

Automating Unit Test Execution, Coverage Analysis, and Performance Tracking

Contents

1. Objective:	2
2. Tools and Dependencies	2
3. Code and Test cases	3
4. Source Code Management	4
5. CI/CD Pipeline	4
5.1 Pull code	4
5.2 Dependency installation	4
5.3 Run code and Unit Test	5
5.4 Code quality Details.....	6
6. Cron Job Setup	7
7. Data collection.....	7
8. Data Visualization	8

1. Objective:

Create a DevOps pipeline to perform below things

1. **Automate Unit Test Execution** every 12 hours.
2. **Collect Code Coverage Metrics.**
3. **Track Execution Time of Tests.**
4. **Store Data in Prometheus and Visualize in Grafana.**
5. **Integrate SonarQube for Code Quality Analysis.**

2. Tools and Dependencies

Item	Item	Description
Code language	Python	Coding language
Test framework	Pytest	Library to automate the test cases
Source Code Management	git	To manage versions of code
CI/CD pipeline	Jenkins	To manage CI/CD
Code quality test	Sonar Qube	To run and test code quality
Matrix collection	Prometheus	To store results of pipeline w.r.t. code quality, execution time etc.
Matrix Visualization	Grafana	To Visualize the data and matrix of code quality

3. Code and Test cases

For this pipeline we will use a sample python code which performs some mathematical operations, each operation is created as module. To test pipeline we will add some cases where the module will fail.

Example code : app.py

```
✓ def read_file_safe(filename):  
    """Safely reads a file and prevents path traversal."""  
    ✓ if ".." in filename or filename.startswith("/"):   
        raise ValueError("Invalid file path!") # ✓ Prevents directory traversal  
    ✓ with open(filename, "r") as file:  
        return file.read()  
  
    ### Vulnerable / Bad Practice Functions ###  
    # Insecure function: Uses eval() leading to command injection  
    ✓ def insecure_eval(user_input):  
        return eval(user_input) # | HIGH-RISK: Arbitrary code execution
```

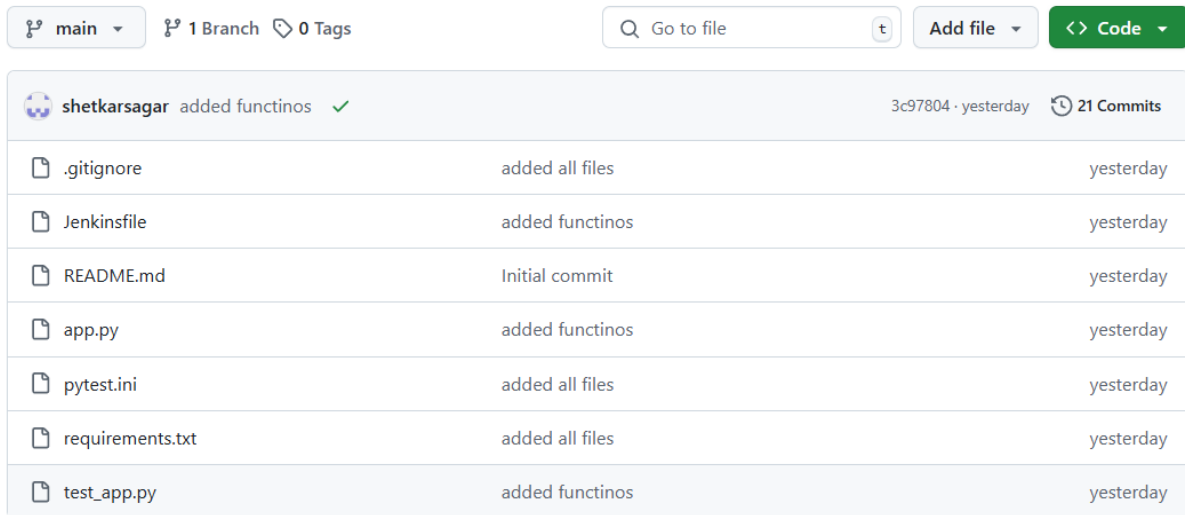
We will create a test cases for each of the modules in python ,

```
def test_read_file_safe():  
    with pytest.raises(ValueError):  
        read_file_safe("../etc/passwd") # Prevents directory traversal  
  
### Test Cases for Bad Functions ###  
def test_insecure_eval():  
    assert insecure_eval("2 + 2") == 4 # Dangerous function
```

as seen in above image we have added test cases for each of the modules.

4. Source Code Management

We will use git for source code management, and we will commit a sample repo to git with app.py file which contains all python modules and pytest framework related files



The screenshot shows a GitHub repository interface. At the top, there's a navigation bar with 'main' branch selected, '1 Branch', and '0 Tags'. A search bar 'Go to file' and buttons 'Add file' and 'Code' are also visible. Below this, the repository name 'shetkarsagar' is shown with a commit message 'added functinos' and a green checkmark. The commit hash '3c97804' and 'yesterday' are also present, along with '21 Commits'. A table below lists the files added in this commit:

File	Commit Message	Time
.gitignore	added all files	yesterday
Jenkinsfile	added functinos	yesterday
README.md	Initial commit	yesterday
app.py	added functinos	yesterday
pytest.ini	added all files	yesterday
requirements.txt	added all files	yesterday
test_app.py	added functinos	yesterday

5. CI/CD Pipeline

We will use Jenkins for CI/CD pipeline we can use git action tool ,for Jenkins we will create a file in our repo jenkinsfile, for this we will use Jenkins Declarative pipeline which will handle all of the requirements . Below we will add different stages.

5.1 Pull code

this stage is responsible for pulling code from git

```
stage('Checkout Code') {
    steps {
        git branch: 'main', url: 'https://github.com/shetkarsagar/UnitTestCode.git'
    }
}
```

5.2 Dependency installation

In this step we will install all the libraries required for running python code and its test cases

```
stage('Install Dependencies') {
    steps {
        bat '%PYTHON_PATH%' -m pip install -r requirements.txt
    }
}
```

5.3 Run code and Unit Test

We will use pytest to run the unit test cases and collect test results.

```
stage('Run Unit Tests') {
    steps {
        script {
            def startTime = System.currentTimeMillis()
            def pytestResult = bat(
                returnStdout: true,
                'python -m pytest --junitxml=pytest-report.xml --cov=my_project --cov-report=xml:coverage.xml'
            ).trim()
            def endTime = System.currentTimeMillis()
            def duration = (endTime - startTime) / 1000
            echo "Execution Time: ${duration} seconds"

            def passedTests = pytestResult.count('passed')
            def failedTests = pytestResult.count('failed')

            // Write Prometheus Metrics
            writeFile file: PROMETHEUS_METRICS_PATH, text: """
jenkins_unit_test_execution_time ${duration}
jenkins_unit_test_passed ${passedTests}
jenkins_unit_test_failed ${failedTests}
            """
        }
    }
}
```

As seen in above image during running of code and test cases we will collection information's such as

- jenkins_unit_test_execution_time – time taken to execute code
- jenkins_unit_test_passed - Get number of test cases passed
- jenkins_unit_test_failed – get number of test cases failed

along with above matrices we will need information's such as code coverage , code_bugs , vulnerabilities and code smells in order to collect this we will Sonar Qube service to which we will create a sonar Qube project for current project.

Sonar Qube needs access to repo on which these analysis is ran, once the setup is done we will add a stage in Jenkins pipeline to get this information from SonarQube .

5.4 Code quality Details

To get code quality first get the access token in **Sonar Qube** and save it in credential management in Jenkins and call an API from Sonar Qube fetch all of the details.

As we need to store this into Prometheus we will store this as text format by maintaining the syntax of Prometheus. Also collect extra parameters which we need to monitor.

```
stage('Fetch SonarCloud Metrics') {
    steps {
        script {
            def sonarResults = bat(
                returnStdout: true,
                script: '''
                    curl -u ${SONAR_TOKEN}: ^
                    "https://sonarcloud.io/api/measures/component?component=shetkarsagar_UnitTestCode^&metricKeys=coverage,bugs,vulnerabilities,code_smells"
                    '''
            ).trim()

            def coverage = sonarResults.find(/"coverage": "(\\d+\\.\\d+)"/) ? sonarResults.find(/"coverage": "(\\d+\\.\\d+)"/)[1] : 0
            def bugs = sonarResults.find(/"bugs": "(\\d+)"/) ? sonarResults.find(/"bugs": "(\\d+)"/)[1] : 0
            def vulnerabilities = sonarResults.find(/"vulnerabilities": "(\\d+)"/) ? sonarResults.find(/"vulnerabilities": "(\\d+)"/)[1] : 0
            def codeSmells = sonarResults.find(/"code_smells": "(\\d+)"/) ? sonarResults.find(/"code_smells": "(\\d+)"/)[1] : 0

            writeFile file: 'execution_metrics.prom', text: """
                jenkins_code_coverage ${coverage}
                jenkins_code_bugs ${bugs}
                jenkins_code_vulnerabilities ${vulnerabilities}
                jenkins_code_smells ${codeSmells}
            """

            echo "SonarCloud Metrics Collected: Coverage=${coverage}%, Bugs=${bugs}, Vulnerabilities=${vulnerabilities}, Code Smells=${codeSmells}"
        }
    }
}
```

And now final stage is to copy the result of file into a path where our Prometheus is configured to fetch these details.

```
stage('Publish Prometheus Metrics') {
    steps {
        bat '''
            copy %PROMETHEUS_METRICS_PATH% \\data\\jenkins_metrics.prom /Y
        '''
    }
}
```

With all the above steps we will create a Declarative Jenkins file called jenkinsfile and push this code to git hub,

6. Cron Job Setup

In Jenkins create a declarative pipeline which can fetch code from the repo on which this unit code needs to be ran. As it will be declarative we will mention the git repo and branch on which we need to run the pipeline, we will use **origin** branch for running pipeline.

As one of the objective is to run the pipeline every 12 hour we will add a trigger in Jenkins for the same . This will make sure this pipeline will run for every 12 hours

Triggers

Set up automated actions that start your build based on specific events, like code changes or scheduled times.

☐ Build after other projects are built ?

☒ Build periodically ?

Schedule ?

H */12 * * *

Would last have run at Monday, March 10, 2025, 12:43:07 AM India Standard Time; would next run at Monday, March 10, 2025, 12:43:07 PM India Standard Time.

☐ GitHub hook trigger for GITScm polling ?

7. Data collection

For data collection we will use Prometheus and as a visualization tool we will use Grafana.

Data Collection :

Configure Prometheus to read the data from output file , As we are using Jenkins in localhost.

Configuration example is:

```
scrape_configs:
  - job_name: 'jenkins'
    metrics_path: '/prometheus'
    static_configs:
      - targets: ['localhost:8080']
```

Note : this configuration changes as per Jenkins.

Once the configuration is done you should see the new Jenkins job in Prometheus as shown below

jenkins		1 / 1 up	
Endpoint	Labels	Last scrape	State
http://localhost:8005/metrics	instance="localhost:8005" job="jenkins"	4.518s ago 17ms	UP

As seen in above image we are successfully pulling the data generated by Jenkins into Prometheus,

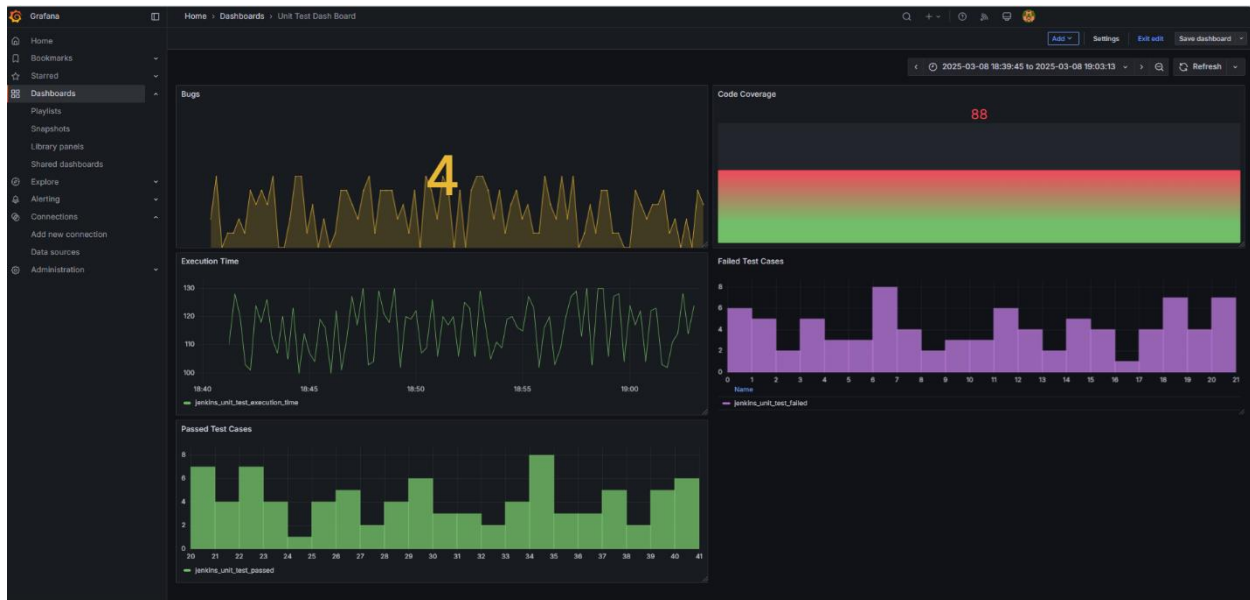
8. Data Visualization

To Visualize the data collected by Prometheus we will use Grafana, As per Jenkins pipeline we are collecting below details

- jenkins_unit_test_execution_time
- jenkins_unit_test_passed
- jenkins_unit_test_failed
- jenkins_code_coverage
- jenkins_code_bugs
- jenkins_code_vulnerabilities
- jenkins_code_smells

In Grafana create a new dash board and create query for each of the above information's.

Below is the sample output from Grafana.



Note: To simulate this graph I have ran the Jenkins pipeline for multiple times where it will generate this matrices and I have used local setup for **Jenkins → Prometheus → Grafana**