DevOps Optimization

Table of Content

1. Introduction

The purpose of this project is to outline a comprehensive plan for DevOps optimization, focusing on standardizing development environments, implementing effective Git hooks, improving code quality, and enhancing documentation. The project aims to streamline the development process, enforce best practices, and establish a robust CI/CD pipeline.

2. Key Objectives

2.1 Standardize Development Environments

Implement a consistent development environment across all developer machines to ensure uniformity and reduce compatibility issues.

2.2 Git Hooks Implementation

2.2.1 Pre-commit Hook

Enforce code formatting standards using Black before each commit to maintain a consistent code style.

2.2.2 Post-commit Hook

Report test coverage and success rates after each commit, providing immediate feedback to developers.

2.2.3 Pre-push Hook

Execute tests (using pytest) and estimate test coverage before each push, preventing the integration of faulty code.

2.3 Code Quality Standards

Adhere to Ruff lint rules for various aspects, including code formatting, documentation, code usage, unused imports, sorting of imports, Pydoc style, and test coverage.

2.4 GitHub Workflows

Implement workflows for testing before merging pull requests to the MAIN branch and regular testing of the DEV and MAIN branches.

2.5 Pull Request Merge Criteria

Ensure that pytest tests pass without failures and Ruff linter produces no violations before allowing pull request merges.

2.6 Automated Documentation and Docstring Checks

2.6.1 Documentation Generation

Utilize Sphinx or Jekyll to automatically generate documentation on push events, enhancing accessibility.

2.6.2 Publish Generated Documentation

Publish the generated documentation on a private webpage (e.g., GitHub Pages) for easy access by internal and external stakeholders.

2.6.3 Docstring Coverage Checks

Enforce minimum docstring coverage thresholds using tools like docstrings-linter, failing the build if coverage falls below the threshold.

3. Phase 1

All the areas listed below have been successfully implemented in Phase 1. Updates have been communicated to Jean, and modifications requested by him have been completed in this phase. More detailed information about the MVP document is provided in the relevant section.

****These implementations are successfully up and running on the company repositories.****

3.1 Standardized Development Environments

- Successfully implemented a consistent development environment across all developer machines.
- Conda environment creation, dependency installation, and setting up git hooks and workflows automated using a single script.

3.2 Git Hooks Implementation

Successfully implemented below hooks:

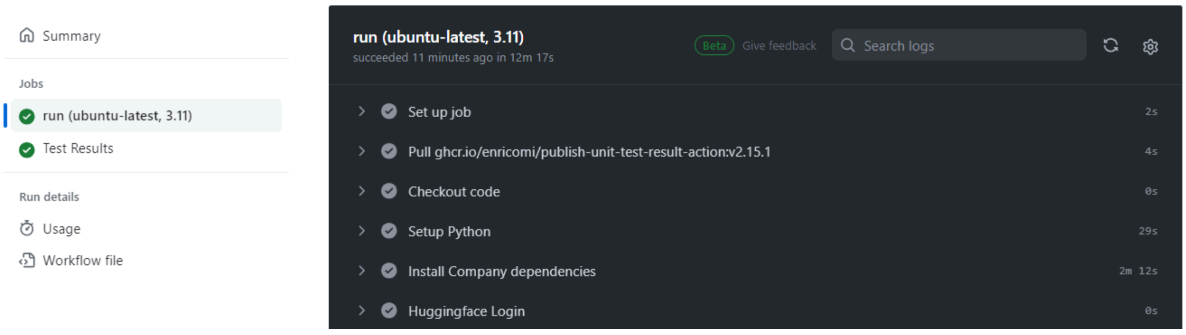
- Pre-commit Hook: Enforcing code formatting standards using Black before each commit.
- Post-commit Hook: Reporting test coverage and success rates after each commit.
- Pre-push Hook: Executing tests (using pytest) and estimating test coverage before each push.

3.3 Code Quality Standards:

Adhering to Ruff lint rules for code formatting, documentation, code usage, unused imports, sorting of imports, Pydoc style, and test coverage.

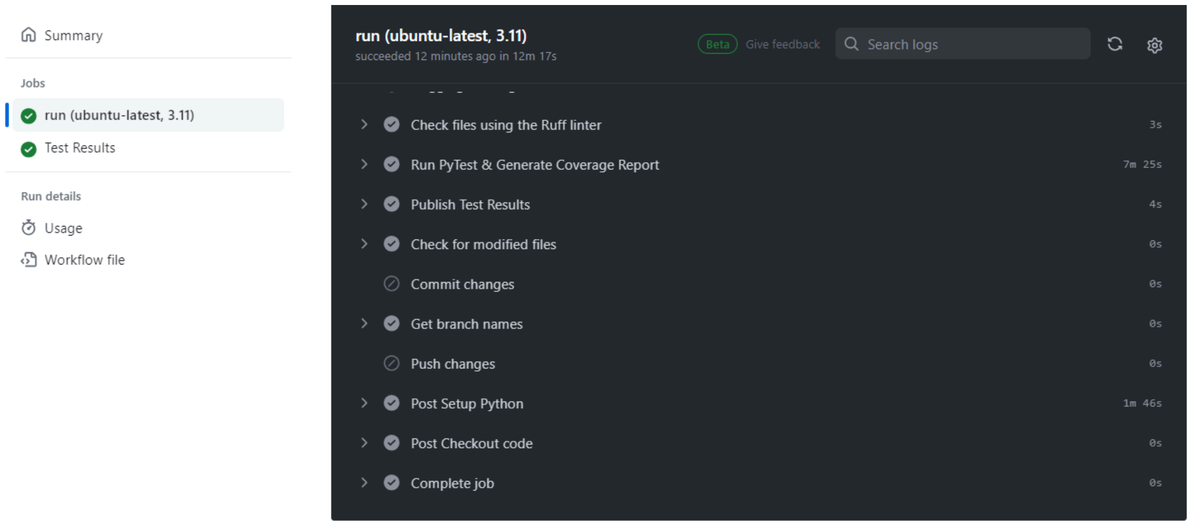
3.4 GitHub Workflows

- Workflows implemented for testing before merging pull requests to the MAIN branch and regular testing of the ML_dev and dev branches.
- Scheduling of workflows also completed.



run (ubuntu-latest, 3.11)
succeeded 11 minutes ago in 12m 17s

```
Set up job                                                   2s
Pull ghcr.io/enricomi/publish-unit-test-result-action:v2.15.1 4s
Checkout code                                                0s
Setup Python                                                29s
Install Company dependencies                            2m 12s
Huggingface Login                                            0s
```



run (ubuntu-latest, 3.11)
succeeded 12 minutes ago in 12m 17s

```
Check files using the Ruff linter          3s
Run PyTest & Generate Coverage Report      7m 25s
Publish Test Results                       4s
Check for modified files                   0s
Commit changes                             0s
Get branch names                           0s
Push changes                               0s
Post Setup Python                          1m 46s
Post Checkout code                         0s
Complete job                               0s
```

3.4.1 Workflow Overview

- Workflow triggers on pull request events (opened, reopened, synchronize) and push events to specific branches.
- Python environment setup, Company dependencies installation, and Huggingface login performed.
- Black formatter, Ruff linter, PyTest, and coverage checks executed.
- Test results published and changes committed and pushed.
- API documentation generated and deployed to GitHub Pages.

3.4.2 Branches Covered in Workflow

- ML_dev, dev, dev_ops, Data-Fenix-patch-1.

3.4.3 Workflow Enhancements

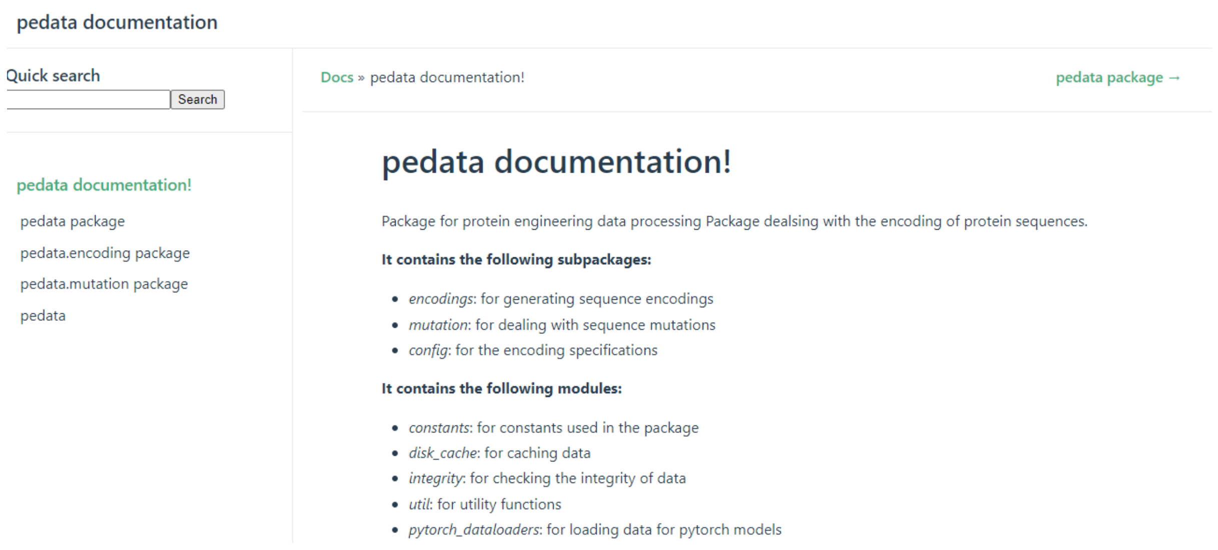
- Environment variable MIN_COV set to 60 for coverage checks.
- Successful integration of various tools for formatting, linting, testing, and documentation generation.

4. Phase 2

4.1 Auto-generating and Publishing Documentation on GitHub Pages

**** After the admin grants the required permissions, only access settings need to be configured. All the necessary code additions have been incorporated into the existing workflows. ****

I successfully implemented auto-generating documentation using Sphinx in my personal repository, with the generated documents stored in the **gh-pages** branch and published on **GitHub Pages**. However, when attempting to replicate this in the company's code repository, I encountered an access issue. To address this challenge, I am requesting the necessary access to resolve the issue and ensure a seamless integration of the documentation generation process into the company's repository.


pedata documentation

Quick search

Docs » pedata documentation!

pedata package →

pedata documentation!

pedata package

pedata.encoding package

pedata.mutation package

pedata

pedata documentation!

Package for protein engineering data processing Package dealsing with the encoding of protein sequences.

It contains the following subpackages:

- *encodings*: for generating sequence encodings
- *mutation*: for dealing with sequence mutations
- *config*: for the encoding specifications

It contains the following modules:

- *constants*: for constants used in the package
- *disk_cache*: for caching data
- *integrity*: for checking the integrity of data
- *util*: for utility functions
- *pytorch_dataloaders*: for loading data for pytorch models

4.1.1 Concerns About Public Access

- Acknowledged that GitHub Pages may make documentation public by default.
- Expressing a desire to ensure that the documentation remains private and exploring options to achieve this.

4.1.2 Confirmation of Access Issue

- Received confirmation that documentation on GitHub Pages was not accessible without being public.
- Mentioned the need to investigate and address this concern in the upcoming week.

4.1.3 Upgrade Required for Private Pages

- Discovered that a repository upgrade is necessary to enable private GitHub Pages.
- Suggested that the company may already have an upgraded (paid) version, and administrators might have the required permissions on GitHub Pages.

4.1.4 Request for Admin Check

Requesting the company admin to check permissions on GitHub Pages and confirm the availability of an upgraded version.

4.1.5 Next Steps:

Once the admin grants the required permissions, only access settings need to be configured. All the necessary code additions have been incorporated into the current workflows.

4.2 Docstring Generation Methods

4.2.1 Research on Docstring Generation

- Explored two docstring generation methods: AI model and VS Code template method.
- The VS Code template method involves manual filling, which might not be ideal.

4.2.2 Preference for AI Model

- Leaning towards using an AI model for docstring generation.
- Acknowledges that it might not be 100% accurate but estimates around 80-90% accuracy.
- Mentioned that using an AI model would require a payment.

4.2.3 Response on VS Code Copilot

- Suggested that, for now, VS Code Copilot is a helpful tool for docstring generation.
- Recognizes the efficiency of LLMs (Large Language Models) for documentation and testing but considers it a topic for future contemplation.

4.2.4 Next Steps

- Await further discussion and input on the preferred docstring generation method.
- Consider cost implications and accuracy trade-offs before making a decision.
- Reflect on the role of LLMs in future development processes.

**** This implementation hasn't been completed as the company deprioritized this step.****

4.3 Development and Testing Section

The workflow checks for modified files, commits changes if present, and runs doctests using PyTest. It also includes environment variables for coverage checks and a list of protected branches to skip certain actions.

4.3.1 Check for Modified Files

- Uses Git to determine if there are modified files in the repository.
- Outputs a boolean variable indicating whether changes are present.

4.3.2 Run DocTest

Executes PyTest with the --doctest-modules option to run doctests in Python modules.

**** The DocTest part is currently commented out since it was developed and tested successfully in the company repository. However, it failed due to improperly written doc tests in the project. Once the doc tests are written properly, the code in Git Workflows can be uncommented, and it will work as expected.****