GITHUB ACTIONS - COMPLETE GUIDE

WHAT IS GITHUB ACTIONS?

GitHub Actions is yet another CI/CD solution.

PRIMARY DIFFERENCE: GITHUB ACTIONS vs JENKINS

The primary difference between GitHub Actions and Jenkins:

GitHub Actions is focused only on GitHub.

Comparison: Just like GitLab CI which offers CI/CD solution for GitLab VCS.

WHEN TO USE GITHUB ACTIONS

If your organization is using GitHub for maintaining code: GitHub Actions is a good solution.

Problem for future: If you plan to migrate to some other VCS, then GitHub Actions should not be the solution used for the purpose of CI/CD.

Why: Because they are platform-oriented solutions.

COMPARISON WITH TERRAFORM vs CLOUDFORMATION

Similar to why we chose Terraform over CloudFormation Template:

Terraform: A tool that can work on multi-cloud or hybrid cloud environment.

CloudFormation: Works only with AWS.

Similarly:

GitHub Actions: Works only with GitHub.

Jenkins: Works with any VCS (GitHub, GitLab, Bitbucket, etc.).

NO PLUGIN INSTALLATION NEEDED

Key difference from Jenkins:

In GitHub Actions: We don't have to install plugins.

Jenkins: Requires manual plugin installation and configuration.

GitHub Actions: Plugins are already available, just reference them in your workflow file.

CREATING GITHUB ACTIONS WORKFLOW

STEP 1: CREATE WORKFLOW FOLDER STRUCTURE

Location: Go to the root of the repository on GitHub.

Create folder structure:

.github/workflows/

Inside this folder: We can have our YAML files.

These are called action files.

STEP 2: CREATE ACTION FILE

Example file: first-actions.yml

Location:

repository-root/

└── .github/

  └── workflows/

    └── first-actions.yml

Content example:

yaml

```
on: [push]


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

```
  steps:
    - uses: actions/checkout@v3
    - name: Run tests
      run: echo "Tests running"
```

TRIGGER CONFIGURATION

Line: on: [push]

Meaning: Whenever a push is done in this repository, this pipeline will get executed.

In this file we also define the jobs: Jobs that are to be done as part of pipeline.

Flow: Whenever there is a push in this repo, this file or the pipeline defined in this file gets executed.

MULTIPLE ACTION FILES

Flexibility: There is no limitation to the number of files present in this folder.

We may have as many action files as we wish.

EXAMPLE USE CASES FOR MULTIPLE FILES

File 1: pr-validation.yml Purpose: Verify that user has provided all the details in the pull request.

Example: Organization might want proper description of what issue a user is trying to solve when he makes a pull request. We can validate all these things in one action file or pipeline.

File 2: code-format.yml Purpose: Verify formatting-related issues.

File 3: ci-test.yml Purpose: Verify if your CI is passing.

File 4: cd-deploy.yml Purpose: Do your CD (deployment).

Conclusion: It depends on how you want to organize things.

Basically, we can have one or more of these action files in this folder for executing the GitHub Actions.

MULTIPLE TRIGGERS IN ONE FILE

We can set up one or more actions in a file.

Examples:

Trigger on push only:

yaml

on: [push]

Trigger on push and pull request:

yaml

on: [push, pull_request]

Trigger on specific events:

yaml

on:

  push:

    branches: [main, develop]

  pull_request:

    branches: [main]

```

Flow:

Whenever any of the actions matches with one of these triggers, the pipeline gets executed.

---

PRACTICAL EXAMPLE: PYTHON APPLICATION

---
```

STEP 1: CREATE PYTHON APPLICATION

Create folder in repository:

```

src/

Create file: src/addition.py

File content:

python

*# addition.py*

def add(a, b):

  return a + b

def test_add():

  assert add(1, 2) == 3

  assert add(1, -1) == 0
```

Code explanation:

Function 1: add(a, b) Very normal addition function that adds two numbers and returns the result.

Function 2: test_add() Function to test the functionality of addition function.

This is unit testing.

What it does:

- Tests if add(1, 2) returns 3
- Tests if add(1, -1) returns 0
- Uses assert to validate correctness

STEP 2: CREATE GITHUB ACTIONS WORKFLOW

Task: Write a pipeline that does the following:

- Whenever there is a new commit getting created
- GitHub Actions should checkout this code
- Create a Python environment
- Test the code

STEP 3: WRITE WORKFLOW FILE

Create file: .github/workflows/first-actions.yml

File content:

yaml

```
name: My First GitHub Actions

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        python-version: [3.8, 3.9]

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: ${{ matrix.python-version }}
```

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

- name: Run tests
  run: |
    cd src
    python -m pytest addition.py
```

WORKFLOW FILE EXPLANATION

Line 1: name: My First GitHub Actions Name of the workflow (displays in GitHub UI).

Line 3: on: [push] Trigger: Run whenever code is pushed to any branch.

Lines 5-6: jobs: build: Define a job named "build".

Correlation with Jenkins: Jobs we can correlate with Jenkins pipeline.

If we create 3 jobs: It means we are executing 3 pipelines.

In one file: We can have as many jobs as we want, which means as many pipelines as we want.

In our example: We have just one job or one pipeline.

Line 7: runs-on: ubuntu-latest We are using ubuntu-latest as the container image.

Inside that: We are setting up a Python environment.

Lines 9-11: strategy: matrix: python-version: [3.8, 3.9]

Matrix strategy explanation: Here we are mentioning that my addition functionality has to be verified on Python 3.8 and Python 3.9.

Result: That's why we will have 2 jobs which will get executed when we run the GitHub Actions.

This is the flexibility GitHub is offering us.

How it works:

- Job 1: Runs with Python 3.8
- Job 2: Runs with Python 3.9
- Both run in parallel
- Both test same code

Lines 13-27: steps: Where we actually write the code.

This is similar to stages in Jenkins.

Flexibility: We can have as many steps as we want.

Step 1: Checkout code

yaml

- uses: actions/checkout@v3

Purpose: Checkout the code from repository.

What is actions/checkout@v3: This is one of the standard actions.

Where does this come from: If we don't know where this came from, we can simply go to browser and check GitHub Actions documentation.

What it is: This is an action (similar to plugin) that we are using.

Difference from Jenkins: In Jenkins, we used to install the plugins. But in GitHub Actions, plugins are already installed by default. All we have to do is whichever plugin we want to use, we have to write it here.

Step 2: Set up Python

yaml

```
- name: Set up Python
  uses: actions/setup-python@v2
  with:
    python-version: ${{ matrix.python-version }}
```

Purpose: Install Python in the runner.

Action: actions/setup-python@v2

Important clarification: When we say actions/setup-python@v2, it doesn't mean version 2 of Python. It means version 2 of this action/plugin.

What it does: Installs Python version specified in matrix (3.8 or 3.9).

${{ matrix.python-version }}: Variable that takes value from matrix (3.8 or 3.9).

Step 3: Install dependencies

yaml

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

Purpose: Install required Python packages.

Commands:

1. python -m pip install --upgrade pip: Upgrade pip to latest version
2. pip install pytest: Install pytest testing framework

Pipe symbol (|): Allows writing multiple commands in YAML.

Step 4: Run tests

yaml

```
- name: Run tests
  run: |
      cd src
```

python -m pytest addition.py

Purpose: Execute the unit tests.

Commands:

1. cd src: Navigate to src folder
2. python -m pytest addition.py: Run pytest on addition.py file

What happens:

- pytest finds test_add() function
- Runs both assertions
- Reports pass/fail

LIMITATION OF GITHUB ACTIONS PLUGINS

The only limitation with this: These actions/plugins are very limited in scope because GitHub Actions is very new as compared to the other CI/CD tools that are available in the market.

Comparison:

Jenkins:

- Thousands of plugins
- Mature ecosystem
- Plugin for almost everything

GitHub Actions:

- Growing ecosystem
- Limited compared to Jenkins
- Most common needs covered

SELF-HOSTED RUNNERS

Option available: We have an option of creating our own self-hosted runner.

How to access: Go to repository Settings → Actions → Runners

What it provides:

- Run workflows on your own infrastructure

- More control
- Better for private/secure code
- Can use your own cloud resources

Setup:

1. Go to Settings → Actions → Runners
2. Click "New self-hosted runner"
3. Choose OS (Linux/Windows/macOS)
4. Follow installation instructions
5. Runner connects to GitHub

Use in workflow:

yaml

runs-on: self-hosted

STORING SECRETS IN GITHUB ACTIONS

Question: If I am doing the entire thing on GitHub Actions, where will I store the passwords or secret information?

Answer: GitHub Actions provides native support to store secrets.

How to store secrets:

Step 1: Go to repository Settings

Step 2: Click on "Secrets and variables" → "Actions"

Step 3: Click "New repository secret"

Step 4: Add secret:

- Name: DATABASE_PASSWORD
- Secret: your-secret-password
- Click "Add secret"

Use in workflow:

yaml

steps:

```
  - name: Use secret
    run: echo ${{ secrets.DATABASE_PASSWORD }}
```

Benefit:

Secrets are encrypted and never exposed in logs.

We can store CI/CD secrets using GitHub's built-in secret management.

GITHUB ACTIONS vs JENKINS COMPARISON

From image provided:

ADVANTAGES OF GITHUB ACTIONS OVER JENKINS

Advantage 1: Hosting

Jenkins:

- Self-hosted
- Requires its own server to run
- You manage infrastructure
- You handle maintenance

GitHub Actions:

- Hosted by GitHub
- Runs directly in your GitHub repository
- No infrastructure to manage
- GitHub handles everything

Advantage 2: User Interface

Jenkins:
- Complex and sophisticated user interface
- Steep learning curve
- Many configuration options

GitHub Actions:
- More streamlined and user-friendly interface
- Better suited for simple to moderate automation tasks
- Integrated into GitHub UI
- Easier to learn

Advantage 3: Cost

Jenkins:
- Can be expensive to run and maintain
- Especially for organizations with large and complex automation needs
- Need to pay for:

- EC2 instances
- Storage
- Maintenance
- Scaling infrastructure

GitHub Actions:

- Free for open-source projects
- Has a tiered pricing model for private repositories
- Making it more accessible to smaller organizations and individual developers
- No infrastructure costs
- Pay only for compute minutes used

Pricing:

- Public repositories: Free unlimited
- Private repositories: 2000 free minutes/month, then $0.008/minute

---

ADVANTAGES OF JENKINS OVER GITHUB ACTIONS

Advantage 1: Integration

Jenkins:

- Can integrate with a wide range of tools and services
- Thousands of plugins available
- Can integrate with any tool (cloud, on-premise, custom)

GitHub Actions:

- Tightly integrated with the GitHub platform
- Making it easier to automate tasks related to your GitHub workflow
- Limited to GitHub ecosystem primarily

CONCLUSION FROM IMAGE

Jenkins is better suited for:
- Complex automation tasks
- Large-scale automation needs
- Multi-platform integration
- Organizations using multiple VCS

GitHub Actions is:
- More cost-effective solution
- User-friendly
- Good for simple to moderate automation needs
- Best when using GitHub exclusively

DISADVANTAGES COMPARISON

GitHub Actions disadvantage:

Very scoped to the platform.

If we want to migrate to other VCS:

This is not a good option.

Need to:

- Rewrite all workflows
- Learn new CI/CD tool for new platform
- Migration effort required

Jenkins advantage:

- Works with any VCS
- Portable across platforms
- No vendor lock-in

PRACTICAL DEMONSTRATION

ARGO CD REPOSITORY EXAMPLE

From image 2:

This shows the Argo CD project's GitHub Actions workflows folder.

Location: argoproj/argo-cd/.github/workflows/

Multiple workflow files visible:

1. ci-build.yaml

2. codeql.yml

3. image.yaml

4. pr-title-check.yml

5. release.yaml

6. update-snyk.yaml

This demonstrates:

Large projects use multiple workflow files for different purposes.

Each file has specific purpose:

- ci-build.yaml: Build and test code

- codeql.yml: Security code scanning

- image.yaml: Build Docker images

- pr-title-check.yml: Validate pull request titles

- release.yaml: Handle releases

- update-snyk.yaml: Security dependency updates

Recent activity shown:

Updates to various workflows with commit messages and PR numbers.

This is a real-world example of GitHub Actions in a major open-source project.

RUNNING YOUR GITHUB ACTIONS

STEP 1: COMMIT WORKFLOW FILE

After creating .github/workflows/first-actions.yml:

Commands:

```
git add .

git commit -m "Add GitHub Actions workflow"

git push origin main
```

STEP 2: AUTOMATIC EXECUTION

What happens:

Every time you make changes in your addition.py and push that code to Git, your GitHub Actions run automatically.

GitHub automatically:

1. Detects push event
2. Finds workflow files in .github/workflows/
3. Checks triggers (on: [push])
4. Trigger matches
5. Starts workflow execution

STEP 3: CHECK WORKFLOW STATUS

Visual indicator:

We can see the orange dot in the root directory.

Where to see:

- Go to your GitHub repository
- Look at commit list
- You'll see colored dots next to commits:
  - Orange/Yellow dot: Workflow running
  - Green checkmark: Workflow passed
  - Red X: Workflow failed

Click on the dot:

Shows details of the GitHub Actions execution.

STEP 4: VIEW WORKFLOW DETAILS

Click on "Actions" tab in GitHub repository.

You will see:

- All workflow runs
- Current status
- Past runs
- Duration

Click on specific workflow run.

You will see:

- Workflow name: My First GitHub Actions

- Triggered by: push

- Jobs: build (3.8), build (3.9)

- Status: Success or Failure

STEP 5: VIEW JOB DETAILS

Click on one of the jobs: build (3.8)

This shows detailed execution log.

EXECUTION LOG EXPLANATION

From document provided:

Header:

```
build (3.8)

succeeded 2 minutes ago in 13s
```

Meaning:

- Job name: build with Python 3.8

- Status: succeeded

- Completed: 2 minutes ago

- Duration: 13 seconds

Step 1: Set up Python (7 seconds)

Log output:

```
Run actions/setup-python@v2

Version 3.8 was not found in the local cache

Version 3.8 is available for downloading

Download from "https://github.com/actions/python-versions/releases/download/3.8.18-12303122501/python-3.8.18-linux-24.04-x64.tar.gz"

Extract downloaded archive

/usr/bin/tar xz --warning=no-unknown-keyword -C /home/runner/work/_temp/... -f /home/runner/work/_temp/...

Execute installation script

Check if Python hostedtoolcache folder exist...

Create Python 3.8.18 folder

Copy Python binaries to hostedtoolcache folder

Create additional symlinks

Upgrading pip...

Successfully setup CPython (3.8.18)
```

What happened:

1. GitHub Actions looked for Python 3.8 in cache

2. Not found, so downloaded Python 3.8.18

3. Extracted the archive

4. Installed Python binaries

5. Created symlinks

6. Upgraded pip

7. Python 3.8.18 ready to use

Duration: 7 seconds

Step 2: Install dependencies (1 second)

Log output:

```
Run python -m pip install --upgrade pip

Requirement already satisfied: pip in /opt/hostedtoolcache/Python/3.8.18/x64/lib/python3.8/site-packages (25.0.1)

Collecting pytest
  Downloading pytest-8.3.5-py3-none-any.whl.metadata (7.6 kB)
Collecting exceptiongroup>=1.0.0rc8 (from pytest)
  Downloading exceptiongroup-1.3.1-py3-none-any.whl.metadata (6.7 kB)
Collecting iniconfig (from pytest)
  Downloading iniconfig-2.1.0-py3-none-any.whl.metadata (2.7 kB)
```

Collecting packaging (from pytest)

Downloading packaging-25.0-py3-none-any.whl.metadata (3.3 kB)

Collecting pluggy<2,>=1.5 (from pytest)

Downloading pluggy-1.5.0-py3-none-any.whl.metadata (4.8 kB)

Collecting tomli>=1 (from pytest)

Downloading tomli-2.3.0-py3-none-any.whl.metadata (10 kB)

Installing collected packages: typing-extensions, tomli, pluggy, packaging, iniconfig, exceptiongroup, pytest

Successfully installed exceptiongroup-1.3.1 iniconfig-2.1.0 packaging-25.0 pluggy-1.5.0 pytest-8.3.5 tomli-2.3.0 typing-extensions-4.13.2

```

What happened:

1. Upgraded pip (already latest)

2. Downloaded pytest and its dependencies

3. Installed all packages

4. pytest ready to use

Duration: 1 second

---

Step 3: Run tests (1 second)

Log output:

```

Run cd src

```
============================= test session starts ==============================

platform linux -- Python 3.8.18, pytest-8.3.5, pluggy-1.5.0

rootdir: /home/runner/work/Devops/Devops/src

collected 1 item

addition.py .                                                            [100%]

============================== 1 passed in 0.01s ===============================
```

What happened:

1. Changed directory to src

2. Ran pytest on addition.py

3. pytest found test_add() function

4. Ran 2 assertions

5. Both assertions passed

6. Test completed successfully

Test results:

- Collected: 1 item (test_add function)

- Status: . (dot means passed)

- Progress: [100%]

- Result: 1 passed in 0.01s

Duration: 1 second

Step 4: Post job cleanup (0 seconds)

Log output:

```
Post job cleanup.

/usr/bin/git version

git version 2.52.0

Temporarily overriding HOME='/home/runner/work/_temp/...' before making global git config changes

Adding repository directory to the temporary git global config as a safe directory

/usr/bin/git config --global --add safe.directory /home/runner/work/Devops/Devops

/usr/bin/git config --local --name-only --get-regexp core\.sshCommand

/usr/bin/git submodule foreach --recursive...

/usr/bin/git config --local --unset-all http.https://github.com/.extraheader
```

What happened:

- GitHub Actions cleaning up

- Removing credentials

- Cleaning Git configuration

- Preparing to terminate runner

Duration: 0 seconds (very fast)

TOTAL EXECUTION TIME

Job build (3.8):

- Set up Python: 7s

- Install dependencies: 1s

- Run tests: 1s

- Cleanup: 0s

- Total: 13s

Job build (3.9):

Similar timing, runs in parallel.

Overall workflow: Completed in approximately 13 seconds.

PARALLEL EXECUTION

Because of matrix strategy:

Two jobs ran simultaneously:

1. build (3.8): Python 3.8 environment

2. build (3.9): Python 3.9 environment

Both jobs:

- Created separate runners

- Ran same tests

- Completed independently

Result:

Verified code works on both Python 3.8 and 3.9.

VIEWING RESULTS ON GITHUB

After push:

Go to GitHub repository.

In commit list:

You will see commits with status indicators.

Status indicators:

Orange/Yellow dot:

Workflow is currently running.

Green checkmark:

All workflows passed successfully.

Red X:

One or more workflows failed.

Click on status indicator:

Shows quick summary of all workflows.

Click "Details":

Opens full workflow run page.

Workflow run page shows:

- All jobs

- Each job status

- Duration

- Logs

CHECKING BUILD SUCCESS

We can check if the build was successful or not by:

Method 1: Commit list

Look for green checkmark next to commit.

Method 2: Actions tab

- Click "Actions" tab

- See all workflow runs

- Latest run shows success/failure

Method 3: Pull request

If triggered by PR, status shown in PR page.

Method 4: Notifications

GitHub can send email notifications for failures.

WHAT HAPPENS ON EVERY PUSH

Process:

Step 1: Developer makes changes

Edit addition.py or any other file.

Step 2: Commit and push

```
git add .
git commit -m "Update addition function"
git push origin main
```

Step 3: GitHub detects push

GitHub receives the push event.

Step 4: Trigger matches

Workflow has on: [push], so it triggers.

Step 5: Workflow starts

GitHub Actions starts executing first-actions.yml.

Step 6: Create runners

- Creates runner for Python 3.8

- Creates runner for Python 3.9

- Both run in parallel

Step 7: Execute steps

Each runner:

1. Checks out code

2. Sets up Python (3.8 or 3.9)

3. Installs dependencies

4. Runs tests

Step 8: Report results

- Both jobs complete

- Status reported to GitHub

- Green checkmark or red X shown

Step 9: Cleanup

- Runners terminated

- Resources freed

- Next workflow can run

This happens automatically every single time you push code.

WORKFLOW FILE LOCATION

Important:

File must be in .github/workflows/ folder.

Full path:

```
repository-root/
├── .github/
│   └── workflows/
│       └── first-actions.yml
└── src/
    └── addition.py
```

If placed elsewhere:

GitHub will not detect it and workflow won't run.

MULTIPLE WORKFLOW FILES EXAMPLE

Real project structure:

```
.github/workflows/
├── ci-build.yaml        (Build and test)
├── codeql.yml           (Security scanning)
├── image.yaml           (Docker image build)
├── pr-title-check.yml   (PR validation)
├── release.yaml         (Release automation)
```

└── update-snyk.yaml   (Dependency updates)

Each file:

- Independent workflow
- Different trigger
- Different purpose
- Runs independently

BENEFITS OF MULTIPLE FILES

Organization: Each workflow has single responsibility.

Clarity: Easy to understand what each workflow does.

Maintenance: Easy to update individual workflows.

Flexibility: Can enable/disable specific workflows.

Performance: Only relevant workflows run for each event.

ADVANCED WORKFLOW EXAMPLE

Multi-job workflow:

yaml

```
name: Complete CI/CD

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

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

```
  strategy:
    matrix:
      python-version: [3.8, 3.9, 3.10]
  steps:
    - uses: actions/checkout@v3
    - uses: actions/setup-python@v2
      with:
        python-version: ${{ matrix.python-version }}
    - run: pip install pytest
    - run: pytest tests/

lint:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v3
    - uses: actions/setup-python@v2
    - run: pip install flake8
    - run: flake8 src/

security:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v3
    - uses: actions/setup-python@v2
    - run: pip install bandit
    - run: bandit -r src/

deploy:
```

```
    needs: [test, lint, security]
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'
    steps:
      - uses: actions/checkout@v3
      - name: Deploy to production
        run: ./deploy.sh
```

This workflow:

- Tests on 3 Python versions
- Runs linting
- Runs security scan
- Deploys only if all pass and on main branch