo and forks it to his own github account
   c. Owner sees who forks his repo
   d. Contributor can submit issues by visiting the original repo
   e. Owner sees issues
   f. Contributor can now clone his forked version and work on it
   g. Contributor makes changes to his local copy and makes a push to his forked repo.

Note: if branch push only
\> git push origin <branch name>

h. Contributor, using his account, can now make a pull request toward the owner of the original repo
i. Owner sees the pull request, checks the code submitted by the contributor and decides if he approves it or not.
j. If owner approves, owner can choose to merge the contributors forked version to original by merging. Verify changes to original repo.
k. If owner disapproves, he can send message to contributor thru the pull request made. The contributor can then make changes, re-push to his fork and issue another pull request.

Activity 8: Demonstrate and perform GitHub Collaboration Operations

Continuous Integration (CI) Concepts

Continuous Integration (CI) using GitHub Actions

Step 1: Create a New GitHub Repository

a. Log in to GitHub and click on the New repository button.
b. Name your repository (for example, python-ci-demo) and add a short description if you like.
c. Initialize the repository with a README (optional) and click Create repository.

d. Clone the repository to your local machine:

```
git clone https://github.com/<your-username>/python-ci-demo.git
cd python-ci-demo
```

Step 2: Create a Simple Python Script

a. Create a file named hello.py in the repository directory:

```
# hello.py
def add(a, b):
  """Return the sum of a and b."""
  return a + b

if __name__ == "__main__":
  result = add(2, 3)
  print("2 + 3 =", result)
```

b. Commit your changes:

```
git add hello.py
git commit -m "Add hello.py with a simple add function"
```

Step 3: Write Tests for Your Python Script

a. We'll use the popular pytest framework to write a simple test for our add function.
b. Create a file named test_hello.py:

```
# test_hello.py
from hello import add

def test_add():
  # Test a few cases
  assert add(2, 3) == 5
  assert add(-1, 1) == 0
  assert add(0, 0) == 0
```

c. Commit your test file:

```
git add test_hello.py
git commit -m "Add tests for hello.py using pytest"
```

Step 4: Configure GitHub Actions Workflow

a. We'll now create a GitHub Actions workflow file to automatically run our tests on every push or pull request to the main branch.
b. Create the workflow directory and file:

```
mkdir -p .github/workflows
```

c. Create a file named .github/workflows/ci.yml and add the following content:

```
name: CI

# Trigger the workflow on push or pull request events targeting the main branch.
on:
 push:
  branches:
   - main
 pull_request:
  branches:
   - main

jobs:
 build:
  # Use a GitHub-hosted runner with Ubuntu.
  runs-on: ubuntu-latest

  steps:
   # Step 1: Check out the repository code.
   - uses: actions/checkout@v3

   # Step 2: Set up Python.
   - name: Set up Python 3.x
    uses: actions/setup-python@v4
    with:
     python-version: '3.x'
```

```
    # Step 3: Install dependencies.
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install pytest

    # Step 4: Run the tests using pytest.
    - name: Run tests
      run: pytest
```

Explanation of the Workflow File:

on: The workflow triggers on pushes and pull requests to the main branch.
jobs.build.runs-on: Specifies that the job will run on the latest Ubuntu runner.
steps:

1) actions/checkout@v3: Checks out your repository code.
2) actions/setup-python@v4: Sets up a Python environment (using any Python 3 version).
3) Install dependencies: Upgrades pip and installs pytest.
4) Run tests: Executes the tests with the pytest command.

d. Commit and push your workflow file:

```
git add .github/workflows/ci.yml
git commit -m "Add GitHub Actions workflow for CI"
git push origin main
```

Step 5: Verify the CI Pipeline

a. Push Your Changes:
   When you push your commits to GitHub, the GitHub Actions workflow will automatically trigger.

b. Monitor the Workflow:
   a. Go to your GitHub repository on the web.
   b. Click on the Actions tab.
   c. You should see your workflow running. Click on it to view the logs and results.

c. Review the Output:
   a. If everything is set up correctly, you'll see output that includes:
      i. The Python version installed.
      ii. Confirmation that pytest ran.
      iii. Test results showing that all assertions in test_hello.py passed.

Activity 9: Demonstrate and perform basic CI Operations using GitHub Actions

Continuous Delivery (CD) and Automated Testing

Introduction to Continuous Delivery (CD)

Continuous Delivery (CD) is a software development practice where code changes are automatically built, tested, and prepared for a release to production. It extends Continuous Integration (CI) by automating the delivery of validated code to one or more environments (such as staging and production) with minimal manual intervention.

Key Steps in a CD Pipeline

1) Code Commit: Developers commit changes to a shared repository.
2) Automated Build & Test (CI): The CI system automatically builds the code and runs a suite of tests.
3) Artifact Creation: Successful builds produce deployable artifacts (e.g., binaries, container images).
4) Deployment to Staging: The artifact is automatically deployed to a staging (or pre-production) environment.
5) Release to Production: With additional checks (manual or automated), the artifact is deployed to production.

Simple Example

Imagine you have a simple web application. Your CD pipeline might look like this:

1) Developer commits code →
2) CI system (e.g., GitHub Actions) builds the app →
3) Tests are run →
4) Artifact (e.g., Docker image) is created →
5) The artifact is automatically deployed to a staging environment →
6) After validation, the same artifact is promoted to production

The Difference Between CI and CD

While CI and CD are closely related, they address different parts of the software development lifecycle:

Continuous Integration (CI):

- Focus: Merging code changes frequently and verifying each change by automatically building and testing.
- Goal: Detect integration issues early.
- Example: Running unit tests and integration tests every time code is pushed.

Continuous Delivery (CD):

- Focus: Automatically deploying the validated code to production-like environments.
- Goal: Ensure that code is always in a deployable state.
- Example: Automatically deploying the application to a staging environment after CI passes.

Comparison Table

Aspect	Continuous Integration (CI)	Continuous Delivery (CD)
Primary Goal	Integrate and test code changes	Automate delivery to production-ready state
Automation Focus	Build and test processes	Deployment process
Frequency	Multiple times per day	On every successful build (or at set intervals)
Feedback	Immediate feedback on code quality	Deployment status feedback

Benefits and Challenges of Continuous Delivery

Benefits

- ✓ Faster Time-to-Market: Automating deployments helps deliver new features quickly.
- ✓ Improved Quality and Reliability: Frequent, small releases reduce risk and make issues easier to diagnose.
- ✓ Enhanced Collaboration: Teams work with a shared understanding of code quality and readiness for release.
- ✓ Reduced Manual Effort: Automation minimizes human error and frees up resources for innovation.

Challenges

- ✓ Complexity in Setup: Building and maintaining robust pipelines can be technically challenging.
- ✓ Cultural Change: Requires teams to adopt new processes and embrace automation.
- ✓ Tooling and Integration: Ensuring compatibility between various CI/CD tools and cloud environments can be demanding.
- ✓ Environment Parity: Keeping development, staging, and production environments consistent is critical and sometimes difficult.

Building an Automated Deployment Pipeline

*Example of creating an automated deployment pipeline using GitHub Actions for a Python web application.

Example Scenario

We'll assume:

- Your code is hosted on GitHub.
- Your application is containerized.
- You want to deploy to a staging environment automatically after tests pass.

Step-by-Step Pipeline Setup

1. Repository Setup: Create your GitHub repository and add your application code and tests.
2. Dockerize Your Application: Create a Dockerfile for your application. For example:

```
# Dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

3. Create a GitHub Actions Workflow: Create a file at .github/workflows/deploy.yml:

```
name: CI/CD Pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

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

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install pytest

      - name: Run tests
        run: pytest

      - name: Build Docker image
        run: |
          docker build -t my-app:latest .

      # Simulated deployment to a staging environment
      - name: Deploy to Staging
        run: |
          echo "Deploying the Docker image to staging environment..."
          # In a real scenario, use commands such as:
          # docker tag my-app:latest registry.example.com/my-app:latest
          # docker push registry.example.com/my-app:latest
          # then trigger deployment (e.g., via SSH, API call, etc.)
```

Explanation:

- Checkout & Setup: The workflow checks out the code and sets up Python.
- Testing: Runs your tests with pytest.
- Build: Builds a Docker image from your application.
- Deploy: A placeholder step simulates deployment to a staging environment. In a real setup, you might push the image to a container registry and trigger a deployment using your cloud provider's API or deployment tool.
- Push and Verify: Commit your changes and push to GitHub. Go to the Actions tab in your repository to monitor the pipeline.

Activity 10: Demonstrate and perform basic CI/CD Operations using GitHub Actions and Docker

Integration with Cloud Providers (AWS, Azure, GCP)

Modern CD pipelines often integrate directly with cloud providers to deploy applications. Below are brief examples for AWS, Azure, and GCP.

AWS Integration

Using AWS CodeDeploy or Elastic Beanstalk:

1. Prepare AWS Credentials: Store AWS credentials securely (e.g., using GitHub Secrets).
2. Configure Deployment: In your GitHub Actions workflow, add steps to deploy:

```
- name: Configure AWS credentials
  uses: aws-actions/configure-aws-credentials@v2
  with:
    aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
    aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    aws-region: us-west-2

- name: Deploy to Elastic Beanstalk
  run: |
    eb init my-app --region us-west-2
    eb deploy my-app-env
```

Note: This uses the Elastic Beanstalk CLI to deploy your application.

Azure Integration

Using Azure Web Apps or Azure Kubernetes Service (AKS):

1. Prepare Azure Credentials: Set up a service principal and store its credentials as GitHub Secrets.
2. Configure Deployment:

```
- name: 'Deploy to Azure Web App'
  uses: azure/webapps-deploy@v2
  with:
    app-name: 'my-azure-app'
    slot-name: 'staging'
```

```
  publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE }}
  package: '.'
```

Note: This action deploys your application directly to an Azure Web App.

GCP Integration

Using Google Cloud Build and Google Kubernetes Engine (GKE) or Cloud Run:

1. Prepare GCP Credentials: Store your service account key in GitHub Secrets.
2. Configure Deployment:

```
- name: Set up Cloud SDK
  uses: google-github-actions/setup-gcloud@v1
  with:
    service_account_key: ${{ secrets.GCP_SA_KEY }}
    project_id: your-gcp-project-id

- name: Deploy to Cloud Run
  run: |
    gcloud run deploy my-service \
      --image gcr.io/your-gcp-project-id/my-app:latest \
      --region us-central1 \
      --platform managed
```

Note: This step uses the Google Cloud SDK to deploy your Docker image to Cloud Run.

Automated Testing in DevOps

Automated testing in a DevOps context means running tests automatically at various stages of the software delivery process. The goal is to catch issues as early as possible, reduce manual work, and ensure that code is always in a deployable state. Automated tests become an integral part of the Continuous Integration (CI) and Continuous Delivery (CD) pipelines.

Key Benefits:

- ✓ Fast Feedback: Quickly identify defects after code changes.
- ✓ Consistency: Tests run the same way every time.
- ✓ Efficiency: Reduce human error and free developers to focus on feature development.
- ✓ Continuous Quality: Ensure that every build meets quality standards.

Types of Automated Tests (Unit, Integration, End-to-End)

Automated tests can be categorized based on their scope and purpose:

Unit Tests:

- Test the smallest parts (functions, methods) of an application in isolation.
- Example: Verifying that a math function returns the correct sum.

Integration Tests:

- Test the interactions between multiple components or systems to ensure they work together as expected.

- Example: Testing that a function that calls a database returns the expected result when both the function and database interactions are combined.

End-to-End (E2E) Tests:

- Simulate real user scenarios by testing the application as a whole—from start to finish—to ensure that all parts of the system work together.
- Example: Using a web browser automation tool to simulate a user logging in and making a purchase on a website.

Test Automation Tools (Selenium, JUnit, etc.)

Various tools are available for automating different types of tests. Some popular tools include:

For Unit Testing:

- Python: pytest, unittest
- Java: JUnit, TestNG

For Integration Testing:

- Python: pytest (with fixtures), tox
- Java: Spring Test, Arquillian

For End-to-End Testing:

- Web Applications: Selenium, Cypress, Puppeteer
- APIs: Postman (with Newman), RestAssured

Integrating Testing into CI/CD Pipelines

In a CI/CD pipeline, automated tests are triggered automatically when code is committed or when a pull request is created. This integration ensures that every change is validated by running a suite of tests before the code is merged or deployed. Below is a step-by-step example using GitHub Actions as the CI/CD platform.

Example: Integrating Automated Testing into a CI/CD Pipeline

Let's assume you have a simple Python project with the following structure:

```
my-python-app/
├── app.py
├── requirements.txt
└── tests/
    ├── unit/
    │   └── test_app.py
    ├── integration/
    │   └── test_app_integration.py
    └── e2e/
        └── test_app_e2e.py
```

Step 1: Create a Simple Python Application
File: app.py

```
# app.py
def add(a, b):
  """Return the sum of a and b."""
  return a + b

if __name__ == "__main__":
  print("Hello, World!")
```

Step 2: Write Different Types of Tests

Unit Test (Testing a single function):
File: tests/unit/test_app.py

```
from app import add

def test_add():
  assert add(1, 2) == 3
  assert add(-1, 1) == 0
  assert add(0, 0) == 0
```

Integration Test (Testing component interaction):
For a simple app, an integration test might look similar to a unit test. In a real-world scenario, you might combine multiple modules or include database/API interactions.
File: tests/integration/test_app_integration.py

```
from app import add

def test_integration_add():
  # Imagine a scenario where add() is used in a larger process
  result = add(2, 3)
  # Here, the integration test validates that the process (using add) works as expected.
  assert result == 5
```

End-to-End Test (Simulating user behavior):
For demonstration, we'll simulate running the application and verifying its output. In web apps, this might be done with Selenium or Cypress. File: tests/e2e/test_app_e2e.py

```
import subprocess

def test_e2e_app_output():
  # Run the application as a subprocess
  result = subprocess.run(['python', 'app.py'], capture_output=True, text=True)
  # Check if the expected output is in the application output
  assert "Hello, World!" in result.stdout
```

Step 3: Create a GitHub Actions Workflow
Create a file .github/workflows/test.yml in your repository with the following content:

```
name: CI Pipeline

on:
 push:
  branches: [ main ]
 pull_request:
  branches: [ main ]

jobs:
 run-tests:
  runs-on: ubuntu-latest

  steps:
   - name: Checkout code
    uses: actions/checkout@v3

   - name: Set up Python
    uses: actions/setup-python@v4
    with:
     python-version: '3.x'

   - name: Install dependencies
    run: |
     python -m pip install --upgrade pip
     pip install -r requirements.txt
     pip install pytest

   - name: Run unit tests
    run: pytest tests/unit

   - name: Run integration tests
    run: pytest tests/integration

   - name: Run end-to-end tests
    run: pytest tests/e2e
```

Explanation:

- Trigger: The workflow runs on pushes or pull requests to the main branch.
- Checkout and Setup: It checks out your code and sets up the required Python version.
- Dependency Installation: It installs the project dependencies as well as pytest for running tests.
- Test Execution: It runs tests in separate directories, so you can easily differentiate between unit, integration, and E2E tests.

Step 4: Commit and Push Your Code
Use Git to add, commit, and push your changes:

```
git add .
git commit -m "Add automated tests and CI workflow"
git push origin main
```

After pushing, navigate to the Actions tab in your GitHub repository to monitor the CI pipeline. You should see your tests running step by step, and if all tests pass, the pipeline completes successfully.

Activity 11: Demonstrate and perform Integrated Testing in CI/CD Operations using GitHub Actions, Language Testing Tool(s) and Docker

Code Quality and Security in DevOps

In DevOps, ensuring that code is both high quality and secure is critical. This focus helps to catch errors and vulnerabilities early in the development lifecycle, reducing risks and preventing costly issues later on. Key practices include:

Code Quality:

- Regularly assessing your code for maintainability, readability, and adherence to coding standards. This is often achieved through code reviews, automated tests, and static analysis tools.

Security:

- Integrating security measures (sometimes called DevSecOps) into the development pipeline so that vulnerabilities are identified and remediated as early as possible. This includes practices such as secure coding, dependency management, and vulnerability scanning.

Key Goals:

- ✓ Reduce technical debt.
- ✓ Ensure code follows best practices.
- ✓ Identify and remediate security issues early.
- ✓ Automate quality and security checks within CI/CD pipelines.

Static Code Analysis

Static Code Analysis is the process of examining your source code without executing it. This analysis checks for code quality issues, potential bugs, style violations, and security vulnerabilities. It is an important step in a CI/CD pipeline because it provides immediate feedback to developers.

Example: Using a Python Example with Flake8
We'll demonstrate static code analysis on a simple Python project using Flake8.

Step 1: Set Up Your Python Project
Create a project directory and files:

```
mkdir my-python-app
cd my-python-app
```

```
touch app.py
```

Add a simple Python script (app.py):

```
# app.py

def add(a, b):
  """Return the sum of a and b."""
  return a + b

def subtract(a, b):
  """Return the difference of a and b."""
  return a - b

if __name__ == "__main__":
  result = add(2, 3)
  print("2 + 3 =", result)
```

Step 2: Install and Run Flake8
Create a virtual environment (optional but recommended):

```
python3 -m venv venv
source venv/bin/activate  # On Windows use: venv\Scripts\activate
```

Install Flake8:

```
pip install flake8
```

Run Flake8 to analyze your code:

```
flake8 app.py
```

Expected Outcome:

- Flake8 will check your code for style violations and potential errors. If your code complies with the guidelines, you'll see no output; otherwise, Flake8 will list warnings or errors with line numbers.

Step 3: Integrate Static Analysis into CI
For integration into a CI pipeline (using GitHub Actions, for example), create a workflow file:

Create a file .github/workflows/static-analysis.yml:

```
name: Static Code Analysis

on:
 push:
  branches: [ main ]
 pull_request:
  branches: [ main ]

jobs:
```

```
lint:
  runs-on: ubuntu-latest

  steps:
    - name: Checkout Code
      uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.x'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8

    - name: Run Flake8
      run: flake8 .
```

Commit and Push Your Changes:

```
git add .
git commit -m "Add static analysis using Flake8"
git push origin main
```

Now, every time code is pushed or a pull request is opened, GitHub Actions will run Flake8 and report any issues.

Vulnerability Scanning Tools

Vulnerability scanning tools assess your code and its dependencies for known security vulnerabilities. These tools help ensure that your application does not include outdated or insecure libraries, and they can even scan for potential issues in the code itself.

Popular Tools:

- OWASP Dependency-Check: Scans project dependencies for known vulnerabilities.
- Snyk: Provides vulnerability scanning and remediation for open-source dependencies.
- Bandit: A tool specifically for Python that identifies common security issues in code.

Example: Using Bandit for a Python Project
We'll use Bandit to scan our Python code for security vulnerabilities.

Step 1: Install Bandit
If you have your virtual environment active (or you can install it globally), run:

```
pip install bandit
```

Step 2: Run Bandit on Your Code
Run Bandit against your Python file:

```
bandit -r .
```

Expected Outcome:
Bandit will output a report indicating any security issues found in your code. It will list the file, line number, issue, and severity.

Step 3: Integrate Vulnerability Scanning into CI
Add vulnerability scanning to your CI pipeline with GitHub Actions. Create a workflow file for security scanning:

Create a file .github/workflows/security-scan.yml:

```
name: Security Scan

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  bandit-scan:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Code
        uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.x'

      - name: Install Bandit
        run: |
          python -m pip install --upgrade pip
          pip install bandit

      - name: Run Bandit Security Scan
        run: bandit -r .
```

Commit and Push Your Changes:

```
git add .
git commit -m "Add vulnerability scanning using Bandit"
git push origin main
```

Now, every push or pull request will trigger Bandit to scan the repository, providing immediate feedback on potential security issues.

Activity 12: Demonstrate and perform Code Quality Checking in CI/CD Operations

Infrastructure as Code (IaC)

Understanding Infrastructure as Code (IaC)

Infrastructure as Code (IaC) is the practice of managing and provisioning infrastructure (servers, networks, databases, etc.) through machine-readable configuration files.

This approach allows you to:

- Version Control: Keep your infrastructure definitions in Git.
- Reproducibility: Recreate environments reliably.
- Automation: Integrate with CI/CD pipelines for automated deployments.
- Consistency: Ensure that all environments (development, staging, production) remain consistent.

Key Benefits:

- ✓ Reduced manual error and drift between environments.
- ✓ Faster provisioning and scaling.
- ✓ Better collaboration between teams through code reviews and automated testing.

Creating Infrastructure with Terraform

Terraform is an open-source IaC tool that uses declarative configuration files to manage infrastructure across various cloud providers (AWS, Azure, GCP, etc.). You define your desired infrastructure state in code, and Terraform creates (or updates) those resources.

Example IAC Procedure with Terraform

Step 1: Write a Terraform Configuration

Create a file named main.tf in your repository:

```
# main.tf

# Specify the AWS provider and region
provider "aws" {
  region = "us-east-1"
}

# Create an EC2 instance resource
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"  # Replace with a valid AMI ID for your region
```

```
  instance_type = "t2.micro"

  # (Optional) Use user_data to install Docker and run a container at boot time
  user_data = <<-EOF
          #!/bin/bash
          sudo apt-get update -y
          sudo apt-get install -y docker.io
          sudo docker run -d -p 80:80 my-python-app:latest
          EOF

  tags = {
    Name = "TerraformExample"
  }
}
```

Note:

- Ensure you have a valid AWS AMI ID for your region.
- The user_data script demonstrates how you might bootstrap the instance (installing Docker and starting a container) but will later be augmented with Ansible.

Step 2: Initialize and Apply Terraform
From your terminal, run the following commands in the directory containing main.tf:

```
terraform init    # Initializes the Terraform working directory and downloads provider plugins.
terraform plan    # Shows the execution plan.
terraform apply   # Applies the changes (type 'yes' when prompted).
```

Step 3: Automate Terraform via GitHub Actions
Create a GitHub Actions workflow file at .github/workflows/terraform.yml:

```
name: Terraform CI/CD

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  terraform:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Code
        uses: actions/checkout@v3

      - name: Set up Terraform
```

```
    uses: hashicorp/setup-terraform@v2
    with:
      terraform_version: "1.0.0"

  - name: Terraform Init
    run: terraform init

  - name: Terraform Plan
    run: terraform plan

  - name: Terraform Apply
    if: github.event_name == 'push'
    run: terraform apply -auto-approve
    env:
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

Setup:
Store your AWS credentials in GitHub Secrets (AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY).

Activity 13: Demonstrate and perform IaC in CI/CD Operations using Terraform

Configuration Management with Ansible

Ansible is a configuration management and automation tool that helps you set up and maintain systems after they are provisioned. It uses YAML-based playbooks to define tasks such as installing software, configuring services, and deploying applications.

Example with Ansible
Step 1: Create an Inventory File
Create an inventory file named inventory.ini listing your target EC2 instance. (Obtain the EC2 public IP from the output of Terraform.)

```
[web]
ec2-instance ansible_host=<EC2_PUBLIC_IP> ansible_user=ubuntu
ansible_ssh_private_key_file=~/.ssh/my-aws-key.pem
```

Note:

- Replace <EC2_PUBLIC_IP> with the actual public IP address.
- Ensure your SSH key (used for AWS) is available at the specified location.

Step 2: Write an Ansible Playbook
Create a playbook named setup.yml to configure your EC2 instance. This playbook will install Docker, pull your Dockerized Python app, and run it.

```
- name: Configure EC2 instance for Python App
  hosts: web
  become: true
```

```
 tasks:
  - name: Update apt cache
    apt:
      update_cache: yes

  - name: Install Docker
    apt:
      name: docker.io
      state: present

  - name: Ensure Docker service is running
    service:
      name: docker
      state: started
      enabled: yes

  - name: Pull Docker image for Python app
    command: docker pull my-python-app:latest

  - name: Run Docker container for Python app
    command: docker run -d -p 80:80 my-python-app:latest
```

Assumption:

- A Docker image named my-python-app:latest is available in your container registry or built locally.

Step 3: Run the Ansible Playbook
Run the playbook from your local machine:

```
ansible-playbook -i inventory.ini setup.yml
```

*This command connects to your EC2 instance and executes the playbook tasks.

Step 4: Automate Ansible via GitHub Actions
You can also integrate Ansible into your CI/CD pipeline. Create a workflow file at .github/workflows/ansible.yml:

```
name: Ansible Deployment

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  ansible:
    runs-on: ubuntu-latest
```

```
steps:
 - name: Checkout Code
  uses: actions/checkout@v3

 - name: Set up Python
  uses: actions/setup-python@v4
  with:
   python-version: '3.x'

 - name: Install Ansible
  run: pip install ansible

 - name: Run Ansible Playbook
  env:
   ANSIBLE_HOST_KEY_CHECKING: 'False'
   # You can add SSH key setup here if needed.
  run: ansible-playbook -i inventory.ini setup.yml
```

Important:

- You may need to configure SSH keys or use an Ansible connection method that works in your CI environment. GitHub Secrets can help securely store SSH keys.

Activity 14: Demonstrate and perform Configuration Management in CI/CD Operations using Ansible

Best Practices for IaC

https://learn.microsoft.com/en-us/devsecops/playbook/articles/infrastructure/best-practices-infrastructure-pipelines

Containerization and Orchestration

Introduction to Containerization with Docker

What is Containerization?

Containerization packages an application and its dependencies into a single lightweight unit (a container) that runs consistently on any system. Unlike traditional virtual machines (VMs), containers share the host OS kernel, making them faster to start and more resource efficient.

What is Docker?

Docker is the most popular containerization platform. It simplifies creating, deploying, and managing containers using images, containers, volumes, and networks.

Docker Concepts: Images, Containers, Volumes, Networks

Images:

- Read-only templates used to create containers. They contain the code, runtime, libraries, and environment variables needed for your application.

Containers:

- Running instances of Docker images. They are isolated and have their own filesystem, processes, and network stack.

Volumes:

- Persistent storage mechanisms for containers. Volumes allow you to store data outside the container's writable layer, so data isn't lost when the container stops.

Networks:

- Allow containers to communicate with each other and with external systems. Docker provides several networking options like bridge, host, and overlay networks.

Working with Images and Containers

List Available Images:

```
docker images
```

This lists all images you have locally.

Run an Interactive Container:

```
docker run -it ubuntu bash
```

*This command pulls the ubuntu image (if not present) and runs it in interactive mode (-it), starting a Bash shell inside the container.

Inside the container:

You can run commands like ls, apt update, etc.

Exit the container:

Type exit to return to your host system.

Using Volumes

Create a Named Volume:

```
docker volume create mydata
```

*This creates a persistent volume called mydata.

Run a Container with a Volume:

```
docker run -d -v mydata:/data --name volume-demo busybox tail -f /dev/null
```

*This runs a busybox container in detached mode (-d), mounts the mydata volume to the /data directory inside the container, and names it volume-demo.

Test the volume:
Open an interactive shell:

```
docker exec -it volume-demo sh
```

Inside the container, create a file in /data:

```
echo "Persistent data" > /data/info.txt
exit
```

*Now, if you restart or remove the container and attach the volume to a new container, the file will persist.

Working with Networks
Create a Custom Network:

```
docker network create mynetwork
```

*This creates a new Docker bridge network called mynetwork.

Run Containers on the Same Network:

```
docker run -d --name container1 --network mynetwork busybox tail -f /dev/null
docker run -d --name container2 --network mynetwork busybox tail -f /dev/null
```

*Both containers are on the mynetwork network and can communicate with each other using container names.

Test Communication Between Containers:
Enter one container:

```
docker exec -it container1 sh
```

Ping the other container:

```
ping container2
```

*You should see successful ping responses, demonstrating inter-container communication.

Creating and Managing Docker Containers

Containers are the runtime instances of Docker images. You can create, start, stop, remove, and inspect containers.

Run a Container in Detached Mode:

```
docker run -d --name mycontainer nginx
```

*This starts an nginx web server in detached mode (-d) with the name mycontainer.

List Running Containers:

```
docker ps
```

*This lists all currently running containers.

View All Containers (including stopped ones):

```
docker ps -a
```

Stop a Container:

```
docker stop mycontainer
```

Start a Container:

```
docker start mycontainer
```

View Container Logs:

```
docker logs mycontainer
```

Remove a Container:
First, stop it (if running):

```
docker stop mycontainer
```

Then remove it:

```
docker rm mycontainer
```

Setting up Docker in AWS EC2 Instances

For Amazon Linux 2

a) Launch an EC2 Instance. Choose the Amazon Linux 2 AMI when launching your instance.
b) Connect to Your Instance. Use SSH to connect to your instance.
c) Update Your Packages

```
sudo yum update -y
```

d) Install Docker. Amazon Linux 2 provides Docker via the amazon-linux-extras repository:

```
sudo amazon-linux-extras install docker -y
```

e) Start the Docker Service

```
sudo service docker start
```

f) Add Your User to the Docker Group. This allows you to run Docker commands without using sudo:

```
sudo usermod -a -G docker ec2-user
```

Note: After running this command, you'll need to log out and log back in for the changes to take effect.

g) Test Your Docker Installation

```
docker run hello-world
```

*This command downloads and runs a test container. If Docker is installed correctly, you'll see a success message.

For Ubuntu

a) Launch an EC2 Instance. Choose an Ubuntu AMI when launching your instance.
b) Connect to Your Instance. Use SSH to connect.

c) Update Your Packages

```
sudo apt update
```

d) Install Docker

```
sudo apt install docker.io -y
```

e) Start and Enable Docker

```
sudo systemctl start docker
sudo systemctl enable docker
```

f) Add Your User to the Docker Group

```
sudo usermod -aG docker $USER
```

Note: Log out and log back in (or reboot) to apply the group changes.

g) Test Your Docker Installation

```
docker run hello-world
```

Additional Considerations

Security Groups:

- Make sure your EC2 instance's security groups allow the necessary traffic if your containers need to be accessible from the internet.

Docker Compose:

- If you plan to use Docker Compose, you can install it by following the official Docker Compose installation guide.

IAM Roles and ECS:

- For production environments or more advanced setups, consider using Amazon ECS (Elastic Container Service) or EKS (Elastic Kubernetes Service) for orchestrating containers.

Building and Sharing Docker Images

Building custom images allows you to package your application and share it with others.

Example: Building a Simple Web Application Image

a) Create a Project Directory:

```
mkdir my-webapp
cd my-webapp
```

b) Create an Application File (For example, a simple Python HTTP server):
Create app.py:

```
# app.py
from http.server import SimpleHTTPRequestHandler, HTTPServer

PORT = 8000
```

```
class Handler(SimpleHTTPRequestHandler):
  pass

if __name__ == "__main__":
  server = HTTPServer(('0.0.0.0', PORT), Handler)
  print(f"Server running on port {PORT}")
  server.serve_forever()
```

c) Create a Requirements File (if needed)
For this simple app, you might not need one, but if you have dependencies, list them in a requirements.txt.

d) Build a Docker Image
We'll write a Dockerfile in the next section. For now, let's assume you've written it and now build the image:

```
docker build -t my-webapp:latest .
```

*This command builds an image tagged my-webapp:latest using the Dockerfile in the current directory.

e) Run Your Image as a Container:

```
docker run -d -p 8000:8000 --name webapp-container my-webapp:latest
```

*This maps port 8000 of the container to port 8000 on your host.

f) Test Your Application:
Open your browser and navigate to http://localhost:8000 (or use curl http://localhost:8000) to see your application in action.

Writing Dockerfiles

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. It automates the image creation process.

Example: Creating a Dockerfile for a Python Web App

a) Create a Dockerfile in Your Project Directory:

```
# Use an official Python runtime as a parent image
FROM python:3.9-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Expose the port the app runs on
EXPOSE 8000
```

```
# Define environment variable
ENV PYTHONUNBUFFERED=1

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

Explanation of Each Instruction:

- FROM: Specifies the base image.
- WORKDIR: Sets the working directory inside the container.
- COPY: Copies files from your host into the container.
- EXPOSE: Documents the port on which the container listens.
- ENV: Sets environment variables.
- CMD: Specifies the command to run when the container starts.

b) Build the Image:

```
docker build -t my-webapp:latest .
```

c) Run the Container:

```
docker run -d -p 8000:8000 --name webapp-container my-webapp:latest
```

d) Verify It's Running: Check the logs:

```
docker logs webapp-container
```

Docker Registries and Image Management

Docker registries (such as Docker Hub) allow you to store and share Docker images. You can push your images to a registry so others can pull and run them.

Example: Pushing an Image to Docker Hub

a) Create a Docker Hub Account (If you haven't already, sign up for Docker Hub)

b) Log In to Docker Hub via the CLI:

```
docker login
```

c) Enter your Docker Hub username and password when prompted.

d) Tag Your Image Appropriately:
Suppose your Docker Hub username is yourusername. Tag your image as follows:

```
docker tag my-webapp:latest yourusername/my-webapp:latest
```

e) Push the Image to Docker Hub:

```
docker push yourusername/my-webapp:latest
```

*This command uploads your image to Docker Hub under your account.

f) Pulling the Image on Another Machine:

On any machine with Docker installed, you can pull your image with:

```
docker pull yourusername/my-webapp:latest
```

Then run it:

```
docker run -d -p 8000:8000 yourusername/my-webapp:latest
```

g) Managing Images Locally:
List Images:

```
docker images
```

Remove an Image:

```
docker rmi yourusername/my-webapp:latest
```

Activity 15: Demonstrate and perform Containerization in CI/CD Operations using Docker

Orchestrating Containers with Kubernetes

For this example, we'll assume you already have an AWS EC2 instance with Docker installed and that you've built your Flask application's Docker image (using the Python script and Dockerfile from the previous example). We will use kind ("Kubernetes IN Docker") to run a local single-node Kubernetes cluster on your EC2 instance. (Alternatively, if you have access to a managed Kubernetes service like AWS EKS, the Kubernetes objects and YAML files below remain essentially the same.)

Note: Running Kubernetes on an EC2 instance can be achieved in several ways. Using kind is a lightweight option for learning and testing.

Kubernetes is an open-source platform designed to automate the deployment, scaling, and management of containerized applications. It groups containers into logical units (called pods) for easy management and enables features such as load balancing, rolling updates, and self-healing.

Kubernetes Architecture and Components

Before diving into the hands-on steps, here's a brief overview of key Kubernetes components:

Control Plane Components:

- API Server: The front end for the Kubernetes control plane that exposes the Kubernetes API.
- etcd: A key-value store used as Kubernetes' backing store for all cluster data.
- Scheduler: Assigns work (pods) to worker nodes based on resource availability.
- Controller Manager: Runs controller processes (e.g., replication, endpoints).

Node (Worker) Components:

- Kubelet: An agent that runs on each node and ensures that containers are running in a pod.
- Kube-proxy: Maintains network rules on nodes and enables communication between pods or between pods and the external world.

Other Concepts:

- Pods: The smallest deployable units in Kubernetes, which encapsulate one or more containers.
- Services: An abstraction that defines a logical set of pods and a policy to access them (e.g., load balancing).
- Ingress: Manages external access to services, typically HTTP.

Deploying Applications in Kubernetes

Below is a step-by-step example of a very simple Flask application designed for learning Docker and Kubernetes on an AWS EC2 instance. We'll cover:

1) Creating the Flask application code.
2) Writing a requirements file.
3) Creating a Dockerfile to containerize the application.
4) Building and running the Docker image.
5) (Optionally) Deploying the container in Kubernetes.

1. Create a Simple Flask App

Create a directory for your project and inside it create a file named app.py with the following content:

```
# app.py
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
  return "Hello, World from Flask running in Docker and Kubernetes on AWS EC2!"

if __name__ == '__main__':
  # Bind to 0.0.0.0 to ensure the container is accessible externally
  app.run(host='0.0.0.0', port=5000)
```

*This application listens on port 5000 and returns a simple greeting.

2. Create a Requirements File

Create a file named requirements.txt in the same directory with the following content:

```
flask
```

*This file tells Docker which Python packages to install.

3. Create a Dockerfile

In the same project directory, create a file named Dockerfile (with no extension) and add the following content:

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim

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

```
# Copy the current directory contents into the container at /app
COPY . /app

# Install the Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Expose port 5000 to the host
EXPOSE 5000

# Define the command to run your application
CMD ["python", "app.py"]
```

This Dockerfile does the following:

a) Uses a lightweight Python 3.8 image.
b) Copies your application code into the container.
c) Installs the required Python packages.
d) Exposes port 5000.
e) Runs the Flask app when the container starts.

4. Build and Run the Docker Image

Step 1: Build the Docker Image

On your AWS EC2 instance (or local machine), open a terminal in your project directory and build the Docker image:

```
docker build -t simple-flask-app .
```

*This command creates a Docker image named simple-flask-app using the Dockerfile in the current directory.

Step 2: Run the Docker Container

Run a container from your new image:

```
docker run -d -p 5000:5000 --name flask-container simple-flask-app
```

-d runs the container in detached mode.
-p 5000:5000 maps port 5000 in the container to port 5000 on the EC2 host.
--name flask-container assigns a name to your container.

You can verify that the container is running by executing:

```
docker ps
```

Now, open your browser and navigate to:

```
http://<EC2_PUBLIC_IP>:5000
```

You should see the message:
"Hello, World from Flask running in Docker and Kubernetes on AWS EC2!"

5. Deploying Your Application in Kubernetes

If you're ready to take the next step and learn about Kubernetes orchestration, follow these steps to deploy your containerized app.

Step 1: Prepare Your Docker Image for Kubernetes

If you are using a local Kubernetes cluster (for example, with kind) on your EC2 instance, load your image into the cluster:

```
kind load docker-image simple-flask-app --name your-cluster-name
```

*Replace your-cluster-name with the name of your kind cluster.

Step 2: Create a Kubernetes Deployment

Create a file named deployment.yaml with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: flask-deployment
spec:
 replicas: 2
 selector:
  matchLabels:
   app: flask
 template:
  metadata:
   labels:
    app: flask
  spec:
   containers:
   - name: flask
    image: simple-flask-app
    ports:
    - containerPort: 5000
```

Apply the deployment using:

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

Step 3: Expose the Deployment via a Service

Create a file named service.yaml with the following content:

```
apiVersion: v1
kind: Service
metadata:
 name: flask-service
spec:
 type: NodePort
 selector:
  app: flask
```

```
 ports:
 - protocol: TCP
   port: 80
   targetPort: 5000
   nodePort: 30007  # Optional: You can let Kubernetes choose a port automatically.
```

Apply the service:

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

*Now, you can test your application by accessing the EC2 instance's public IP at the specified node port (e.g., http://<EC2_PUBLIC_IP>:30007).

Final Recap

Flask App:

- ✓ app.py contains a very simple Flask application.
- ✓ requirements.txt lists the dependency (Flask).

Dockerization:

- ✓ A Dockerfile is used to build an image that installs dependencies and runs the Flask app.
- ✓ You built and ran the Docker container on your AWS EC2 instance.

Kubernetes Deployment (Optional):

- ✓ A Deployment YAML file runs multiple replicas of your Flask app.
- ✓ A Service YAML file exposes your application to external traffic.

Kubernetes Services, Pods, and Ingress

Pods:

The smallest deployable unit in Kubernetes. A pod encapsulates one or more containers that share networking and storage resources. When you deploy an application (for example, via a Deployment), Kubernetes creates one or more pods.

Services:

A Service provides a stable, discoverable endpoint to access a group of pods. Services decouple the client from the pod lifecycle. Common service types include:

- ClusterIP: Exposes the service on an internal IP in the cluster.
- NodePort: Exposes the service on a static port on each node's IP.
- LoadBalancer: Provisions an external load balancer (supported in many cloud providers).

Step 1: Create a Service

Let's assume you already have a Deployment named flask-deployment with pods labeled app: flask (as shown in previous examples). Create a service that exposes the application.

Create a file named service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: flask-service
spec:
```

```
 type: NodePort  # Use ClusterIP or LoadBalancer as needed.
 selector:
  app: flask
 ports:
  - protocol: TCP
    port: 80       # Internal service port
    targetPort: 5000   # Port the Flask container is listening on
    nodePort: 30007  # Optional: Node port (if you want a fixed port)
```

Apply the service:

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

Test the service by accessing http://<EC2_PUBLIC_IP>:30007 (or use port-forwarding).

Step 2: Deploy an Ingress Controller

An Ingress Controller manages external access to services in your cluster. The most common open-source solution is ingress-nginx.

Deploy ingress-nginx:

Run this command (for a kind or test cluster):

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/kind/deploy.yaml
```

Verify that the Ingress controller is running:

```
kubectl get pods -n ingress-nginx
```

Step 3: Create an Ingress Resource

Create an ingress resource that maps a hostname or path to your service. Create a file named ingress.yaml:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: flask-ingress
 annotations:
  nginx.ingress.kubernetes.io/rewrite-target: /
spec:
 ingressClassName: nginx
 rules:
 - host: flask.local
  http:
   paths:
   - path: /
     pathType: Prefix
     backend:
      service:
       name: flask-service
```

```
      port:
        number: 80
```

Apply the ingress resource:

```
kubectl apply -f ingress.yaml
```

Testing Ingress:
For local testing, update your /etc/hosts file to map flask.local to your EC2 public IP (or to 127.0.0.1 if port-forwarding). Then open a browser at http://flask.local.

Scaling and Managing Kubernetes Clusters

Manual Scaling

Once your application is deployed, you might want to adjust the number of running pods.

Step 1: Scale a Deployment

Use the kubectl scale command to increase or decrease the number of replicas.
For example, to scale the flask-deployment to 4 replicas:

```
kubectl scale deployment/flask-deployment --replicas=4
```

Verify the change:

```
kubectl get deployments
kubectl get pods
```

Step 2: Rolling Updates

When you need to update your application (for instance, deploying a new version of your container), you can perform a rolling update. Update your Deployment YAML file with the new image tag and then apply the changes:

Edit deployment.yaml (change the image tag, e.g., to simple-flask-app:v2):

```
spec:
 replicas: 4
 template:
  spec:
   containers:
   - name: flask
     image: simple-flask-app:v2
     ports:
     - containerPort: 5000
```

Apply the update:

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

Check rollout status:

```
kubectl rollout status deployment/flask-deployment
```

Auto-scaling, Load Balancing

Auto-scaling with Horizontal Pod Autoscaler (HPA)

Kubernetes can automatically adjust the number of pod replicas based on observed CPU usage or custom metrics.

Step 1: Create an HPA

For example, to automatically scale your flask-deployment between 2 and 10 replicas based on CPU utilization (target 50%):

Ensure your cluster metrics are available:

In many clusters, you'll need the metrics-server deployed. For a kind cluster, install it if not already available:

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

Create the HPA resource:

```
kubectl autoscale deployment flask-deployment --cpu-percent=50 --min=2 --max=10
```

View HPA status:

```
kubectl get hpa
```

Load Balancing

Service Load Balancing:

When you use a Service of type LoadBalancer (on supported cloud providers), Kubernetes automatically provisions an external load balancer that distributes incoming traffic across your pods.

Ingress as a Load Balancer:

The Ingress Controller (like ingress-nginx) can act as a load balancer by routing HTTP(s) traffic to different backend services based on host or path rules.

Example:

If you were running in AWS with a supported Kubernetes service (like EKS), you could change your service type to LoadBalancer:

```
spec:
 type: LoadBalancer
 ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
```

Then apply:

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

*The cloud provider will provision an external load balancer that automatically distributes traffic to your pods.

Managing Stateful and Stateless Applications

Stateless Applications

Stateless applications do not maintain any persistent state between requests. Most web applications (like our Flask app) are stateless; if a pod fails, a new one can be created without worrying about lost data.

Deployment:

- ✓ Use a standard Deployment. We already used this for our Flask app.

Stateful Applications

Stateful applications require persistent data storage and stable network identities. Examples include databases and message queues.

Key Kubernetes Objects:

- StatefulSet: Manages the deployment and scaling of stateful applications. Each pod in a StatefulSet gets a unique, stable network identity.
- PersistentVolume (PV) and PersistentVolumeClaim (PVC): Used for persistent storage that outlives pod lifecycles.

Step 1: Example of a Simple Stateful Application

Let's create a simple example using a StatefulSet. In this example, we'll deploy a basic application (e.g., a simple database like MySQL) with persistent storage.

Create a PersistentVolume Claim (PVC):

Create a file named mysql-pvc.yaml:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: mysql-pvc
spec:
 accessModes:
  - ReadWriteOnce
 resources:
  requests:
   storage: 1Gi
```

Apply the PVC:

```
kubectl apply -f mysql-pvc.yaml
```

Create a StatefulSet for MySQL:
Create a file named mysql-statefulset.yaml:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
 name: mysql
spec:
 serviceName: "mysql"
 replicas: 1
 selector:
  matchLabels:
   app: mysql
 template:
  metadata:
   labels:
    app: mysql
  spec:
   containers:
   - name: mysql
    image: mysql:5.7
    ports:
     - containerPort: 3306
    env:
     - name: MYSQL_ROOT_PASSWORD
      value: "example"
    volumeMounts:
     - name: mysql-storage
      mountPath: /var/lib/mysql
 volumeClaimTemplates:
 - metadata:
   name: mysql-storage
  spec:
   accessModes: ["ReadWriteOnce"]
   resources:
    requests:
     storage: 1Gi
```

Apply the StatefulSet:

```
kubectl apply -f mysql-statefulset.yaml
```

Create a Headless Service for the StatefulSet:
Create a file named mysql-headless-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: mysql
spec:
  clusterIP: None
  selector:
    app: mysql
  ports:
    - port: 3306
      targetPort: 3306
```

Apply the headless service:

```
kubectl apply -f mysql-headless-service.yaml
```

Note: StatefulSets are used when each pod requires a unique identity and persistent storage. For stateless applications (like our Flask app), Deployments and ReplicaSets are sufficient.

Activity 16: Demonstrate and perform Container Orchestration using Kubernetes

Monitoring, Logging, and DevOps in the Cloud

Importance of Monitoring in a DevOps Environment

In a modern DevOps workflow, monitoring is essential because it helps you:

- ✓ Detect issues early: Quickly identify and resolve performance bottlenecks, errors, or failures.
- ✓ Ensure availability: Keep your applications and infrastructure running reliably.
- ✓ Measure performance: Gather metrics on resource usage (CPU, memory, latency) and business KPIs.
- ✓ Plan capacity and scalability: Use data to plan scaling events or to allocate resources effectively.
- ✓ Improve deployments: Monitor deployments and rolling updates to catch regressions immediately.

Monitoring Tools (Prometheus, Grafana, ELK Stack)

Prometheus & Grafana: Collecting Metrics and Visualizing Data

- Prometheus scrapes metrics from your applications and infrastructure.
- Grafana provides dashboards to visualize those metrics.

Step 1: Running Prometheus and Grafana with Docker
For a quick demo, you can run Prometheus and Grafana as Docker containers on your AWS EC2 instance.

Create a Prometheus Configuration File
Create a file named prometheus.yml in a new directory (e.g., monitoring/):

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'flask-app'
    static_configs:
```

```
    - targets: ['host.docker.internal:5000']  # Adjust if using EC2's IP or container networking
```

Note: If you're running on EC2 and not Docker-for-Desktop, replace host.docker.internal with the appropriate hostname or IP address where your Flask app is accessible.

Start Prometheus Container
Run the following command (from within your monitoring/ directory):

```
docker run -d \
 -p 9090:9090 \
 -v "$(pwd)/prometheus.yml:/etc/prometheus/prometheus.yml" \
 --name prometheus \
 prom/prometheus
```

Start Grafana Container
Run Grafana on port 3000:

```
docker run -d \
 -p 3000:3000 \
 --name=grafana \
 grafana/grafana
```

Configure Grafana

a) Open your browser and navigate to http://<EC2_PUBLIC_IP>:3000.
b) Log in (default username/password: admin/admin).
c) Add Prometheus as a data source using the URL http://<EC2_PUBLIC_IP>:9090 (or use http://prometheus:9090 if using an internal Docker network).
d) Import or create dashboards to visualize metrics.

Activity 17: Demonstrate and perform Monitoring and Logging in CI/CD Operations using Prometheus and Grafana

ELK Stack: Centralized Logging

- Elasticsearch stores logs.
- Logstash can process and transform log data.
- Kibana visualizes log data and allows you to query it interactively.

Step 1: Running a Basic ELK Stack Using Docker
For a basic example, you can use pre-built Docker images to set up Elasticsearch and Kibana. (In production, you'd often include Logstash or Filebeat to ship logs.)

Run Elasticsearch

```
docker run -d \
 -p 9200:9200 \
 -p 9300:9300 \
 --name elasticsearch \
 -e "discovery.type=single-node" \
 docker.elastic.co/elasticsearch/elasticsearch:7.17.0
```

Run Kibana

```
docker run -d \
 -p 5601:5601 \
 --name kibana \
 --link elasticsearch:elasticsearch \
 docker.elastic.co/kibana/kibana:7.17.0
```

Test Your Setup

```
Access Elasticsearch at http://<EC2_PUBLIC_IP>:9200 to see cluster info.
Access Kibana at http://<EC2_PUBLIC_IP>:5601 to begin exploring logs.
```

Step 2: Sending Application Logs to ELK
For a basic demonstration, you can have your Flask app write logs to stdout (which Docker collects). Then use a lightweight log shipper (like Filebeat) or configure Docker's logging driver to send logs to Elasticsearch.

For example, to run your Flask container with JSON-formatted logs:

```
docker run -d \
 -p 5000:5000 \
 --log-driver json-file \
 --name flask-container \
 simple-flask-app
```

Next, you could deploy Filebeat (or use Logstash) to read these logs and forward them to Elasticsearch. Detailed configuration is beyond this simple demo, but the core idea is:

- Filebeat reads logs from Docker's log files.
- Filebeat ships logs to Elasticsearch.
- Kibana displays them for analysis.

Setting Up Application and Infrastructure Monitoring

Application Monitoring

Instrument your Flask app:

You can integrate libraries (e.g., Prometheus client or an APM agent) into your app to expose metrics or performance data.

For example, install the Prometheus client in your Flask app:

```
pip install prometheus_client
```

Then add a basic metrics endpoint to your app.py:

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

app = Flask(__name__)
```

```
# A simple counter metric
request_counter = Counter('http_requests_total', 'Total HTTP Requests')

@app.route('/')
def hello():
  request_counter.inc()  # Increment the counter for each request
  return "Hello, World from Flask!"

@app.route('/metrics')
def metrics():
  return Response(generate_latest(), mimetype=CONTENT_TYPE_LATEST)

if __name__ == '__main__':
  app.run(host='0.0.0.0', port=5000)
```

With this endpoint, Prometheus can scrape your metrics from /metrics.

Infrastructure Monitoring

Kubernetes Monitoring:

In a Kubernetes environment, you can deploy the Prometheus Operator or use pre-built Prometheus Helm charts to collect metrics from:

- The Kubernetes API server, kubelets, and nodes
- System components and custom application metrics

Node Exporter:

You can run the Node Exporter on each EC2 instance to collect host-level metrics and have Prometheus scrape those.

Centralized Logging with the ELK Stack

A simple centralized logging workflow using Docker might look like this:

Step 1: Configure Filebeat (Optional Simple Setup)

Create a Filebeat configuration file (e.g., filebeat.yml):

```
filebeat.inputs:
 - type: container
  paths:
   - /var/lib/docker/containers/*/*.log
  processors:
   - add_docker_metadata: ~

output.elasticsearch:
 hosts: ["<EC2_PUBLIC_IP>:9200"]
```

Run Filebeat as a Docker container (mounting the Docker logs directory):

```
docker run -d \
 --name=filebeat \
 --user=root \
 --volume="/var/lib/docker/containers:/var/lib/docker/containers:ro" \
 --volume="$(pwd)/filebeat.yml:/usr/share/filebeat/filebeat.yml:ro" \
 docker.elastic.co/beats/filebeat:7.17.0 filebeat -e -strict.perms=false
```

View Logs in Kibana:

- After a few minutes, open Kibana (http://<EC2_PUBLIC_IP>:5601) and use the Discover tab to query and visualize your container logs.

Note: This is a very basic setup. In production, you would fine-tune the Filebeat configuration and consider using Logstash for additional processing.

Activity 18: Demonstrate and perform Monitoring and Logging in CI/CD Operations using ELK Stack

Application Performance Monitoring (APM)

For a simple APM demo, we can integrate an APM agent into our Flask app. One popular choice is the Elastic APM Python agent. This lets you capture performance metrics, errors, and traces.

Step 1: Set Up an APM Server
Run the Elastic APM Server as a Docker Container:

```
docker run -d \
 -p 8200:8200 \
 --name=apm-server \
 -e "apm-server.host=0.0.0.0:8200" \
 docker.elastic.co/apm/apm-server:7.17.0
```

Verify the APM Server

- Open http://<EC2_PUBLIC_IP>:8200 to see that the server is running.

Step 2: Instrument the Flask App with Elastic APM
Install the APM Agent in Your Flask App
In your Flask project, install the agent:

```
pip install elastic-apm[flask]
```

Update Your app.py to Integrate APM
Add the following at the top of your app.py (adjust the server URL as needed):

```
from elasticapm.contrib.flask import ElasticAPM

app = Flask(__name__)
```

```
# Configure the Elastic APM agent
app.config['ELASTIC_APM'] = {
  'SERVICE_NAME': 'flask-app',
  'SERVER_URL': 'http://<EC2_PUBLIC_IP>:8200',  # Replace with your APM server URL
  'DEBUG': True,
}

# Initialize APM
apm = ElasticAPM(app)
```

*Keep the rest of your Flask endpoints as before. The agent will automatically capture errors and performance data.

Deploy and Test

- Rebuild your Docker image and restart your container.
- Trigger some requests and then open Kibana (if you have integrated the APM UI in Kibana via the ELK stack) or review the APM Server logs to see incoming performance data.

Cloud-Native DevOps: Serverless, Containers, and Managed Services

Cloud-Native DevOps focuses on:

- ✓ Rapid Iteration and Deployment: Utilizing agile practices with continuous integration and continuous delivery (CI/CD).
- ✓ Microservices Architecture: Breaking applications into smaller, loosely coupled services.
- ✓ Elastic Scalability: Automatically adjusting resources based on demand.
- ✓ Operational Efficiency: Offloading routine tasks to managed services, so teams can focus on core business logic.

By designing applications with these principles in mind, you can achieve robust systems that automatically scale and are easier to manage.

Serverless Computing

Serverless Computing (also known as Function-as-a-Service or FaaS) lets you run code without provisioning or managing servers. The cloud provider handles the underlying infrastructure, scaling your application automatically in response to events.

Key Characteristics

- ✓ No Server Management: Developers write code, and the provider takes care of running it.
- ✓ Automatic Scaling: Functions scale based on the event load.
- ✓ Pay-as-You-Go Pricing: You pay only for the compute time you use.
- ✓ Event-Driven: Ideal for handling intermittent or unpredictable workloads.

Example: Deploying a Simple AWS Lambda Function
Let's create a basic serverless function on AWS Lambda that responds to an API request.

Step 1: Write Your Function Code
Create a file named lambda_function.py:

```
def lambda_handler(event, context):
  return {
    'statusCode': 200,
    'body': 'Hello, Cloud-Native World from AWS Lambda!'
  }
```

Step 2: Package and Deploy the Function

a) Log in to the AWS Management Console and navigate to the AWS Lambda service.
b) Create a New Function:
    - Choose "Author from scratch."
    - Give your function a name (e.g., HelloLambda).
    - Select Python 3.x as the runtime.
c) Upload Your Code:
    - In the function code section, you can either paste your code or upload a ZIP file containing lambda_function.py.
d) Configure a Trigger:
    - For example, add an API Gateway trigger so the function can be invoked via HTTP.
e) Test Your Function:
    - Use the built-in test feature in AWS Lambda or invoke it via the API Gateway endpoint.
f) Outcome:
    - You now have a serverless function that automatically scales and only charges you based on usage.

Containers
Containers encapsulate an application and its dependencies into a single, portable unit that can run consistently in any environment. Docker is the most popular containerization platform, and Kubernetes is widely used to orchestrate containerized applications.

Key Characteristics

- ✓ Consistency: Run the same container image on your local machine, in test, and in production.
- ✓ Portability: Containers can run on any infrastructure—on-premises, in the cloud, or hybrid.
- ✓ Microservices-Friendly: Containers make it easier to build and manage microservices architectures.
- ✓ Resource Isolation: Containers share the host OS kernel but run isolated from one another.

Example: Containerizing a Simple Flask App

Assume you have a basic Flask app (as we created in a previous lesson). Here's a quick refresher:

Step 1: Flask App Code (app.py)

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
   return "Hello, World from Flask running in a container!"

if __name__ == '__main__':
   app.run(host='0.0.0.0', port=5000)
```

Step 2: Create a Dockerfile

Create a file named Dockerfile in the same directory:

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir flask

# Expose port 5000 to the host
EXPOSE 5000

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

Step 3: Build and Run the Container

Build the Docker Image:

```
docker build -t simple-flask-app .
```

Run the Docker Container:

```
docker run -d -p 5000:5000 --name flask-container simple-flask-app
```

Test the Application: Navigate to http://<EC2_PUBLIC_IP>:5000 to see your Flask app running.

Outcome:

*You now have a containerized application that can be deployed consistently across different environments.

Managed Services
Managed Services are cloud-based services that offload operational tasks (such as maintenance, scaling, security, and backups) to the provider. This allows you to focus on developing your application rather than managing infrastructure.

Key Characteristics
- ✓ Reduced Operational Overhead: Cloud providers manage routine tasks.
- ✓ Built-In Scalability and High Availability: Services are designed to scale automatically.
- ✓ Security and Compliance: Providers handle patching, backups, and security updates.
- ✓ Cost Efficiency: You pay for what you use and avoid the costs associated with over-provisioning.

Examples of Managed Services
- ✓ Databases: AWS RDS (MySQL, PostgreSQL), DynamoDB, Google Cloud SQL, Azure SQL Database.
- ✓ Storage: AWS S3, Google Cloud Storage, Azure Blob Storage.
- ✓ Messaging: AWS SQS, Google Cloud Pub/Sub, Azure Service Bus.
- ✓ Container Services: AWS Fargate (serverless containers), Amazon EKS (managed Kubernetes), Google Kubernetes Engine (GKE), Azure Kubernetes Service (AKS).

Example: Using AWS RDS with Your Application
Let's say your application needs a database. Instead of managing your own MySQL instance, you can use AWS RDS.

Step 1: Create an RDS Instance
a) Log in to the AWS Management Console and navigate to RDS.
b) Launch a New Database Instance:
   - Choose MySQL as the engine.
   - Select a free-tier or appropriate instance type.
   - Configure the database settings (username, password, etc.).
c) Configure Security:
   - Ensure your EC2 instance (or other compute resources) can access the RDS instance by setting up the correct VPC security groups.
d) Connect Your Application:
   - In your application configuration, use the RDS endpoint, username, and password to connect to the database.

Outcome:
*Your application now leverages a managed database service that handles backups, scaling, and patching.

Integrating Serverless, Containers, and Managed Services in a Cloud-Native DevOps Workflow
Imagine a modern application architecture that combines all three elements:

Containers for the Core Application:
- Your microservices (e.g., a Flask app) run in Docker containers orchestrated by Kubernetes (or a managed service like AWS EKS).

Serverless for Event-Driven Processing:

- AWS Lambda functions process asynchronous events (e.g., image processing, notifications) without requiring dedicated servers.

Managed Services for Supporting Infrastructure:

- AWS RDS for relational data storage.
- AWS S3 for static file storage.
- AWS SQS for messaging and decoupling services.

Example Workflow:

CI/CD Pipeline:

- Code is pushed to a Git repository.
- A CI/CD system (such as AWS CodePipeline, GitLab CI, or Jenkins) builds the Docker image and deploys it to a managed Kubernetes cluster.

Application Execution:

- The containerized Flask app handles web requests.
- For time-consuming tasks (e.g., sending emails or processing images), the app triggers a serverless function (AWS Lambda) via an API or event (such as an S3 upload event).

Data Management:

- The application stores user data in AWS RDS.
- Files are stored in AWS S3, and messaging between components is managed via AWS SQS.

Monitoring and Scaling:

- The container platform (e.g., EKS) scales based on load.
- AWS Lambda automatically scales with demand.
- Managed services handle scaling, backups, and performance optimizations automatically.

Outcome: This integrated architecture reduces the operational burden on your team while providing a highly scalable and resilient application environment.

Using Jenkins

Jenkins is an open-source automation server widely used for continuous integration (CI) and continuous delivery (CD). It helps automate parts of the software development process, enabling development teams to build, test, and deploy their applications quickly and reliably. Here are some key points about Jenkins:

Automation Server:

Jenkins acts as a central hub where you can automate tasks such as compiling code, running tests, and deploying applications. It can trigger these tasks automatically whenever code is committed to a version control system (like Git).

Continuous Integration and Continuous Delivery:
By integrating with source code repositories and various testing frameworks, Jenkins continuously builds and tests code changes. This ensures that integration issues are detected early. It also helps streamline the process of releasing new software versions.

Plugin Ecosystem:
One of Jenkins' major strengths is its extensive plugin system. Plugins extend Jenkins' functionality to integrate with almost any tool in the software development and deployment process—ranging from version control systems and build tools to deployment platforms and monitoring systems.

Flexible Pipeline Configuration:
Jenkins allows you to define your build, test, and deployment pipelines as code (using a Jenkinsfile), making it easier to version control and maintain the automation processes. Pipelines can be configured in a declarative or scripted format, depending on your preference.

Community and Support:
Being open-source and widely adopted, Jenkins has a large community of users and contributors. This means a wealth of tutorials, plugins, and community support is available.

In this example, the pipeline will:

1) Checkout code from a GitHub repository.
2) Run tests using pytest.
3) Build a Docker image for the Flask app.
4) Push the image to a container registry.
5) Deploy the updated application to a Kubernetes cluster.
6) (Optionally) Have Prometheus and Grafana monitor the deployed app.

You can run Jenkins on an AWS EC2 instance. The instructions below provide pointers for both Ubuntu and RHEL/Amazon Linux 2.

1. Setting Up Jenkins on an AWS EC2 Instance

For Ubuntu:

a) Launch an Ubuntu EC2 Instance (ensure security groups allow ports 8080 for Jenkins, 5000 for your app, etc.).
b) Install Java (required by Jenkins):

```
sudo apt update
sudo apt install openjdk-11-jdk -y
```

c) Add the Jenkins Repository and Install Jenkins:

```
wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
sudo apt update
sudo apt install jenkins -y
```

d) Start and Enable Jenkins:

```
sudo systemctl start jenkins
sudo systemctl enable jenkins
```

e) Access Jenkins:

Open a browser and navigate to http://<EC2_PUBLIC_IP>:8080. Follow the setup wizard (you may need to copy the initial admin password from /var/lib/jenkins/secrets/initialAdminPassword).

For RHEL/Amazon Linux 2:

a) Launch a RHEL/Amazon Linux 2 EC2 Instance (ensure ports are open as needed).

b) Install Java:

```
sudo yum update -y
sudo yum install java-11-openjdk-devel -y
```

c) Add the Jenkins Repository: Create a file /etc/yum.repos.d/jenkins.repo with the following content:

```
[jenkins]
name=Jenkins-stable
baseurl=https://pkg.jenkins.io/redhat-stable/
gpgcheck=1
```

Import the GPG key and Install Jenkins:

```
sudo rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io.key
sudo yum install jenkins -y
```

Start and Enable Jenkins:

```
sudo systemctl start jenkins
sudo systemctl enable jenkins
```

Access Jenkins:

Navigate to http://<EC2_PUBLIC_IP>:8080 and complete the setup wizard.

2. Repository Setup and Git/GitHub Integration

Create (or update) a GitHub repository that contains your project files. A sample project structure might look like this:

```
.
├── app.py                  # Your Flask app code
├── Dockerfile              # Docker build instructions
├── requirements.txt        # Python dependencies (includes flask and any test libraries)
├── tests/                  # Directory for pytest test files (e.g., test_app.py)
├── k8s/                    # Kubernetes manifests (deployment.yaml, service.yaml, etc.)
└── Jenkinsfile             # Your pipeline definition
```

In Jenkins, install and configure the Git plugin. Then, create a new Pipeline job that points to your GitHub repository (using webhooks for automatic triggering is recommended).

3. Building the Jenkins Pipeline

Pipeline Overview

Your Jenkins pipeline (defined in a Jenkinsfile) will typically include stages such as:

1) Checkout: Pull code from GitHub.
2) Test: Run pytest to verify application correctness.
3) Build: Create a Docker image.
4) Push: Upload the Docker image to a registry (for example, Docker Hub or AWS ECR).
5) Deploy: Apply Kubernetes manifests to update the deployment.
6) Notify/Monitor: (Optionally) trigger notifications or record metrics for Prometheus/Grafana.

Prerequisites on the Jenkins Host

Docker: Install Docker on the Jenkins server so that the pipeline can build images.

For Ubuntu:

```
sudo apt install docker.io -y
sudo usermod -aG docker jenkins
```

For RHEL/Amazon Linux 2:

```
sudo amazon-linux-extras install docker -y
sudo systemctl start docker
sudo usermod -aG docker jenkins
```

Restart Jenkins after modifying group membership.

kubectl: Install and configure kubectl on the Jenkins server with access to your Kubernetes cluster.

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
chmod +x kubectl
sudo mv kubectl /usr/local/bin/
```

Configure your kubeconfig file (usually placed in /var/lib/jenkins/.kube/config) so that Jenkins can run kubectl apply commands.

Credentials:

Configure credentials in Jenkins (for GitHub, Docker registry, and Kubernetes if necessary) using the Credentials Manager.

4. Sample Jenkinsfile

Below is an example Jenkinsfile using the declarative pipeline syntax (groovy) that demonstrates the overall CI/CD flow:

```
pipeline {
  agent any

  environment {
    // Set environment variables
    DOCKER_IMAGE = "yourdockerhubusername/simple-flask-app"
    REGISTRY_CREDENTIALS = "dockerhub-credentials"  // ID for your Docker Hub credentials in Jenkins
    KUBE_CONFIG = credentials('kubeconfig')        // Jenkins credential for kubeconfig (if needed)
  }

  stages {
    stage('Checkout') {
      steps {
        echo 'Checking out code from GitHub...'
        git branch: 'main', url: 'https://github.com/yourusername/your-flask-repo.git'
      }
    }
    stage('Test') {
      steps {
        echo 'Running tests with pytest...'
        sh 'pip install -r requirements.txt'
        sh 'pytest'
      }
    }
    stage('Build Docker Image') {
      steps {
        echo 'Building Docker image...'
        sh "docker build -t ${DOCKER_IMAGE}:latest ."
      }
    }
    stage('Push Docker Image') {
      steps {
        echo 'Pushing Docker image to registry...'
        withCredentials([usernamePassword(credentialsId: env.REGISTRY_CREDENTIALS,
usernameVariable: 'DOCKER_USER', passwordVariable: 'DOCKER_PASS')]) {
          sh """
            echo "$DOCKER_PASS" | docker login -u "$DOCKER_USER" --password-stdin
            docker push ${DOCKER_IMAGE}:latest
          """
        }
      }
    }
    stage('Deploy to Kubernetes') {
```

```
        steps {
            echo 'Deploying application to Kubernetes...'
            // If your manifests are under k8s/ directory
            dir('k8s') {
                // Update image in deployment.yaml if needed (or use a templating tool)
                sh "sed -i 's|image: .*|image: ${DOCKER_IMAGE}:latest|g' deployment.yaml"
                sh 'kubectl apply -f deployment.yaml'
                sh 'kubectl apply -f service.yaml'
            }
        }
    }
  }

  post {
    success {
        echo 'CI/CD Pipeline completed successfully.'
        // Optional: Send a notification or trigger Prometheus alerting here
    }
    failure {
        echo 'CI/CD Pipeline failed.'
        // Optional: Send notifications (e.g., email, Slack)
    }
  }
}
```

Explanation of Key Steps:

- Checkout: Pulls the source code from GitHub.
- Test: Installs Python dependencies and runs pytest.
- Build: Uses Docker to build an image from the provided Dockerfile.
- Push: Authenticates to Docker Hub (or another registry) and pushes the image.
- Deploy: Uses kubectl to deploy the updated Kubernetes manifests. (In this example, a simple sed command updates the image reference in the manifest.)
- Post Steps: Provide basic success/failure notifications.

5. Integrating Prometheus and Grafana

While Prometheus and Grafana aren't directly invoked by the Jenkins pipeline, they are an important part of your overall DevOps setup. Typically, you will:

- Deploy Prometheus and Grafana on your Kubernetes cluster (as shown in earlier examples).
- Instrument Your Flask App:
  - For example, expose a /metrics endpoint using the Prometheus client.
- Create Grafana Dashboards:
  - Configure Grafana to visualize metrics scraped by Prometheus.
- Use Alerts:
  - Configure Prometheus alerting rules (or Grafana alerts) to notify you if the application or infrastructure deviates from expected performance.