Module 4: AST-1

TITLE: Project with GitHub Repo and Continuous Integration

LEARNING OBJECTIVES

You will be able to understand and implement the following aspects:

1. Connect local repo with GitHub remote repo
2. Understanding & using cloud development environment: GitHub Codespaces and AWS Cloud9
3. Using Linux Command Line
4. Continuous Integration: GitHub Actions for automated training and testing.

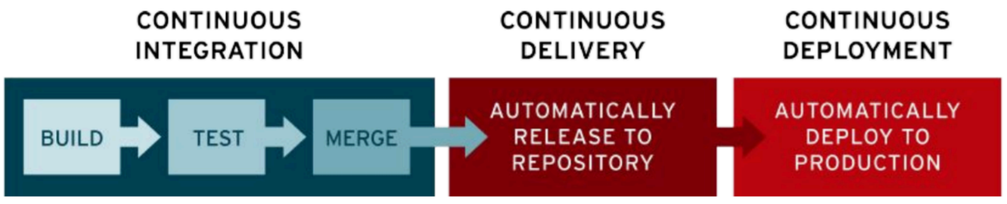
INTRODUCTION

CI/CD

CI/CD stands for continuous integration and continuous deployment, which is fundamentally, about automating the stages of development and deployment.

The diagram below shows an overview of these stages. The **Build** refers to preparing everything need to run the code. In the case of Python, that's installing dependencies in the requirements.txt file and making sure that the operating system that's going to run the code is set up or the virtual machine or container is built and ready to go. The **Test** refers to automatically testing the code.

The **Merge** refers to merging in a feature branch, which automatically releases that code either to production or a testing environment where it can undergo further tests, or in the case of deploying automatically to production, then immediately be monitored and used by real users.



Automating all these steps ensures there's no need for a person to ever run a script or SSH into a machine, and that adds a huge amount of value. Why does automating these steps matter? Adopting this practice means that the system is always in a releasable state and allows us to react to issues quickly. Also, this ensures that the chances of breaking something upon changes or doing something that interferes with another system are reduced. This is more valuable for enterprises having regular and faster release cycles i.e. the code is being put out into production every day, or at least on a regular basis.

GitHub Actions

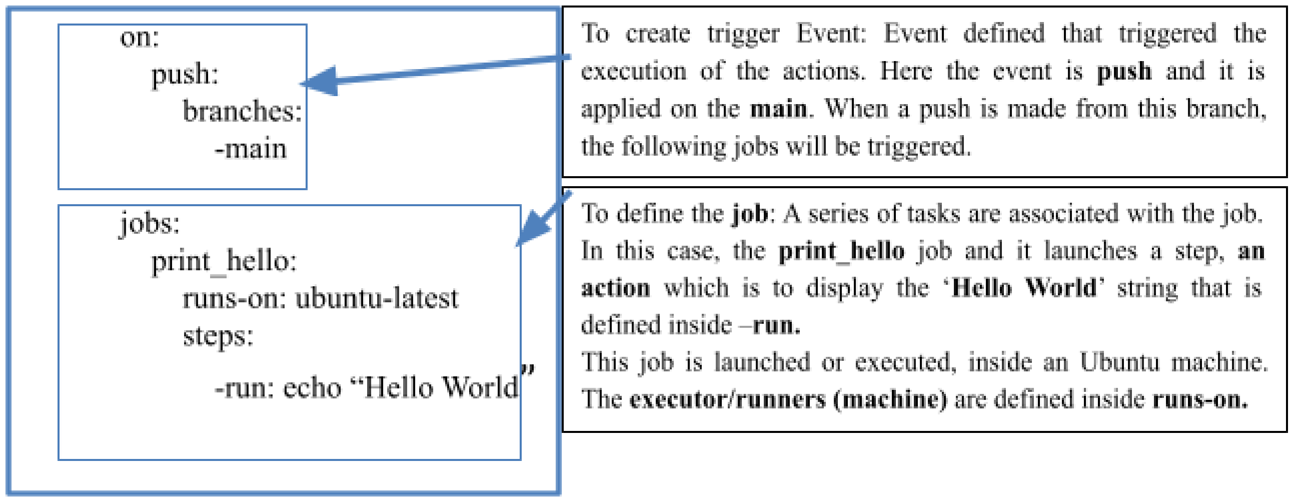
GitHub Actions is a platform that automates the creation, testing, and deployment of software. It also allows us to execute any code when a certain event occurs. The following three are the main components of GitHub actions, and this gives an understanding of the complete picture:

1. **Events:** An event refers to any action or occurrence that can take place within the repository. These events encompass a wide range of activities, such as submitting code changes, creating new branches, opening merge requests, commenting on issues, and more. Essentially, an event is anything that can trigger an action or initiate a process within the GitHub environment.

2. **Workflows:** A workflow is an automated process that is made up of a series of jobs that are executed when triggered by an event. Workflows are defined in YAML files and stored in a directory called GitHub Workflows.

3. **Jobs:** Jobs are a series of tasks that are executed within a workflow when triggered by an event. Each step in a script or a GitHub action would be a job at the end.

Basic workflow: The minimum components needed inside a GitHub action workflow are shown in figure below. This is an example of a workflow YAML file.



```
on:
    push:
        branches:
            -main

jobs:
    print_hello:
        runs-on: ubuntu-latest
        steps:
            -run: echo “Hello World”
```

To create trigger Event: Event defined that triggered the execution of the actions. Here the event is **push** and it is applied on the **main**. When a push is made from this branch, the following jobs will be triggered.

To define the **job**: A series of tasks are associated with the job. In this case, the **print_hello** job and it launches a step, **an action** which is to display the **‘Hello World’** string that is defined inside –**run.**
This job is launched or executed, inside an Ubuntu machine. The **executor/runners (machine)** are defined inside **runs-on.**

Explanation of different components:

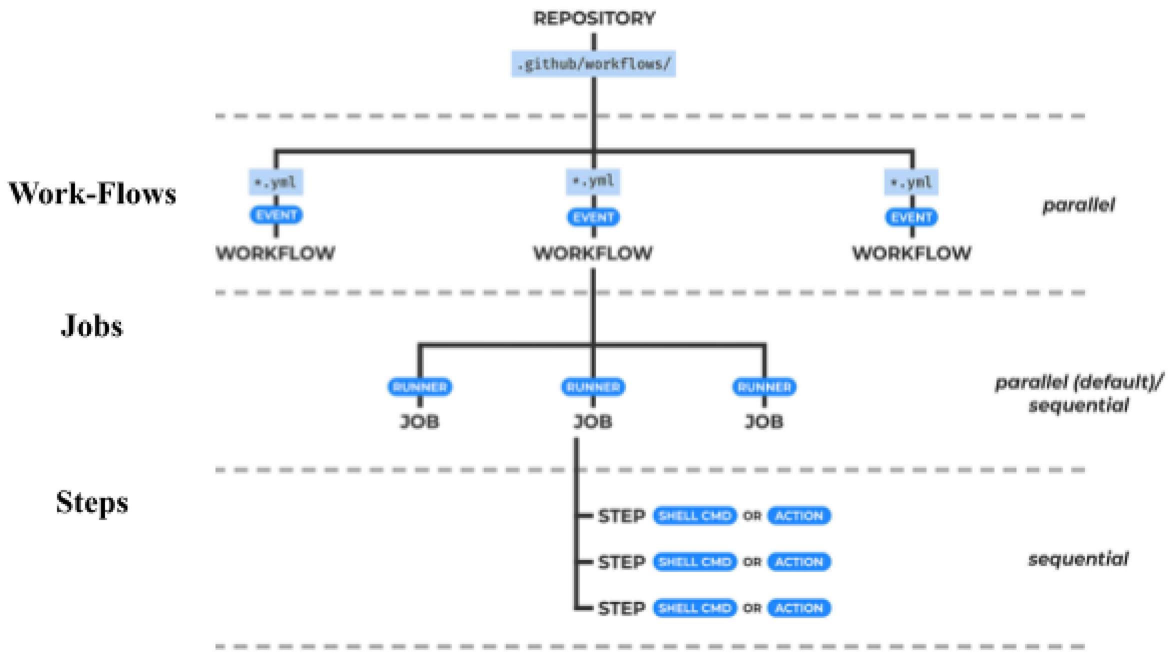
- **on** The event is defined that triggers the workflow.
- **runs on**: It indicates the machine where each job should be executed.
- **jobs** It contains the list of jobs that make up the workflow.
- **steps:** It contains a series of tasks that are executed within each job.

- **run** It indicates what is going to be executed.

Actions: An action is a command that is executed on a runner, the core element of GitHub Actions, which is named after it.

Runners: A runner is a GitHub Actions server. It listens for available jobs, runs each in parallel, and reports back progress, logs, and results. Each runner can be hosted by GitHub or self-hosted on a localized server. GitHub Hosted runners are based on Ubuntu Linux, Windows, and macOS.

GitHub Action Diagram



The diagram above depicts the GitHub actions. There is a GitHub repository and within that repository, there is a GitHub workflows folder. There are different workflows (separate YAML files) inside the GitHub workflows folder. The specified event is going to trigger a different workflow.

A workflow can have several jobs that run in parallel. Here, in the diagram, we can see a single workflow having three jobs running in parallel. Then within the job, we have different actions or steps. These actions, in the end, are individual tasks that are within a job. These actions can be very different and range from, for example, publishing a Python package in PyPI or sending a notification email to configuring Python to a specific version before running a particular script.

Tools for Linting and Formatting

Linting highlights syntactical and stylistic problems in the Python source code, which often helps in identifying and correcting subtle programming errors or unconventional coding practices that can lead to errors. For example, linting detects the use of an uninitialized or undefined variable, calls to undefined functions, missing parentheses, and even more subtle issues such as attempting to redefine built-in types or functions. We are going to use the **‘pylint’** package as a linting tool.

Formatting makes code easier to read by humans. It applies specific rules and conventions for line spacing, indents, spacing around operators, and so on. Keep in mind, formatting doesn't affect the functionality of the code itself. We are going to use the ‘**black**’ package as a formatting tool and it not only reports format errors but also fixes them automatically.

Linting is distinct from formatting because linting analyzes how the code runs and detects errors, whereas formatting only restructures how code appears**.** Although there is a little overlap between formatting and linting, the two capabilities are complementary. The purpose of any kind of tooling like this is to manage the package, make the code easy for other people to use, and also reduce the chances of introducing bugs.

PROCEDURE

To conduct this experiment, familiarity with **Git and GitHub** accounts is required. Please review the reference document on the same topic [**Git Steps & Commands**]. The runtime for all cloud platforms is almost guaranteed to be the **Linux operating** system. Considering the same we need to set up the Linux terminal in the local VS-Code. Follow the document [**Run Linux Terminal**] for setting the **Linux terminal in VS-code**. A few frequently used **Linux commands** are provided in the document [**Bash Shell & Commands**]. We don't have to do any complex shell scripting, knowing a few basic commands are sufficient for our purpose. First, we will conduct this experiment using local VS-code , repo, and GitHub repo.

The conducting of this experiment with cloud platforms like **AWS Cloud9**, **GitHub Codespaces**, etc. remains the same, except that we have to follow a few extra steps for setting up the cloud accounts and environment. We will demonstrate that as well. Follow the document [**AWS Account creation**] for creating an **AWS account**.
Follow the document [**GitHub codespaces**] for getting started with **GitHub Codespaces**.

A. Experiment with local VS-Code, repo and remote GitHub repo

Steps:

1. Go to your **GitHub account** and create a repo with a name, say '**github_action_demo**', choosing 'Add a README file' and 'Add .gitignore' with the Python template option and keeping all other default settings as it is.

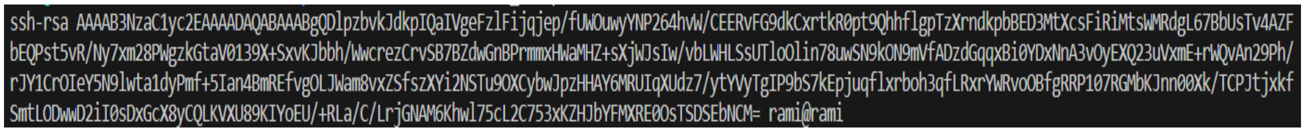
2. Open the VS-Code and click on the Left bottom corner (blue colored box) in the VS-Code window and choose 'connect to wsl using distro….' and select the one in your system. Now you can see the terminal is opening in the Linux environment. In my case it is like:
   - rami@rami: ~$
   - rami@rami: ~$ pwd

   which returns: /home/rami [This is the Linux home directory in my system.]

3. We want to clone the '**github_action_demo**' repo to the desktop. Need to change the directory that points to the desktop like this:
   - rami@rami:~$ cd /mnt/c/users/karna/desktop

   Now we are inside desktop like this:
   - rami@rami:/mnt/c/users/karna/desktop$

4. **Generating SSH Key for Authentication of GitHub connection with the system:**
   - rami@rami:/mnt/c/users/karna/desktop$ ssh-keygen -t rsa
     Read the message and keep the default setting, press enter three times.
     Copy the public key location and run the below command
   - rami@rami:/mnt/c/users/karna/desktop$ cat /home/rami/.ssh/id_rsa.pub
     Copy the entire output which would be something like this:



```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDlpzbvkJdkpIQaIVgeFzlFijqjep/fUWOuwyYNP264hvW/CEERvFG9dkCxrtkR0pt9QhhflgpTzXrndkpbBED3MtXcsFiRiMtsWMRdgL67BbUsTv4AZF
bEQPst5vR/Ny7xm28PWgzkGtaV0139X+SxvKJbbh/WwcrezCrvSB7BZdwGnBPrmmxHWaMHZ+sXjWJsIw/vbLWHLSsUTloOlin78uwSN9kON9mVfADzdGqqxBi0YDxNnA3vOyEXQ23uVxmE+rWQvAn29Ph/
rJY1CrOIeY5N9lwta1dyPmf+5Ian4BmREfvgOLJWam8vxZSfszXYi2NSTu9OXCybwJpzHHAY6MRUIqXUdz7/ytYVyTgIP9bS7kEpjuqflxrboh3qfLRxrYWRvoOBfgRRP107RGMbKJnn00Xk/TCPJtjxkf
SmtLODwwD2iI0sDxGcX8yCQLKVXU89KIYoEU/+RLa/C/LrjGNAM6Khwl75cL2C753xKZHJbYFMXRE0OsTSDSEbNCM= rami@rami
```

Go to your GitHub account → Setting → SSH & GPG keys → New SSH key

→Provide **Title** as 'vscode_pc' and paste that copied text from above rsa.pub inside **Key box**. Keep **Key type** as **Authentication Key**. Finally click **Add SSH key**. This will prompt you to provide a password for your GitHub account.

5. **Cloning the GitHub repo**: Go to the particular repo in github [‘**github_action_demo**’] and copy the clone url for ssh: click on → **Code** → SSH → and copy the URL.

   Finally run the following command in terminal: git clone _copied_url_
   Like this:

   - `rami@rami:/mnt/c/users/karna/desktop$ git clone git@github.com:Rami-RK/github_action_demo.git`

   Now you can see the ‘github_action_demo’ folder on the desktop.

6. In **VS-Code → File → Open Folder**, and select the **‘github_action_demo’** folder from the desktop. Open the terminal. Initially, the terminal is opened in a window environment. Type **wsl** and enter. Now you can see the terminal is Linux env. and pointing to the repo.

   Now we are ready to create and write different sub-folders, scripts-that can be executed and pushed back to GitHub remote Repo.

7. **Creating and activating virtual environment:**

   - `…..$ python3 -m venv venv`
   - `…..$ source venv/bin/activate`

   Here, a virtual environment has been created and activated. You can notice the → (venv) at the start of the new command line.

8. **Creating requirement file and installing the requirements**: Run the command below:

   - `…..$ touch requirements.txt`

   A blank text file is created. Open that in vs code, type these libraries as given below and save it.

```
# requirements.txt
black
pylint
pytest
```

   Check the installed packages in venv:
   - `…..$ pip freeze`

   Nothing will be displayed. Now install the requirement:

- …..$ pip install -r requirements.txt

Check the installed packages in venv:

- …..$ pip freeze

Note: Ctrl +L command clears the terminal screen.

9. Writing python script:

- …..$ touch script.py

A blank python file is created. Open that in vs code, type the code given below and save it.

```
# script.py
def addition (A, B):
    return A + B
```

Create a blank __init__.py file so that the folder can be treated as a module and import is possible.

- …..$ touch __ini__.py

10. Creating scripts for test cases:

- …..$ touch test_{1,2}.py

Open these files in vs code, type the code given below and save it.

```
# test_1.py
import os, sys
sys.path.insert(0, os.getcwd())

from script import addition
def test_add():
    subj = addition(2, 3)
    assert subj == 5
```

```
# test_2.py
import os, sys
sys.path.insert(0, os.getcwd())

from script import addition

def test_data_type():
    subj = addition(2, 3)
    assert isinstance(subj, int)
```

11. **Creating test directory and moving test cases inside that**:

- …..$ mkdir tests

Notice a blank folder name 'tests' is created. Now the following command moves test files into the tests folder.

- …..$ mv test*.py ./tests

Go inside the tests folder and you can see test_1.py and test_2.py files inside it.

12. **Running the test cases**: Pytest automatically is able to find a folder with name as '**test**' and run scripts starting with **'test'.**

- …..$ pytest

You will notice the result of test cases.

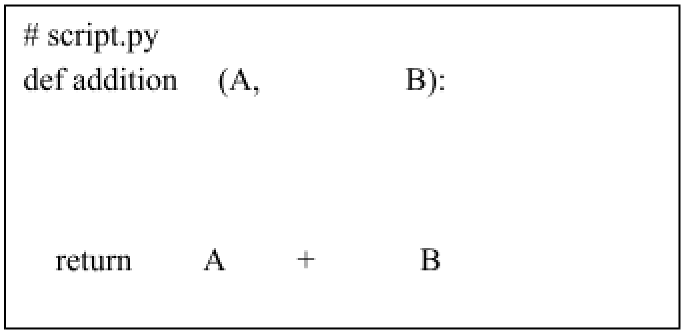
13. **Understanding linting and formatting**: Run the command below for linting:

- …..$ pylint *.py

You can see the complaints about style guides and conventions of best practices of writing codes. Now run the command below for formatting:

- …..$ black *.py

Notice the result above and re-write the code in script.py as given below and save it.



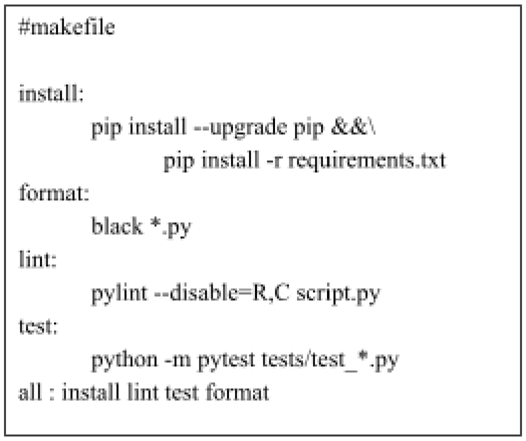
```
# script.py
def addition    (A,              B):



    return      A      +          B
```

Now run the command below for formatting and you will notice the code is reformatted automatically.

- …..$ black *.py

14. **Understanding and creating makefile**: Create a file with a name '**makefile**' without any extension, open and write the syntax as given below:

- …..$ touch makefile



```
#makefile

install:
        pip install --upgrade pip &&\
                pip install -r requirements.txt
format:
        black *.py
lint:
        pylint --disable=R,C script.py
test:
        python -m pytest tests/test_*.py
all : install lint test format
```

A Makefile runs “recipes” via the Make system, which comes with Unix-based operating systems. Therefore, a Makefile is an ideal choice to simplify the steps involved in continuous integration and saves us from writing the same tedious code again and again.

Run the commands like: …..$ make test

Try all commands given below, explanation is written.

- make install: This step installs software via the make install command
- make format: This step reformats the code if required via make format command.
- make lint: This step checks for syntax errors via the make lint command
- make test This step runs tests via the make test command:
- make all: This step runs all above step one by one in one make all command.

15. **Pushing all the changes in local repo into remote GitHub repo**: Whatever files and folders have been created, those are in local repo only and need to be pushed into GitHub repo. First check the remote repo, it contains only readme and gitignore file. Now, run the following syntax and the local repo is pushed into the remote repo.

    - …..$ git status [Get information about untracked files and folders]
    - …..$ git add . [Staging the untracked files and folders]
    - …..$ git commit -m “Adding all contents” [Snap-shot of staging contents]
    - …..$ git push [Finally pushing all contents from local to remote repo]

    Now again check the remote repo, all contents are pushed.

16. **GitHub Action**: The YAML file for GitHub workflow is given in the next page. Go to the ‘**github_action_demo**’ folder inside **GitHub** and click on **Actions.** There are many suggested templates, but we are going with the set up a workflow yourself → option. Click on it. It will open a blank yml file inside ‘**github_action_demo/.github/workflows’** directory. We can name the .yml file, say it is **demo.yaml**. Now copy the code from the given workflow in the next page and paste inside that blank yml file. Now click on ‘**Commit changes.** Select **‘Commit directly to the main branch’.** Again, click on

Actions click on **Create demo.yml** click on **3 jobs completed.** click on any one

build You can see each step and actions executed.

Now go to the repo and notice a blue tick **(√).** Click on it. It would display “All checks have passed”. We can see the details as well.
Note make file is not necessary for creating GitHub workflow. Instead of make we can directly write syntax on workflow yml file. GitHub workflow without make is implemented in the titanic example.

```
# demo.yaml. This workflow will install Python dependencies, run tests with a variety of
# Python versions
name: Python app
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      fail-fast: false
      matrix:
        python-version: ["3.8", "3.9", "3.10"]
    steps:
    - uses: actions/checkout@v3
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v3
      with:
        python-version: ${{ matrix.python-version }}
    - name: Install dependencies
      run: |
        make install
    - name: Lint with pylint
      run: |
        make lint
    - name: test with pytest
      run: |
        make test
    - name: Format code with Black
      run: |
        make format
```

To update the local repo as per the changes in the remote repo run: `…..$ git pull`

Whenever there is any push on the GitHub repo, the GitHub workflow would be triggered to run all the tests along with all other actions. To check this, update the readme doc in the local repo like given in the text box below. Open readme.me file in vs code and type the explanation below and save it.

```
### github_action_demo
This repo is created for the demonstration of GitHub actions.
Connecting local repo with GitHub repo and working the vs code Linux terminal.
```

Now run the following syntax:

- `…..$ git status`
- `…..$ git add .`
- `…..$ git commit  -m "Explanation added in readme"`
- `…..$ git push`

Open the GitHub actions you will see a new workflow has started to run. So, every push or commit in the remote repo will trigger the workflow.

The complete example above is a demonstration of Continuous integration (CI) along with the best practices of writing code in a production environment. This is a solid foundation for further implementation of CI in any other projects.

In a nutshell, CI is the process of continuously testing a software project and improving the quality based on these test results. It is automated testing using open source and SaaS build servers such as GitHub Actions, Jenkins, Gitlab, CircleCI, or cloud-native build systems like AWS Code Build. We have used GitHub Actions in our demonstration.

17. Now we will implement the CI concept in the **'Titanic Project'** from **Module 3**. Since the project is ready and we don't have to write any new script, we have to push it into our GitHub account and write the GitHub Action workflows. Follow the **steps** given below:

   a. Download the project folder [project_with_test]in your local system.
   b. Go to your GitHub account and create a repo having the name '**MLOps**' with readme and .gitignore (choose python template) file.
   c. Clone the remote **'MLOps**' folder in your local system and move the local project folder inside this **'MLOps**' folder. Finally, push this folder into the remote GitHub repo. You can create a virtual environment, run a train pipeline and carry out all the tests inside VS-Code as done previously.

d. Go to the remote repo (inside **'MLOps'**) and through the **Action** option create the following workflow yml file with the name **'python-package'** and commit it.

```
# This workflow will install Python dependencies, run tests with a variety of
# Python version.
name: Python package
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:


    runs-on: ubuntu-latest
    strategy:
      fail-fast: false
      matrix:
        python-version: ["3.8", "3.9", "3.10"]

    steps:
    - uses: actions/checkout@v3
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v3
      with:
        python-version: ${{ matrix.python-version }}
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r project_with_test/requirements/test_requirements.txt
    - name: train_pipeline
      run: |
          python project_with_test/titanic_model/train_pipeline.py
    - name: test with pytest
      run: |
        pytest
```

Now click on the Action option again and check the workflow run. Note that here, we are not using Makefile. You can try with Makefile on your own.

Be careful Linux commands are different from Windows powershell commands, and the same command may behave differently in both environments.

B. Experiment with AWS Cloud9, Git & GitHub

Steps:

1. Log in to AWS and search for Cloud9 → Click on **Create environment.**
2. Fill the informations: **Name** say 'model_dev' **; Description** say 'ci implementation'; **New EC2** → choose **t2.micro; Amazon Linux2;** keep all other settings as default and click on → **Create** **.**
   It will take some time to create the '**model_dev**' environment. You can see - **Successfully created model_dev** message.
3. Click on → **Open** [Option Available below the text **'Cloud9 IDE**]
   It will open an IDE almost equivalent to VS-Code.
4. Go to the → File → Upload Local Files →
   Select the 'project_with_test' folder from your local system.
5. Now think as if this Cloud9 IDE is your local system. Authenticate the communication between Cloud9 to GitHub Repo by SSH method as demonstrated in part A.
6. Now follow everything from part A 17. b.
7. We have not implemented formatting, linting tools and Makefile. Try on your own.

8. **Caution! This step is important to complete, or else the EC2 instance keeps running and incurs unnecessary cost.**
   After completing the experiment, delete everything. On the top right corner of Cloud9 IDE, there is a **circular account icon** (on the left of the share option) → **Click on** that and choose **'Manage EC2 instance'** → **Check the square box** in front of the instance name → Go to the **Instance state** and chose → **Stop the instance.** Go to the **AWS Cloud9** window and **delete** the instance**. Finally logout.**