



Grant Agreement No.: 101096342

Call: HORIZON-JU-SNS-2022

Topic: HORIZON-JU-SNS-2022-STREAM-B-01-04

Type of action: HORIZON-JU-RIA



Holistic, omnipresent, resilient services
for future 6G wireless and computing ecosystems

CI/CD Guidelines

Revision: v.1.0

Work package	WP 5
Task	T5.2 – HORSE platform integration
Version	1
Authors	UMU

DOCUMENT REVISION HISTORY

Version	Date	Description of change	List of contributor(s)
V0.1	20/09/2022	1st version of the template for comments	Miguel Alarcón (Martel)
V0.2	30/11/2023	1st version of the guidelines	Juan Tamboleo (UMU) Alejandro Molina (UMU) Antonio Skarmeta (UMU)

Table of contents

DOCUMENT REVISION HISTORY	1
Table of contents.....	2
1 Introduction.....	5
2 Overview of GitHub CI/CD.....	6
3 Setting Up GitHub Repositories	7
3.1 Public Repository Setup	7
3.1.1 Create a New Repository	7
3.1.2 Initial Configuration:	7
3.1.3 Clone the Repository:.....	7
3.2 Private Repository Setup:	7
3.2.1 Create a New Repository:.....	7
3.2.2 Initial Configuration and Cloning:	7
3.2.3 Manage Access:.....	7
4 Configuring Local Workspace	8
4.1 Cloning the Repository	8
4.1.1 Copy the Repository URL	8
4.1.2 Clone with Terminal	8
4.1.3 Navigate to the Project Directory	8
4.2 Setting Up Initial Files	8
4.2.1 Create a .gitignore File (if not already present)	8
4.2.2 Set Up the README.md.....	8
4.2.3 Add a License (if applicable)	8
4.3 Setting Up the CI/CD Pipeline with GitHub Actions:	9
4.3.1 Create Workflow Files	9
4.3.2 Define Basic CI/CD Configurations.....	9
4.3.3 Commit and Push Your Changes	9
5 Understanding and Creating GitHub Workflow Files	11
5.1 Structure of GitHub Workflow Files.....	11
5.2 Key Directives	11
5.3 Creating GitHub Workflow Files	11
5.3.1 Start With Basic Configuration	11
5.3.2 Add Jobs and Steps	11
5.3.3 Specify Execution Conditions	11
5.3.4 Define Environment Variables and Caching	12
5.3.5 Customize for Your Project	12
5.3.6 Validate Your Configuration	12
6 Node.js Example CI/CD Setup.....	13
6.1 Initial Setup	13

6.2	Create GitHub Workflow File	13
6.3	Set up Event Triggers	13
6.4	Job: Build	13
6.5	Job: Run Tests	14
6.6	Job: Deploy Application.....	14
6.7	Push Workflow and Package.json to GitHub	14
6.8	Verify Pipeline Execution	14
7	Python Example CI/CD Setup	15
7.1	Prerequisites	15
7.2	Create GitHub Workflow File	15
7.3	Define Workflow	15
7.4	Job: Install Dependencies	16
7.5	Job: Run Tests	16
7.6	Job: Deploy Application.....	16
7.7	Push Workflow to GitHub	17
7.8	Monitoring the Pipeline.....	17
8	Java Example CI/CD Setup	18
8.1	Prerequisites	18
8.1.1	Create the pom.xml file.	18
8.1.2	Create the basic project structure:	19
8.1.3	Create a directory for your Java Application:	19
8.1.4	Write your HelloWorld.java Java Application file in the new directory:	19
8.1.5	Create a directory for your Test Cases:	19
8.1.6	Write your HelloWorldTest.java Test Code file in the new directory. For a basic test,	19
8.2	Create GitHub Workflow File	20
8.3	Define Workflow	20
8.4	Job: Build the Project	20
8.5	Job: Run Tests	20
8.6	Job: Deploy Application.....	21
8.7	Push Workflow to GitHub	21
8.8	Monitor the Pipeline	21
9	CI/CD Pipeline Fundamentals	23
9.1	Workflows	23
9.1.1	Multi-Job Workflows	23
9.1.2	Workflow Triggers	23
9.2	Jobs.....	23
9.2.1	Job Artifacts	23
9.2.2	Job Dependencies	23
9.3	Runners.....	23
9.3.1	Shared vs. Self-Hosted Runners	23

9.3.2	Runner Environments	24
9.4	Artifacts and Caching	24
9.4.1	Artifacts	24
9.4.2	Caching	24
9.5	Actions.....	24
9.5.1	GitHub Actions	24
9.5.2	Reusable Components.....	24
9.6	Workflow Files	24
9.6.1	YAML Syntax	24
9.6.2	Custom Scripts and Commands	24
9.7	Monitoring and Feedback.....	24
9.7.1	Workflow Visualizations	25
9.7.2	Logs and Artifacts	25
10	Using and Managing Self-Hosted Runners.....	26
10.1	Understanding Self-Hosted Runners Advantages	26
10.1.1	Control Over the CI Environment.....	26
10.1.2	Enhanced Security	26
10.1.3	Performance Optimization.....	26
10.2	Setting Up Self-Hosted Runners	26
10.2.1	Install the GitHub Actions Runner	26
10.2.2	Download and Register the Runner.....	26
10.2.3	Configure the Runner as a Service.....	26
10.2.4	Test Your Self-Hosted Runner (Optional)	26
10.3	Managing Self-Hosted Runners.....	27
10.3.1	Check Runner Status	27
10.3.2	Assign Runner to Specific Jobs	27
10.3.3	Monitor Runner Performance.....	27
10.3.4	Update Runner Version.....	27
10.3.5	Runner Security	28

1 Introduction

Continuous Integration and Continuous Deployment (CI/CD) are practices that automate the process of integrating code changes and deploying them to a production environment. This ensures that new code is tested and deployed quickly and reliably, which is crucial for maintaining a high pace of development and ensuring that your application remains stable and secure.

2 Overview of GitHub CI/CD

One of the most essential components of the GitHub's environment is GitHub CI/CD, which provides automation in the software development process. GitHub CI/CD starts with the initial commit and accompanies you through all the software development lifecycle. Below the key components of GitHub's CI/CD process are outlined:

- **GitHub Actions:** This is GitHub's native CI/CD tool. It automates workflows based on a variety of events within GitHub, like a push. Actions are custom automations that you create by writing YAML files and storing them in your repository.
- **Workflow Files:** GitHub uses YAML files to define workflow configurations. These files, typically named `.github/workflows/.yml`, outline the steps and jobs that should be executed automatically upon the specified triggers.
- **Workflows:** These are automated processes that you set up in your GitHub repository. A workflow can be triggered by GitHub events like a push to the repository. Workflows are composed of one or more jobs.
- **Jobs:** Within a workflow, jobs are defined to carry out specific tasks, such as building or testing code. Each job can run in an isolated environment or on a specific runner.
- **Runners:** GitHub provides hosted runners for running your workflows, or you can selfhost your runners for more control and customization. Runners execute the jobs defined in your workflows.
- **Artifacts:** GitHub allows workflows to generate artifacts, which are files produced during jobs such as binaries or logs. These can be downloaded or used in subsequent jobs.
- **Environments:** GitHub supports deployment to different environments, allowing for distinct configurations and protection rules for each.

3 Setting Up GitHub Repositories

3.1 Public Repository Setup

3.1.1 Create a New Repository

- Sign in to GitHub.
- Click on the "New" button, which you will find in the upper left corner in your dashboard.
- Enter your repository name and description (optional).
- Choose the owner.
- Set the repository visibility to Public.
- Optionally, initialize the repository with a README, **.gitignore**, and license.
- Click "Create repository".

3.1.2 Initial Configuration:

- After creation, you'll be directed to the repository's homepage.
- Here you can customize your README and add additional files or documentation.

3.1.3 Clone the Repository:

- Go to "Code" and copy the provided URL.
- On your local machine, open a terminal.
- Use **git clone <repository_url>** to clone the repository.
- Navigate into your project folder using **cd <repository_name>**.

3.2 Private Repository Setup:

3.2.1 Create a New Repository:

- Follow the same steps as above.
- When setting visibility, choose Private to ensure that only users you authorize can see this repository.

3.2.2 Initial Configuration and Cloning:

- The process for adding files and cloning the repository is the same as for public repositories.

3.2.3 Manage Access:

- In a private repository, you need to manage user access
- Go to "Settings" > "Collaborators".
- Click on "Add people" to add users and define their roles

4 Configuring Local Workspace

4.1 Cloning the Repository

4.1.1 Copy the Repository URL

- Navigate to your GitHub repository's main page.
- Go to "Code" and choose to clone with HTTPS or SSH, based on your setup.

4.1.2 Clone with Terminal

- Open a terminal on your computer.
- Enter the clone command with the URL you copied: **git clone <repository_url>**.
- Replace **<repository_url>** with the actual URL of your GitHub repository.

4.1.3 Navigate to the Project Directory

- After cloning, a new folder with the repository name will be created on your local machine.
- Navigate to this directory using **cd <repository_name>**.

4.2 Setting Up Initial Files

4.2.1 Create a .gitignore File (if not already present)

- This file specifies files that Git should intentionally ignore.
- Examples include dependencies, build folders, or environment files with sensitive information.

4.2.2 Set Up the README.md

- A README file is important for explaining your project, its setup, and contribution guidelines.

4.2.3 Add a License (if applicable)

- If your project is open source or requires a license, add it now. GitHub offers templates for common licenses.

4.3 Setting Up the CI/CD Pipeline with GitHub Actions:

4.3.1 Create Workflow Files

- Define your CI/CD configurations in workflow files within the **.github/workflows** directory in your repository. For that you might need to create those directories using **mkdir -p .github/workflows**.

4.3.2 Define Basic CI/CD Configurations

- Start by setting up jobs in your workflow, such as build, test, and deploy.
- Specify the steps within these jobs, outlining the commands or actions to be executed.
- Here's an example workflow file (main.yml):

```
# Workflow name
name: CI
# Trigger the workflow on every 'push' event
on: [push]
# Define jobs in the workflow
jobs:
  build: # Specifies the runner environment (latest Ubuntu version)
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3 # This action checks out your repository's source code to the GitHub Actions runner. Placing it in a directory called $GITHUB_WORKSPACE. This step allows all the other steps in the job to have access to your codebase.
      - name: Build
        run: echo "Building the project" # Execute the actual commands
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Test
        run: echo "Running tests"
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Deploy
        run: echo "Deploying the project"
```

4.3.3 Commit and Push Your Changes

Stage your changes for commit using **git add ..**

Commit the changes with **git commit -m "Initial commit with CI/CD configurations"**.

Push the commit to your GitHub repository with **git push -u origin main**.

5 Understanding and Creating GitHub Workflow Files

GitHub workflow files are YAML files used to configure specific instructions for GitHub Actions, serving as the blueprint for your CI/CD pipeline. They define what actions to take, when to perform them, and how. Below, we'll explore the key components and syntax to help you understand how to create and customize these files.

5.1 Structure of GitHub Workflow Files

A typical GitHub workflow file contains the following key sections:

- **on:** Specifies the events that trigger the workflow.
- **jobs:** The core section where you define the tasks to be executed in the workflow.
- **steps:** Under each job, steps define individual tasks that run commands or actions.
- **runs-on:** Specifies the type of machine to run the job on, such as Ubuntu or Windows.
- **env:** Here, you can define environment variables accessible in the workflow.

5.2 Key Directives

- **uses:** Specifies an action to use as part of a step in a job.
- **run:** The main set of commands or shell script that a step will execute.
- **if:** Defines conditions under which a step or job should be executed.
- **name:** Gives a name to a step or a job for clarity.

5.3 Creating GitHub Workflow Files

5.3.1 Start With Basic Configuration

Begin by defining the events that trigger your workflow. For instance:

on: [push, pull_request]

5.3.2 Add Jobs and Steps

Define jobs and include steps within them. For example:

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: echo "Building the project..."
      - run: build_command
```

5.3.3 Specify Execution Conditions

Use conditions to control when jobs or steps run. For example:

```
deploy:
```

```
runs-on: ubuntu-latest
if: github.ref == 'refs/heads/main' # This job will only
run if triggered by an event to the main branch
steps:
  - uses: actions/checkout@v3
  - run: deploy_command
```

5.3.4 Define Environment Variables and Caching

Set environment variables and cache dependencies as needed:

```
env:
  MY_VAR: value
steps:
  - uses: actions/cache@v3 # To cache dependencies and
reused files between jobs or workflow runs.
  with:
    path: path_to_dependencies # Path where your project's
dependencies are stored.
    key: cache-key # The 'key' is an identifier used to save
and retrieve the cache.
```

5.3.5 Customize for Your Project

Tailor the workflow file to your project's needs, including setup commands and specifying the exact environment or Docker image you need.

5.3.6 Validate Your Configuration

GitHub Actions automatically validates your workflow file when you commit it to your repository. Any syntax errors or issues will be flagged in the Actions tab of your GitHub repository.

Refer to the GitHub Actions documentation for a comprehensive list of all available options and directives.

6 Node.js Example CI/CD Setup

Here's a step-by-step guide on setting up a basic CI/CD pipeline for a Node.js application using GitHub Actions:

6.1 Initial Setup

Ensure you have a Node.js application with a **package.json** file that defines the project's dependencies and scripts.

Use **npm init -y** to initialize your project and modify the test script in **package.json** to asimple echo command as a placeholder. For example, using the following command:

```
sed -i 's/"test": "echo \\"Error: no test specified\\" "&& exit 1"/"test": "echo \\"Testing...\\" " ; exit 0"/' package.json
```

6.2 Create GitHub Workflow File

Create a workflow file in **.github/workflows**. You might need to create those directories using **mkdir -p .github/workflows**. For example, name it **nodejs.yml**.

6.3 Set up Event Triggers

Specify the events that will trigger your jobs.

```
name: Node.js CI
on: [push]
jobs:
```

6.4 Job: Build

Create a job to install your project's dependencies.

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

    - name: Set up Node.js
      uses: actions/setup-node@v3 # This ensures that the specified
      # Node.js version is installed and available for subsequent steps
      with:
        node-version: '18.0' # You could replace it with a
        # different version of Node.js
    - name: Install dependencies
      run: npm install
```

6.5 Job: Run Tests

Define a job for running tests. This assumes you have a script named `test` in your `package.json`.

```
test:
  needs: build # Ensures that this job runs only after the 'build'
               job has completed successfully
  runs-on: ubuntu-latest
  steps:
    - name: Checkout code
      uses: actions/checkout@v3

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

6.6 Job: Deploy Application

For deployment, you can use a simple `echo` command as a placeholder. Replace it with your actual deployment script.

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

    - name: Deploy
      run: echo "Deploying application..."
```

6.7 Push Workflow and Package.json to GitHub

With your `nodejs.yml` and `package.json` configured, commit them to your GitHub repository.

```
git add .github/workflows/nodejs.yml package.json
git commit -m "Add GitHub Actions CI/CD for Node.js"
git push origin main
```

6.8 Verify Pipeline Execution

Once pushed, GitHub Actions will detect the workflow file and execute the pipeline. You can check the pipeline status in the Actions tab of your GitHub repository.

7 Python Example CI/CD Setup

Here's how you can configure a basic CI/CD pipeline for testing a Python application with GitHub Actions:

7.1 Prerequisites

Ensure your Python project has a **requirements.txt** file listing all dependencies. Your file could look as follows:

pytest

Structure your project with a **src** and **tests** directory. You can execute:

```
mkdir -p src
mkdir -p tests
```

Write tests in the **tests/** directory, for example, using pytest. Inside your **/tests** directory create a **test_math.py** file to illustrate the example with two math tests:

```
# tests/test_guideline.py
import sys
from os.path import abspath, dirname
sys.path.insert(0, dirname(dirname(abspath(__file__))))

from src.guideline import add
def test_addition():
    assert add(1, 2) == 3
    assert add(-1, 1) == 0
    assert add(10, -5) == 5
```

7.2 Create GitHub Workflow File

In the root of your Python project directory, create a workflow file in **.github/workflows**, such as **python-app.yml**. You might need to create those directories using:

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

7.3 Define Workflow

Set up your workflow to run on specific triggers and define the jobs that suit your needs.

```
# Workflow name
name: Python CI
# Trigger the workflow on push events
on:
  push:
# Define the jobs
jobs:
```

7.4 Job: Install Dependencies

Configure a job to set up a Python environment and install dependencies.

```
build:
  runs-on: ubuntu-latest
  # Specify 'python-version' as an output variable to be used by
  # subsequent jobs in this workflow.
  outputs:
    python-version: 3.9
  steps:
    - uses: actions/checkout@v3 # Check out the repository's code
      # Set up Python 3.9 environment
    - name: Set up Python 3.9
      uses: actions/setup-python@v3
      with:
        python-version: 3.9 # Specify Python version
```

7.5 Job: Run Tests

Set up a job to run your tests. We can use pytest for this purpose.

```
# Job for running tests
test:
  needs: build # This job needs to wait for build to complete
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v3
      # Set up Python (version from build job)
    - name: Set up Python
      uses: actions/setup-python@v3
      with:
        python-version: ${needs.setup.outputs.pythonversion} #
        Python version based on the output of the build job. This ensures
        consistency across different jobs.
      # Install dependencies and run tests
    - name: Install dependencies and run tests
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
        pip install pytest # Install pytest for running tests
        pytest # Execute tests
```

7.6 Job: Deploy Application

Use a placeholder for the deploy stage and replace it with your actual deployment commands when ready.

```
# Job for deployment
deploy:
  needs: test # This job needs to wait for test to complete
```



```
runs-on: ubuntu-latest
if: github.ref == 'refs/heads/main' # Only run on the main branch
steps:
  - name: Deploy
    run: echo "Deploying application..."
    # Add your deployment commands here
```

7.7 Push Workflow to GitHub

Commit and push your workflow file and other project files to your GitHub repository.

```
git add .
git commit -m "Add Python project structure and GitHub Actions CI/CD configuration"
git push origin main
```

7.8 Monitoring the Pipeline

Once you push the workflow file, GitHub Actions will automatically start the pipeline. Monitor its progress in the Actions tab of your repository and troubleshoot any failures.

8 Java Example CI/CD Setup

Configuring a CI/CD pipeline for a Java project in GitHub typically involves compiling the code, running tests, and potentially deploying artifacts using Maven. Here's a basic guide:

8.1 Prerequisites

Ensure your Java project has a pom.xml file in the root directory that defines the project's build script.

Create the basic project structure and files, including source and test directories.

8.1.1 Create the pom.xml file.

Below is an example of a basic, commented pom.xml file:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <!-- Model version of Maven POM file; always 4.0.0 for Maven 2 &
3-->
    <modelVersion>4.0.0</modelVersion>
    <!-- Basic project information -->
    <groupId>com.example</groupId>
    <artifactId>java-github-example</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!-- Java version compatibility -->
    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>
    <!-- Project dependencies -->
    <dependencies>
        <!-- Example dependency: JUnit, a popular testing framework
for Java -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
        <!-- Add more dependencies as needed -->
    </dependencies>
    <!-- Build plugins -->
    <build>
        <plugins>
            <!-- Maven Compiler Plugin: configures how the project
source code should be compiled -->
            <plugin>
```

```

        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
            <!-- Use the properties defined above for the
source and target Java versions -->
            <source>${maven.compiler.source}</source>
            <target>${maven.compiler.target}</target>
        </configuration>
    </plugin>
    <!-- Other plugins can be added here -->
</plugins>
</build>
</project>

```

8.1.2 Create the basic project structure:

```

mkdir -p src/main/java
mkdir -p src/test/java

```

8.1.3 Create a directory for your Java Application:

```

mkdir -p src/main/java/com/example

```

8.1.4 Write your HelloWorld.java Java Application file in the new directory:

```

package com.example;
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}

```

8.1.5 Create a directory for your Test Cases:

```

mkdir -p src/test/java/com/example

```

8.1.6 Write your HelloWorldTest.java Test Code file in the new directory. For a basic test,

you might use JUnit (it must be included in your pom.xml dependencies):

```

package com.example;
import org.junit.Test;
import static org.junit.Assert.*;
public class HelloWorldTest {
    @Test
    public void testMain() {
        // Your test code here
    }
}

```

```
    assertTrue(true); // Example test
  }
}
```

8.2 Create GitHub Workflow File

In the root directory of your Java project, create a workflow file in **.github/workflows**, such as **java-app.yml**. You might need to create those directories using **mkdir -p .github/workflows**.

8.3 Define Workflow

Set up your workflow to run on specific triggers and define the jobs that suit your needs.

```
# Workflow's name, it will appear in the GitHub Actions tab
name: Java CI
# This defines the events that trigger the workflow
on:
  push: # Triggers on push events
# Jobs that the workflow will execute
jobs:
```

8.4 Job: Build the Project

Configure a job to build your project using Maven.

```
# Job for building the application
build:
  runs-on: ubuntu-latest # Specifies the runner
  environment
  steps: # Steps to execute in the build job
    - uses: actions/checkout@v3 # Checks out the repository code
    - name: Set up JDK 11
      uses: actions/setup-java@v3 # Sets up Java Development Kit version
11
    with:
      java-version: '11'
      distribution: 'adopt'
    - name: Build with Maven
      run: mvn compile # Compiles the code using Maven
```

8.5 Job: Run Tests

Set up a job to execute your tests.

```
# Job for testing the application
test:
```

```

needs: build # This job depends on the successful completion of the
build job
runs-on: ubuntu-latest
steps:
  - uses: actions/checkout@v3
  - name: Set up JDK 11
    uses: actions/setup-java@v3
    with:
      java-version: '11'
      distribution: 'adopt'
  - name: Test with Maven
    run: mvn test # Runs tests using Maven

```

8.6 Job: Deploy Application

Use a placeholder for deployment and replace it with your actual deployment commands.

```

# Job for deploying the application
deploy:
  needs: test # This job depends on the successful completion of the
test job
  runs-on: ubuntu-latest
  if: github.ref == 'refs/heads/main' # Runs only if the push is to
the 'main' branch
  steps:
    - uses: actions/checkout@v3
    - name: Set up JDK 11
      uses: actions/setup-java@v3
      with:
        java-version: '11'
        distribution: 'adopt'
    - name: Deploy
      run: echo "Deploying application..." # Placeholder for
deployment commands
# Add your actual deployment commands here

```

8.7 Push Workflow to GitHub

Commit the workflow file and push it to your GitHub repository.

```

git add .
git commit -m "Initial Java project setup and GitHub Actions CI/CD
configuration"
git push origin main

```

8.8 Monitor the Pipeline

Check the pipeline's progress and logs in the Actions tab of your GitHub repository. Troubleshoot any issues that arise.

9 CI/CD Pipeline Fundamentals

Understanding key concepts and components is essential for effectively utilizing GitHub Actions for CI/CD.

9.1 Workflows

Workflows are automated procedures that run one or more jobs. Defined in YAML files within the **.github/workflows** directory, they can be triggered by various GitHub events like push, pull requests, manual triggers, or schedules.

9.1.1 Multi-Job Workflows

Workflows can consist of multiple jobs, which can run in parallel or sequentially, depending on their dependencies.

9.1.2 Workflow Triggers

You can configure workflows to trigger on different GitHub events, such as on code commits (push, pull_request), manual triggers (workflow_dispatch), schedules (schedule), or even external events (repository_dispatch).

9.2 Jobs

Jobs are a set of steps that execute on the same runner. Each job can run a series of commands or actions.

9.2.1 Job Artifacts

After a job completes, it can produce artifacts that can be used by subsequent jobs or stored for future use.

9.2.2 Job Dependencies

Jobs can depend on the completion of other jobs, allowing you to orchestrate complex workflows.

9.3 Runners

Runners are servers that execute your jobs. GitHub provides hosted runners for Linux, Windows, and macOS, or you can self-host runners for more control and customization.

9.3.1 Shared vs. Self-Hosted Runners

Shared runners are managed by GitHub and available to all repositories. Self-hosted runners are managed by you and can be specific to a repository or organization.

9.3.2 Runner Environments

Runners can use various environments, such as Docker containers or virtual machines, to provide the necessary context for your jobs.

9.4 Artifacts and Caching

9.4.1 Artifacts

Files generated by jobs, such as binaries or reports, can be uploaded as artifacts and shared between jobs or downloaded after a workflow completes.

9.4.2 Caching

GitHub Actions allows caching dependencies or other directories to speed up workflow execution.

9.5 Actions

9.5.1 GitHub Actions

GitHub Actions are individual tasks that you can combine to create jobs and customize your CI/CD workflows. They are the building blocks of a workflow.

9.5.2 Reusable Components

Actions can be reused across different workflows within the same repository or across different repositories, promoting efficiency and consistency.

9.6 Workflow Files

The workflow file (e.g., `.github/workflows/main.yml`) is where you define your CI/CD pipeline. It includes the triggers, jobs, steps, and more.

9.6.1 YAML Syntax

Workflow files use YAML syntax. Correct syntax is crucial for the workflow to run correctly.

9.6.2 Custom Scripts and Commands

Within each job's steps, you can execute custom scripts and commands.

9.7 Monitoring and Feedback

GitHub provides tools to monitor workflows and give feedback on the performance and status of jobs.

9.7.1 Workflow Visualizations

The Actions tab in a GitHub repository provides a visual representation of workflows, showing the status of each job.

9.7.2 Logs and Artifacts

Detailed logs are available for each job, and artifacts can be downloaded post-workflow completion.

10 Using and Managing Self-Hosted Runners

In GitHub Actions, self-hosted runners are machines set up by you to run CI/CD jobs, offering control over the CI environment, enhanced security, and performance optimization.

10.1 Understanding Self-Hosted Runners Advantages

10.1.1 Control Over the CI Environment

Customize the environment with the necessary software and dependencies.

10.1.2 Enhanced Security

A dedicated runner for your project reduces the risk of sensitive information leakage.

10.1.3 Performance Optimization

Optimize the hardware and software configuration to meet your project's needs.

10.2 Setting Up Self-Hosted Runners

10.2.1 Install the GitHub Actions Runner

Download and install the GitHub Actions runner on your machine.

For general detailed instructions, refer to the [GitHub Actions Runner Documentation](#).

10.2.2 Download and Register the Runner

Obtain a registration token from your repository's or organization's settings in GitHub.

Register the runner using the token and select the appropriate configuration.

Detailed guidance is provided in the [Runner Registration Documentation](#).

10.2.3 Configure the Runner as a Service

After installation and registration, set up the GitHub Actions runner as a service on your server as explained step by step in the documentation.

10.2.4 Test Your Self-Hosted Runner (Optional)

Create a workflow file in your repository with a job that specifies runs-on: self-hosted to use the self-hosted runner. For instance:

```
name: CI Workflow run on Self-hosted Runner
on: [push, pull_request]
jobs:
  build:
    runs-on: self-hosted # Using your self-hosted runner
    steps:
      - name: Build Job
        run: echo "Placeholder for Build Job"
  test:
    runs-on: self-hosted
    needs: build
    steps:
      - name: Test Job
        run: echo "Placeholder for Test Job"
  deploy:
    runs-on: self-hosted
    needs: test
    steps:
      - name: Deploy Job
        run: echo "Placeholder for Deploy Job"
```

Commit and push these changes, then check the Actions tab in your repository to see the job executed by your self-hosted runner.

10.3 Managing Self-Hosted Runners

10.3.1 Check Runner Status

In your repository or organization settings, you can view the status of your self-hosted runners under "Actions" → "Runners" → "Self-hosted runners". Ensure they are active and running correctly.

10.3.2 Assign Runner to Specific Jobs

Use labels when registering your self-hosted runner and specify these labels in your workflow files to control which jobs the runner will execute.

10.3.3 Monitor Runner Performance

Regularly check your self-hosted runner's performance and adjust hardware resources if necessary to avoid slow build times or timeouts.

10.3.4 Update Runner Version

Keep your GitHub Actions runner updated to the latest version to ensure compatibility with new features and security patches.

10.3.5 Runner Security

Regularly update the server where your runner is hosted. Follow the principle of least privilege when configuring access to resources.