Lecture 16: Automating Software Development

AC215

Pavlos Protopapas
SEAS/Harvard

Outline

- Linters and formatters
- Testing
- Automating workflows
- Continuous Integration

Outline

- **Linters and formatters**
- Testing
- Automating workflows
- Continuous Integration

Linters and Formaters

Code is meant to be read more than it is executed. Clear, readable code is easier to understand, maintain, and review.

This statement refers to the clarity of the code itself, but also on the way it is written.

Python has standards (e.g., PEP8) for quality code. However, each organization may have its own additional style and formatting standards.

Good code should be well-written in both function and form.

Linters

A linter is a tool that analyzes code for errors, style issues, and compliance to coding standards.

It identifies issues for the developer to review and fix.

Linters help standardize code across teams, making it easier to read, maintain, and collaborate within the organization.

Linters

Why:

Code structure should comply with standards and organizational guidelines.

When:

Linters should be used while coding, before committing changes or during the CI stage.

How:

By executing the scripts manually, or triggered by an automated pipeline.

Example #1

In the following example, we will analyze the mega-pipeline code with linters, in order to have the same style across all the repositories.

Flake8 and pylint are popular Python linters.

Hadolint for Dockerfiles.

Formatters

Formatters enforce code structure by automatically adjusting code to meet style standards, improving readability and consistency.

Linters detect and report style inconsistencies, while formatters modify the code directly to correct these issues.

Popular formatters are Black, autopep8, and YAPF.

Example #2

Running Black, autopep8 and YAPF with custom configuration.

To ignore the limit of characters per line:

- black --line-length 200 . (entire directory)
- autopep8 --max-line-length 200 --in-place --recursive .
- yapf --style='{based_on_style: google, column_limit: 200}' --in-place --recursive .

Outline

- Linters and formatters
- **Testing**
- Automating workflows
- Continuous Integration

Testing

Software testing is the process of checking that a program works as expected and meets its requirements.

Testing can cover functionality, performance, security, and more.

It helps to detect errors early in the development cycle, improving code reliability and maintainability.

Tests motivation

While developing new features is great, skipping tests to speed up development is risky. It can lead to undetected issues and fragile code. This will hurt you in the long term.

Testing ensures that code functions as expected, increasing reliability and maintainability.

Effective testing often requires writing significantly more code for tests than for the original functions. Around ten times more!

Don’t be like this

CrowdStrike, a security firm, deployed an update that disrupted millions of systems, affecting critical sectors like airlines and banks.

Proper testing in the deployment pipelines could have helped prevent this issue.

CrowdStrike IT outage: bug in quality control process caused faulty update

Verdict · Photo by Jessica McGowan/Getty Images

Testing

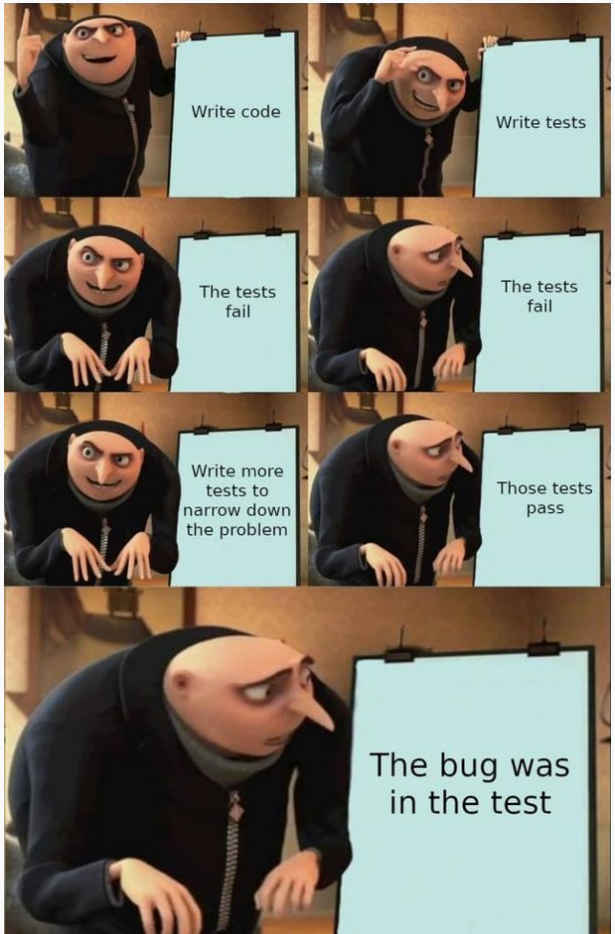
Different types of test exist, designed to check different parts of the entire pipeline.
Among them, we can find:

Unit tests: Tests individual components (like functions or classes) in isolation.

Integration Tests: Checks how different components work together. Unit tests alone cannot guarantee a correct interdependency among units.

System Tests: Validates the entire system's functionality as a whole, simulating a production environment.

Testing: As solid as the test code



Testing

All the code must be tested. This includes:

- Python scripts
- Dockerfiles
- YAML files
- Any other configuration files

Testing: A Python example with Pytest

Given the simple function

```
def multiply(a, b):
    return a * b
```

We must test normal behavior, edge and extreme cases, as well as incorrect inputs.

Ideally, your function should handle gracefully any errors, or raise an informative exception.

Testing: Python scripts

Pytest usage

Outline

- Linters and formatters
- Testing
- **Automating workflows**
- Continuous Integration
- Additional tools

Revisiting Git workflow

1. Make a copy of the code by
   - If you have write access to the repo. Cloning the repository.
   - If you don’t have write access to the repo. Fork the repository.
2. Create a new branch
3. Develop your new feature
   - Write functions that do one thing and one thing only
   - Write multiple tests for each function. Test normal behavior, edge cases and incorrect inputs, etc.
   - Document thoroughly all the code via docstrings and comments.

Revisiting Git workflow

4. Run the tests. Ensure that all test pass.
    - Ensure enough code coverage. The metric is defined by the organization.
5. Commit your code locally.
    - Write meaningful and detailed commit messages.
6. Push to the branch in the remote repository.
    - This action is permanent.
7. Create a pull request (PR)
    - Wait for colleagues or senior developers to approve your request.

Enjoy!

Revisiting Git workflow

A Git workflow is designed to create a systematic history, enabling you to track bugs and individual features effectively.

This process depends on running the test suite frequently to catch issues early.

To avoid executing tests manually each time, tools like Git Hooks can automate tasks like testing, formatting, and linting during the workflow.

Git hooks

Hooks are scripts that run automatically when specific events occur in Git, such as commits, merges, or pushes.

They work similarly to TensorFlow callbacks, responding to events to automate tasks.

Git hooks are classified into local hooks (running on your machine) and server-side hooks (running on the remote server).

Local Git hooks

Local Hooks are scripts that run on your machine, automating tasks within your workflow.
Some of them are:

pre-commit:Runs before a commit is finalized. Often used to inspect code, run tests, and execute linters to ensure code quality.

Commit-msg: Checks if the commit message follows a specified pattern, guaranteeing consistency in commit messages.

pre-push:Executes before pushing changes to the remote repository. It can verify that all requirements are met, such as passing tests or updating documentation.

Remote Git hooks

Local Hooks are scripts that run on your machine, automating tasks within your workflow.
Some of them are:

Pre-rebase: Runs before a rebase starts. This can warn the user or halt the process if certain conditions aren't met.

post-checkout: Executes after switching branches. It's useful for setting up environment variables or configuration specific to the checked-out branch.

Post-merge: Runs after a successful merge. You can run integration tests or update dependencies here to ensure everything is compatible.

Example #3: Local hooks

In this example, we will run a formatter during the pre-commit stage.

Option A: Writing your hooks manually.

Option B: Using pre-commit framework. More YAML files!

Remote Git hooks

Remote Hooks are executed on the server side (e.g., GitHub Actions, GitLab, or other remote servers). They are enforcers across the entire organization.

pre-receive: Runs before code is accepted by the repository. Commonly used to enforce standards, run tests, execute linters, or send notifications.

update: Similar to pre-receive, but runs on a per-branch basis. Useful for branch-specific policies or checks.

post-receive: Runs after code has been integrated into the repository. Often used to run regression or integration tests, send notifications, or perform maintenance tasks like purging large files.

Example #4:

Setting up a pre-receive hook with GitHub Actions and Rules.

GitHub Webhooks

Webhooks are similar to hooks in that they trigger actions after specific events (e.g., a push).

Unlike hooks, webhooks communicate with external services outside of GitHub.

Some examples include

- Triggering CI pipelines (i.e., Jenkins)
- Deploying code to a production server.
- Sending notifications (Slack, Discord, etc)

These automated actions support Continuous Deployment (CD), enabling new features to be deployed automatically.

Outline

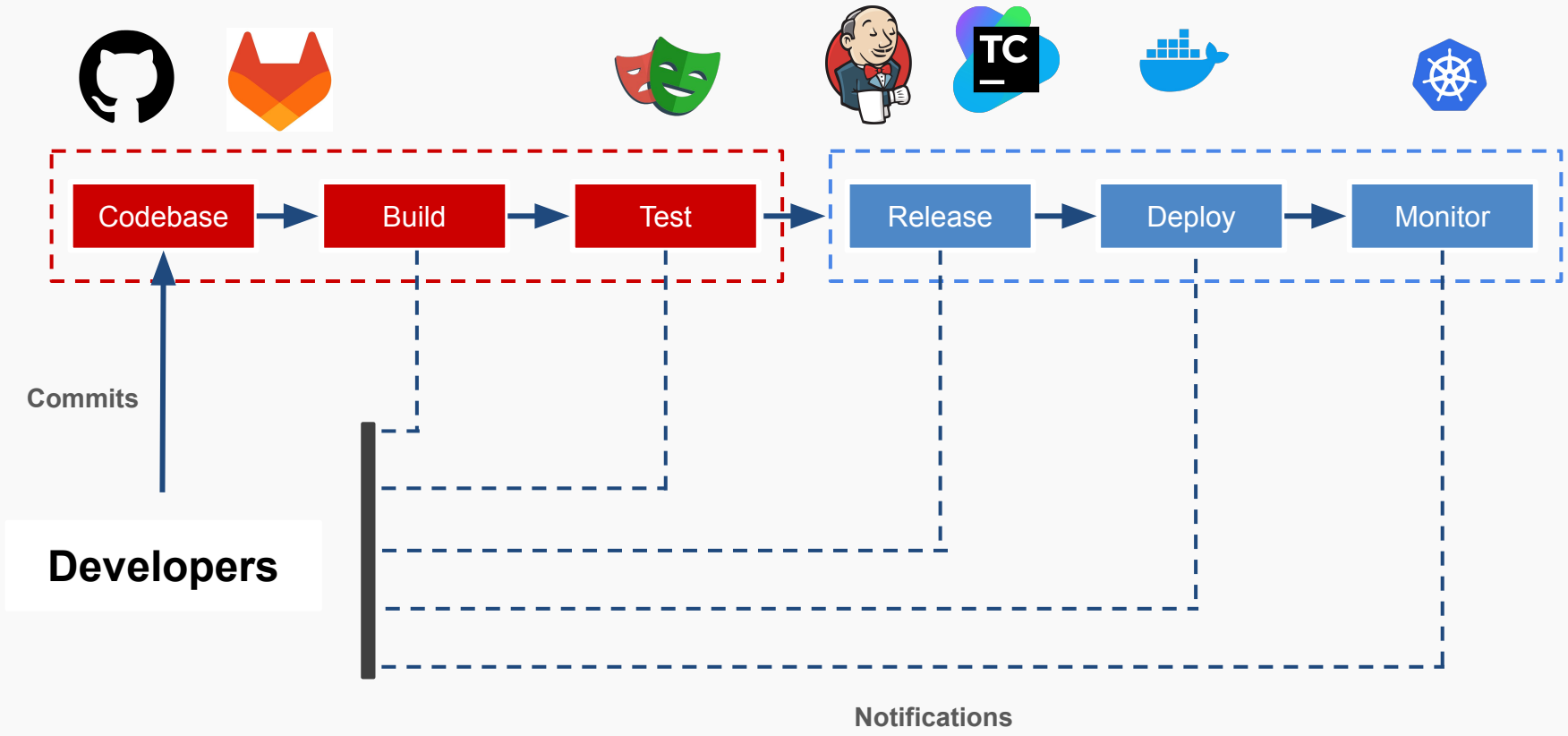
- Linters and formatters
- Testing
- Automating workflows
- **Continuous Integration**

Continuous Integration (CI)

In traditional software development CI implies faster release cycles, improved code quality, and better collaboration

In MLOps: managing the unique challenges of ML model development, testing, and deployment.

Continuous Integration (CI)



Continuous Integration (CI)

Continuous Integration (CI): run a series of scripts automatically, anytime changes are pushed

- Continuously Integrate our changes
- Automated tests
- Ensure coding standards
- Static code analysis

Continuous Integration (CI)

In AI/ML, CI involves regularly merging ML code, data, and models into a shared repository and automatically testing these integrations, ensuring that new features do not break the system..

Quick detection of issues in data, code, or models is paramount. It can cost time and money!

Challenges

Balancing need for frequent integration with the computational demands of training and testing ML models.

It depends on the organization.

CI/CD Providers and Tools

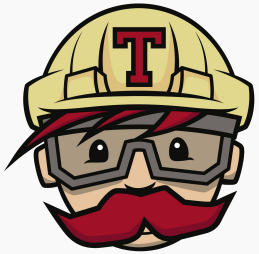
TeamCity

Github

GitLab

Jenkins

CircleCI



TravisCI

just naming a few…

CI example

Running simple_CI