GIT

Agenda

What will we talk about today

Lecture

1. CICD Code testing
2. Environment management
3. Infrastructure as Code (IaC)

Lab

4. lab1-github-basics
5. lab2-model-training-versioning
6. lab3-github-actions-aws-beginner-lab

Wrap-up last week

LIÈGE université

Which type of drift is this?

In predictive maintenance of machinery, a model might predict failures based on sensor readings like temperature, vibration, and pressure.

Drift could occur if the machinery is upgraded or if the materials used in the manufacturing process change, altering the failure modes despite similar sensor readings.

Which type of drift is this?

Suppose a machine learning model predicts credit card fraud based on features like purchase amount, location, and time.

Over time, the introduction of new payment technologies and the shift in consumer spending behavior due to seasonal trends or economic changes can cause significant changes in these features' distributions.

CICD

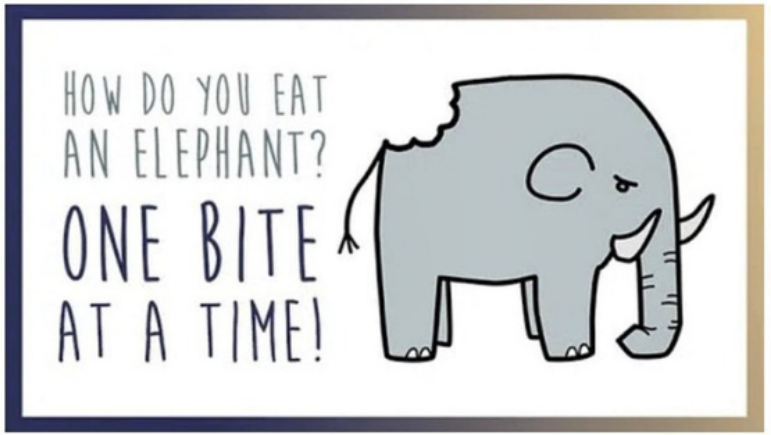
Why do we need CICD

It is better to ship a small change than shipping a GIANT change.

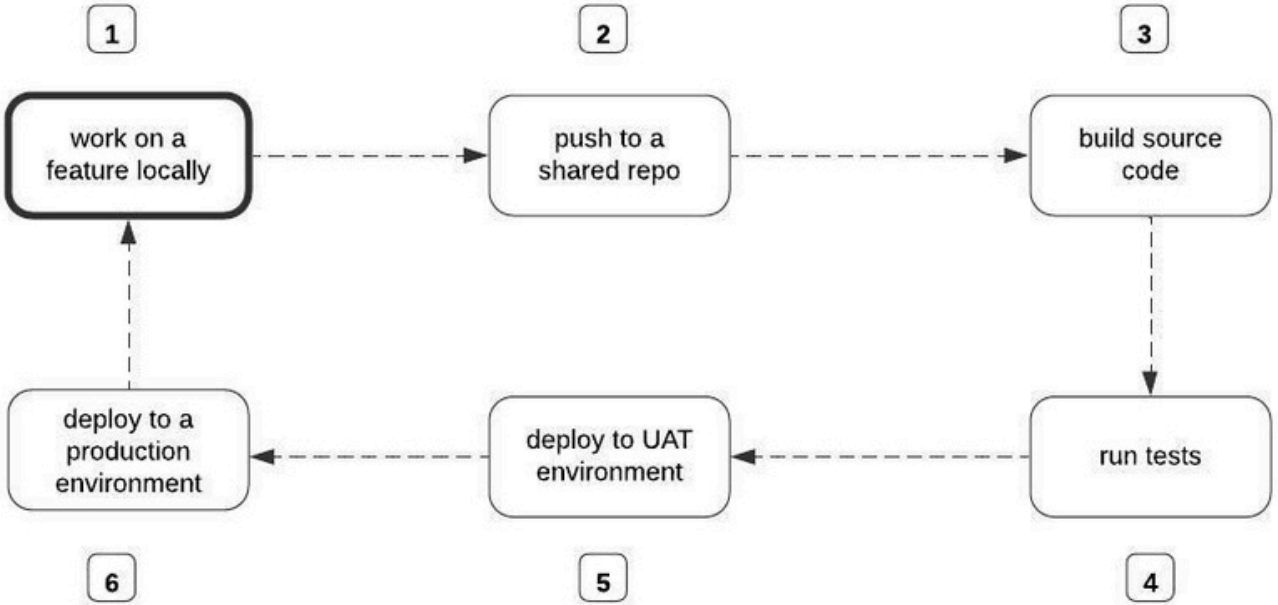
Making frequent changes mean having to *frequently redeploy* your solution ⇒ Automate **deployment**

You want to make sure that the *system works before deploying it* ⇒ Automate **testing**

"How do you eat an elephant? one bite at a time"



Example of a CICD workflow



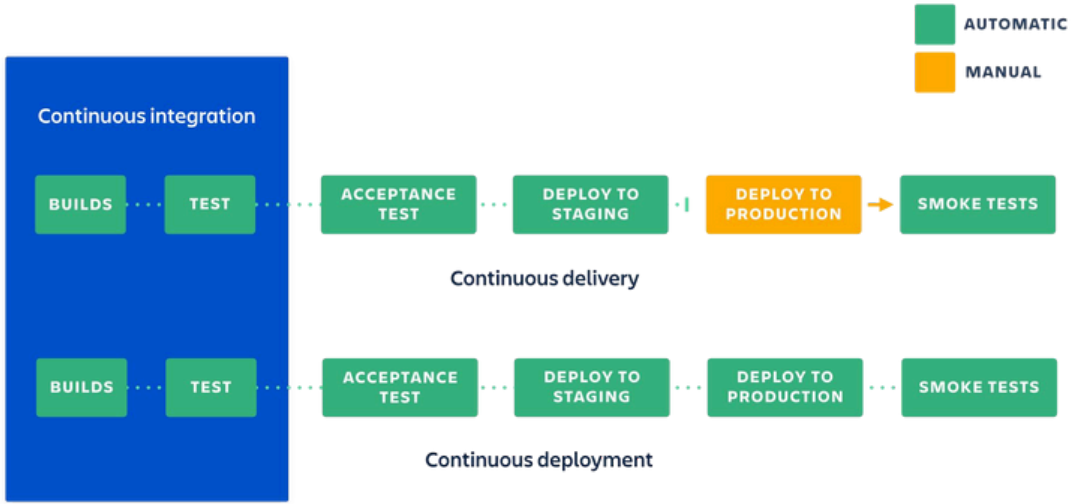
Definition of CICD

Continuous Integration (CI) is a machine learning development practice in which all developers and data scientists merge their code, model definitions, and configuration changes into a central repository multiple times a day. Each integration is automatically verified through automated builds, unit tests, and data validation checks.

Continuous Training (CT) extends CI by automating the model retraining process. Models are automatically retrained when new data becomes available or when model performance degrades, ensuring models stay current and accurate.

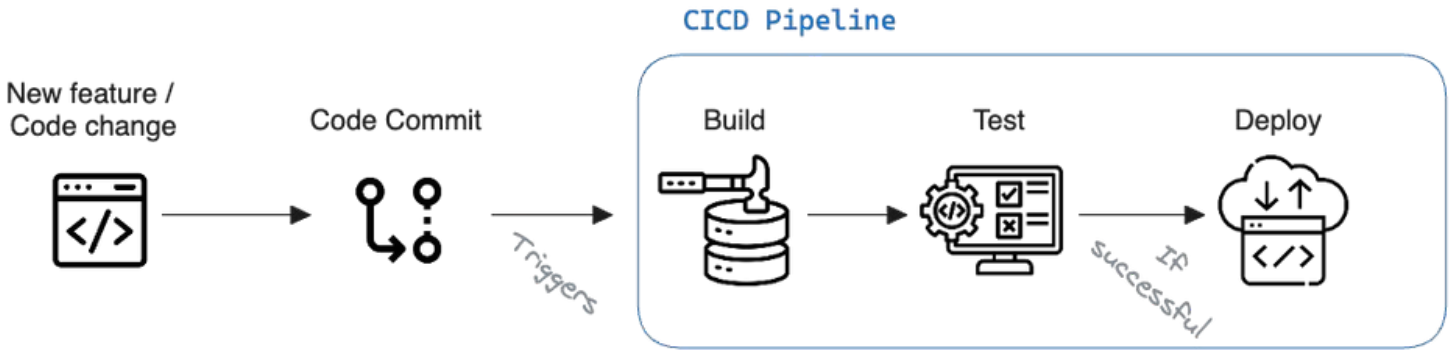
Continuous Delivery (CD) builds on top of CI/CT by automating the entire model release process. Models are automatically tested, validated, and packaged for deployment, but require manual approval before being deployed to production.

Continuous Delivery vs Continuous Deployment



CICD Pipeline

Allowsyoutocontinuously work on your application and efficiently deploy new changes to it.



CICD Pipeline

Integrated in code versioning tools

CICD: Build stage

We combine the source code and its dependencies to build a runnable instance of our product that we can potentially ship to our end users.

Build the **Docker containers** we covered in a previous lecture.

Failure to pass the build stage is an indicator of a fundamental problem in a project’s configuration, and it’s best to address it immediately.

CICD: Test stage

(covered in next section).

CICD: Deploy stage

If our codes passed the previous steps, we can **deploy** the feature we built.

For example, if we updated the logic of an API this part of the pipeline would deploy the API in the Cloud. That could also be a ML model training pipeline.
In development we typically have resources in different **environments**.

Example CICD pipeline with Github Actions

```yaml
name: Build and Test

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

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2
    - name: Set up Python Environment
      uses: actions/setup-python@v2
      with:
        python-version: '3.x'
    - name: Install Dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Run Tests
      run: |
        python manage.py test
```

```yaml
deploy:
    needs: [test]
    runs-on: ubuntu-latest

    steps:
    - name: Checkout source code
      uses: actions/checkout@v2

    - name: Generate deployment package
      run: zip -r deploy.zip . -x '*.git*'

    - name: Deploy to EB
      uses: einaregilsson/beanstalk-deploy@v20
      with:

        // Remember the secrets we embedded? this is how we access them
        aws_access_key: ${{ secrets.AWS_ACCESS_KEY_ID }}
        aws_secret_key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}

        // Replace the values here with your names you submitted in one of
        // The previous sections
        application_name: django-github-actions-aws
        environment_name: django-github-actions-aws

        // The version number could be anything. You can find a dynamic way
        // Of doing this.
        version_label: 12348
        region: "us-east-2"
        deployment_package: deploy.zip
```

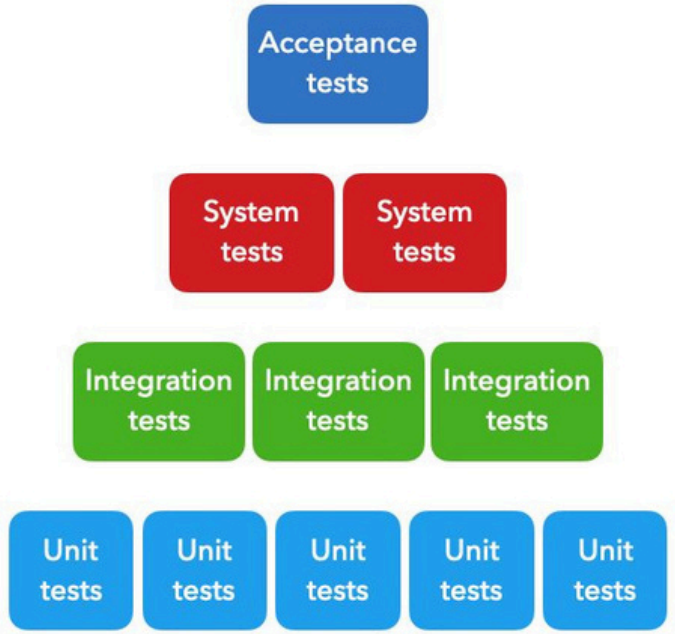
Code testing

Why should we have automatic code tests?

Catchbugs early : Identify problems early in the development cycle, making it easier and cheaper to fix them.
Facilitate Refactoring : Makes it safer to apply changes, as developers know that there is a safety net to deploying changes.
Improve Code Quality: Writing tests forces developers to consider edge cases and error conditions, leading to more robust code.

Documentation: Tests can serve as documentation, showing how a piece of code is intended to be used.

Different types of code testing levels



tests on the **requirements** for the application

Qualitative. *E.g. user testing*

tests on the **design** of a system by validating inputs with outputs

Quantitative. *E.g. training, inference, …*

tests on the **integration** of individual components

E.g. Data processing

tests on individual **components** that have *single responsibilities*

E.g. A single responsibility function

🧠 Testing best practices

Atomic: Single Responsibility Principle (SRP) states that “a module (or function) should be responsible to one, and only one, actor” → Allows for *clear testing*

Compose: Create tests as you implement methods! Catch errors early on and reliably.

Reuse: Reuse similar tests across different projects (can maintain a single repo for tests)

Regression: Test against known errors. If a new error occur → Create a new test to prevent it from happening again in the future.
Coverage: We want to ensure 100% coverage for our codebase. This doesn't mean writing a test for every single line of code but rather accounting for every single line.
Automate: Not only run tests manually but also as part of, for example, your CICD pipelines.

CICD: Test stage

Let’s look at two **tools** to enable two **types** of testing

1. **Functionality** tests with **Pytest**
2. **Code quality** check with **Pylint**

Pytest

Pytest makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries.

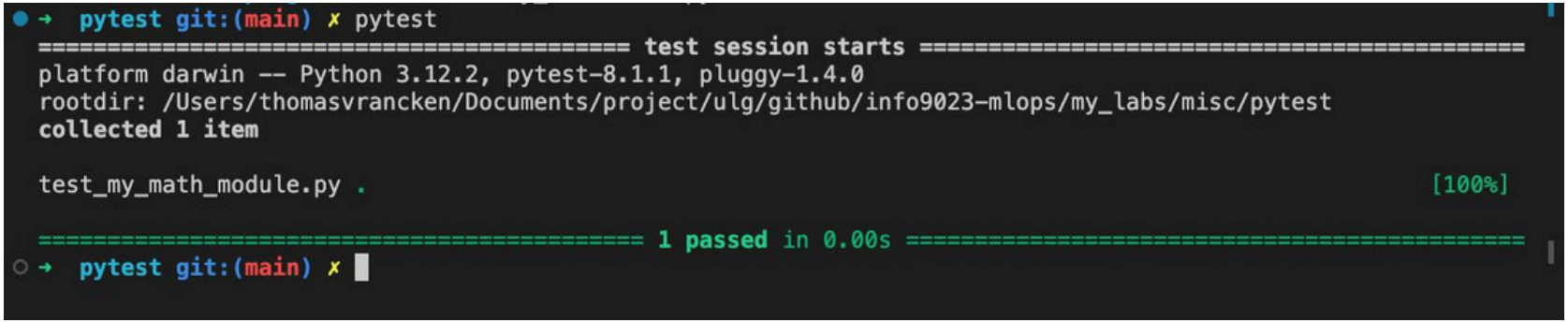
By default, pytest identifies files starting with test_ or ending with _test.py as test files.

```python
# my_math_module.py

def add(a, b):
    """Add two numbers together."""
    return a + b
```

```python
# test_my_math_module.py

from my_math_module import add

def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0
    assert add(-1, -1) == -2
```

Pytest

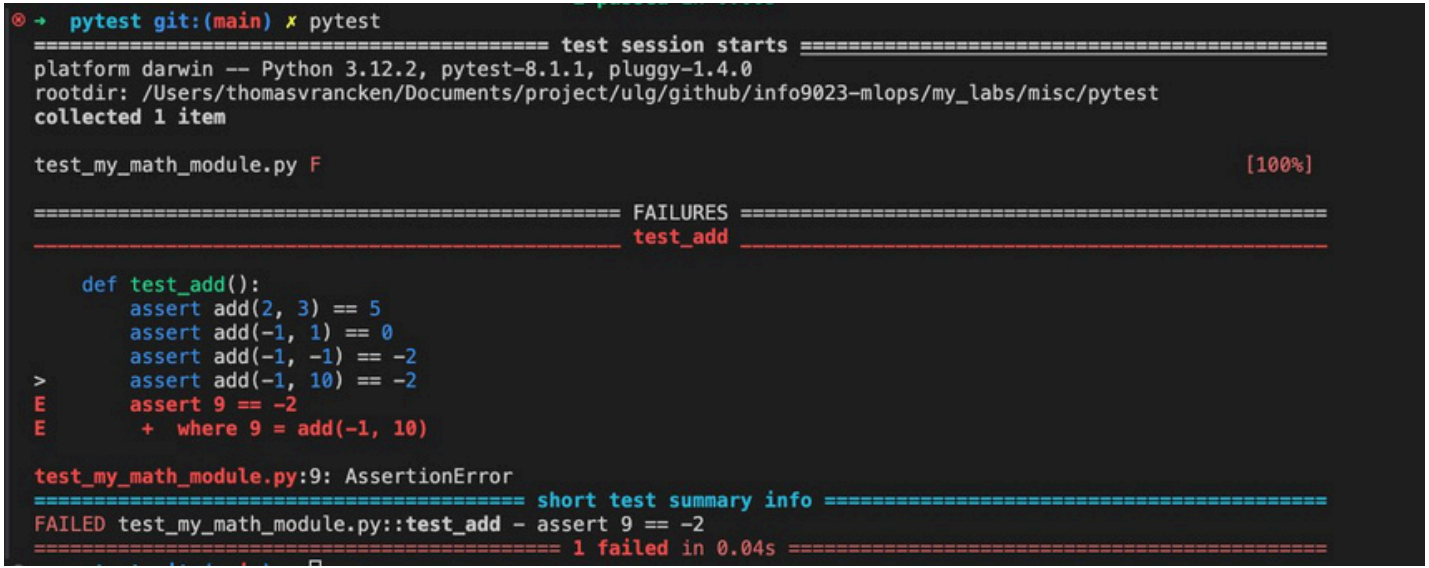
You can then run your tests by just using the `pytest` command in your root directory.



Pytest

```python
# test_my_math_module.py

from my_math_module import add

def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0
    assert add(-1, -1) == -2
    assert add(-1, 10) == -2
```

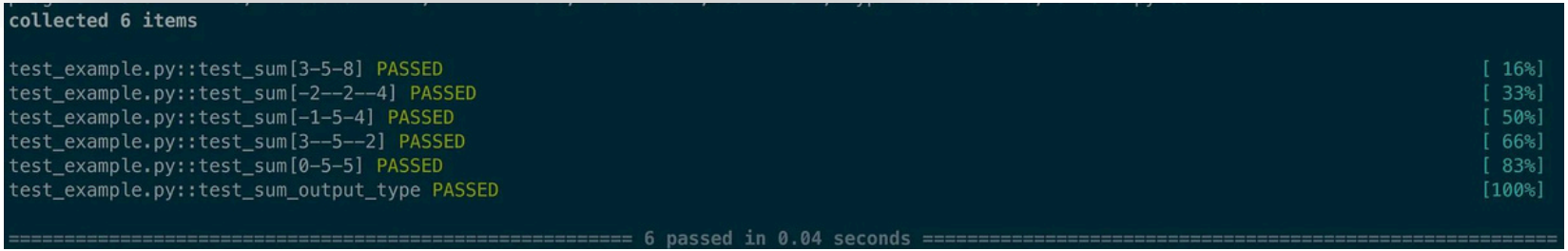
Pytest


pytest git:(main) ✗ pytest
test session starts
platform darwin -- Python 3.12.2, pytest-8.1.1, pluggy-1.4.0
rootdir: /Users/thomasvrancken/Documents/project/ulg/github/info9023-mlops/my_labs/misc/pytest
collected 1 item
test_my_math_module.py F [100%]
FAILURES
test_add
def test_add():
assert add(2, 3) == 5
assert add(-1, 1) == 0
assert add(-1, -1) == -2
> assert add(-1, 10) == -2
E assert 9 == -2
E + where 9 = add(-1, 10)
test_my_math_module.py:9: AssertionError
short test summary info
FAILED test_my_math_module.py::test_add - assert 9 == -2
1 failed in 0.04s

Pytest

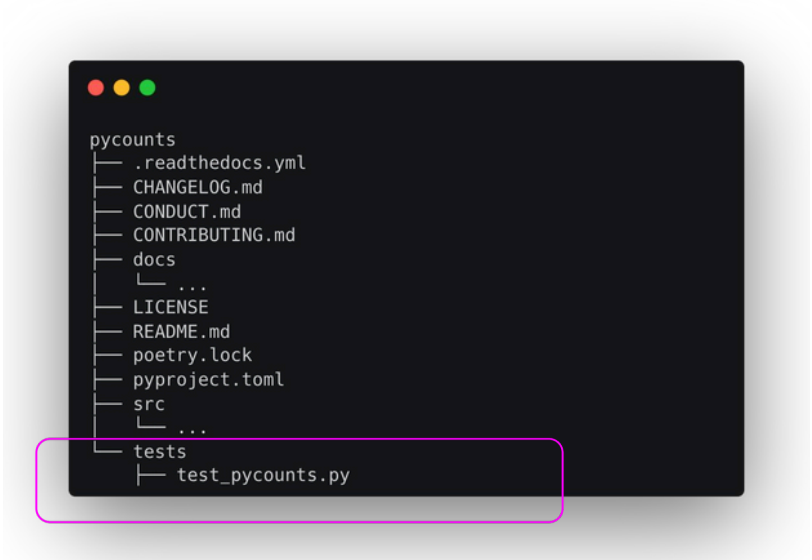
```python
import pytest

# Use the parametrize decorator to test a list of arguments
@pytest.mark.parametrize('num1, num2, expected',[(3,5,8),
(-2,-2,-4), (-1,5,4), (3,-5,-2), (0,5,5)])
def test_sum(num1, num2, expected):
        assert sum(num1, num2) == expected
```

One can use the **parameterized decorator** to run the test on a series of parameters. Define a set of tests to run using methods defined in your codes (e.g. API logic).



Pytest

- Tests are defined as functions prefixed with test_ and contain one or more statements using assert to test against an expected result or raises a particular error
- Tests are put in files of the form `test_*.py or *_test.py`
- Tests are usually placed in a directory called tests/ in a package's root.



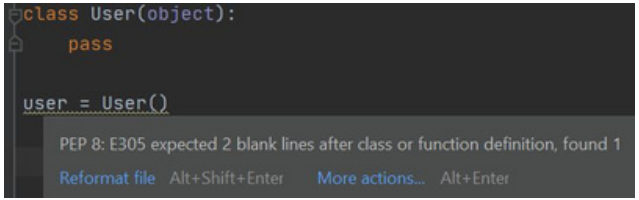
Pylint

Pylint is a **static code analyser** for Python

⇒ Pylint analyses your code without actually running it. It checks for errors, enforces a coding standard, looks for code smells, and can make suggestions about how the code could be refactored.

Code convention: Python PEP8

- Ensures a consistent code quality across a team.
- Set of rules on styling, indenting, naming conventions and documentation
- Integrate/automate in your code editor (IDE) to make it easy. Often enforced PEP8 during pull request submission.
- 🕵️ Developers can be judgy… Make your life easy, adapt clean codes

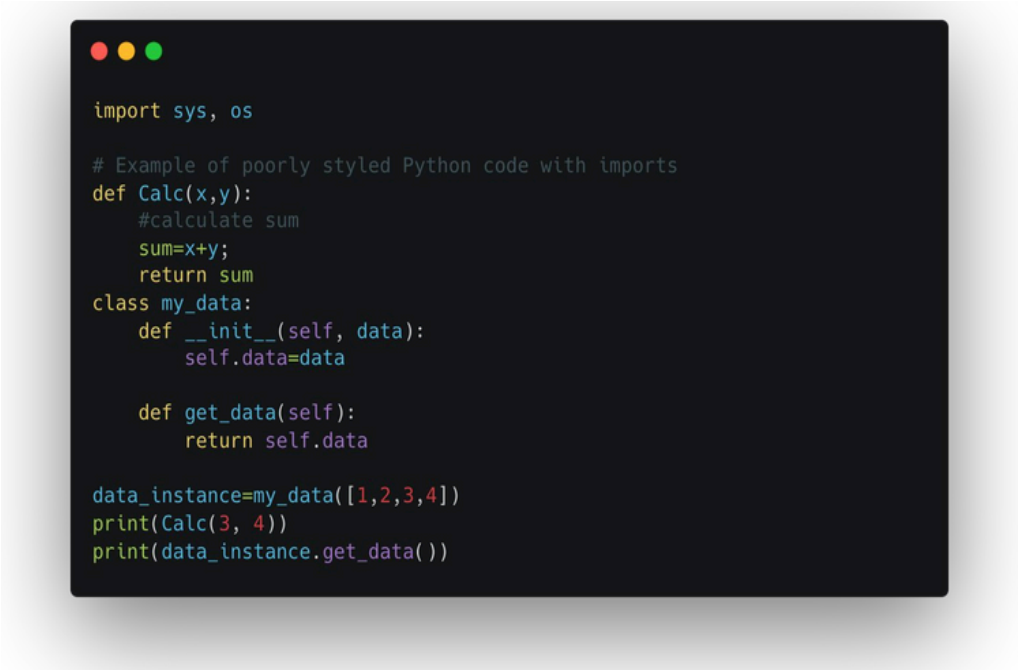


Tools to automatically check compliance with PEP8 will be covered in the CICD lecture

Code convention: Python PEP8

- Code Layout and Indentation
  - Use 4 spacesperindentation level. Avoid mixing tabs and spaces
- Line Length
  - Limit lines to 79 characters, and 72 characters for comments or docstrings
- Blank Lines
  - Surround top-level function and class definitions with two blank lines. Use a single blank line to separate logical sections within a function or method.
- Imports
  - Group imports into three sections: standard library, third-party libraries, and local application/library-specific imports. Each group should be separated by a blank line, and avoid wildcard imports (from module import *).
- Naming Conventions
  - Use snake_case for variable and function names, UPPERCASE for constants, and CamelCase for class names. Avoid single-character variable names, except for counters or iterators.
- Comments and Docstrings
  - Write clear and concise comments that explain why something is done, not how. Use triple double-quotes (""") for module, class, and method docstrings.
- Function and Variable Annotations
  - Use typehints for function arguments and return values (e.g., def add(x: int, y: int) -> int:).
- Exceptions
  - Use specific exception types rather than a generic except: block. Always clean up resources (e.g., files) with context managers like with open().

Pylint



```python
import sys, os

# Example of poorly styled Python code with imports
def Calc(x,y):
    #calculate sum
    sum=x+y;
    return sum
class my_data:
    def __init__(self, data):
        self.data=data

    def get_data(self):
        return self.data

data_instance=my_data([1,2,3,4])
print(Calc(3, 4))
print(data_instance.get_data())
```

Any mistakes?

Pylint

```
⊗ ➜  pylint git:(main) pylint example_script.py
************* Module example_script
example_script.py:6:0: W0301: Unnecessary semicolon (unnecessary-semicolon)
example_script.py:1:0: C0114: Missing module docstring (missing-module-docstring)
example_script.py:1:0: C0410: Multiple imports on one line (sys, os) (multiple-imports)
example_script.py:4:0: C0116: Missing function or method docstring (missing-function-docstring)
example_script.py:4:0: C0103: Function name "Calc" doesn't conform to snake_case naming style (invalid-name)
example_script.py:6:4: W0622: Redefining built-in 'sum' (redefined-builtin)
example_script.py:8:0: C0115: Missing class docstring (missing-class-docstring)
example_script.py:8:0: C0103: Class name "my_data" doesn't conform to PascalCase naming style (invalid-name)
example_script.py:12:4: C0116: Missing function or method docstring (missing-function-docstring)
example_script.py:8:0: R0903: Too few public methods (1/2) (too-few-public-methods)
example_script.py:1:0: W0611: Unused import sys (unused-import)
example_script.py:1:0: W0611: Unused import os (unused-import)

-----------------------------------
Your code has been rated at 0.00/10
```

Best practice: Local pre-commit

pre-commit package.

```
(venv) ➜  madewithml git:(dev) ✗ git add .
(venv) ➜  madewithml git:(dev) ✗ git commit -m "added pre-commit hooks"
trim trailing whitespace.....................................................Passed
fix end of files.............................................................Passed
check for merge conflicts....................................................Passed
check yaml...................................................................Passed
check for added large files..................................................Passed
check yaml...................................................................Passed
clean........................................................................Passed
```

Environment management

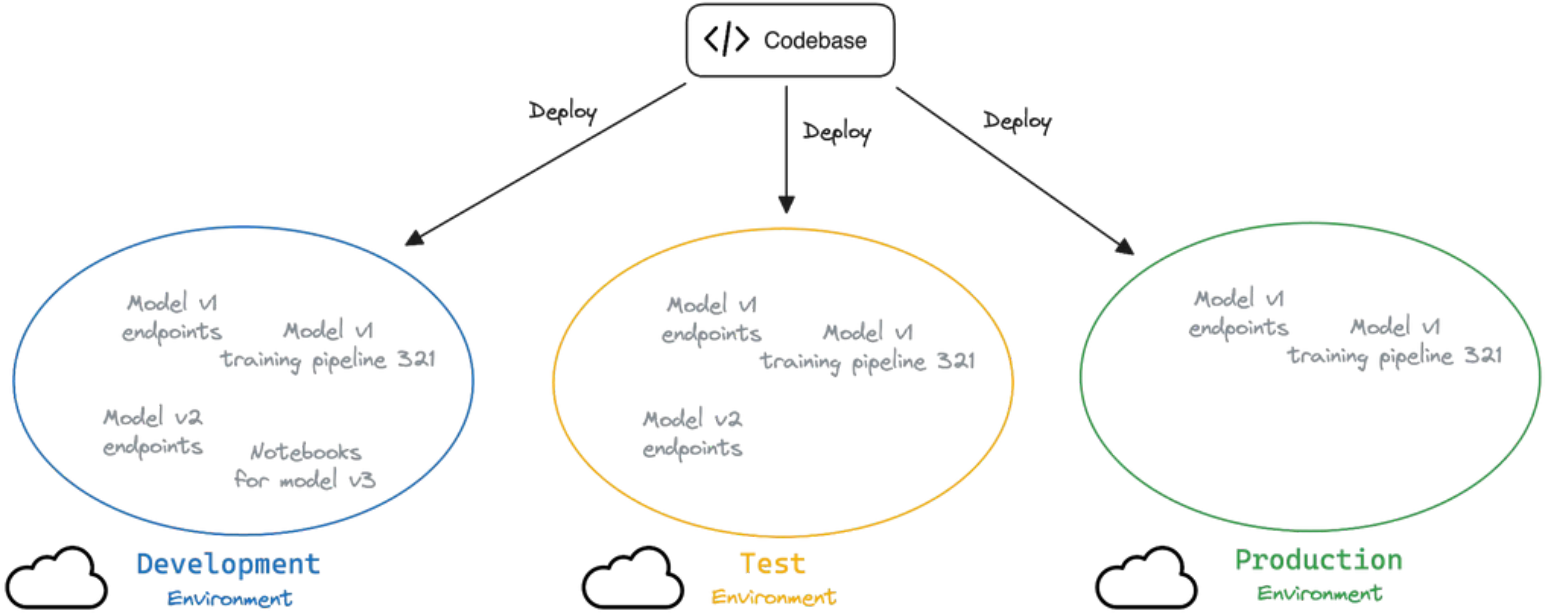
Using multiple environments

In CICD we want to **quickly** and **safely** deliver new features/changes to production. You can deliver small changes frequently, which reduces the risk of problems.

A **multi-environment approach** lets you build, test, and release code with greater speed and frequency to make your deployment as straightforward as possible. You can remove manual overhead and the risk of a manual release, and instead automate development with a multistage process targeting different environments.

Using multiple environments

Example



List of common environments

Environment	**Description**
Development	Where developers can implement new features. Where features can be fully tested once implemented.
Test	Testing can be done with unit testing or manual testing from developers.
Staging	Staging is where you do final testing (load testing, integration tests) immediately prior to deploying to production. Each staging environment should mirror an actual production environment as accurately as
Production	Your production environment (production), sometimes called *live*, is the environment your users directly interact with.

Best practices for multi environments

A few best practices

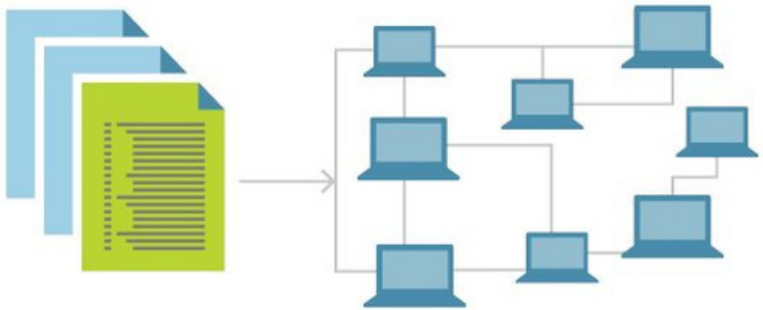
- **Test environments** are important because they allow platform developers to test changes before deploying to production, which reduces risk related to delivery in production.
- Keep your **environments** as **similar** as possible! Helps for reproducibility and finding environment related errors.
- If there are discrepancies in the configuration of your environments, "**configuration drift**" happens, which can result in data loss, slower deployments, and failures.
- Consider adopting methods like **A/B** or **Canary** Deployments that make new features available only to a limited set of test users in production and help reduce the time to release into production.
- **Avoid silos** by allowing all developers to access all environments. (--> Careful with prod)
- You can speed up deployments, improve environment consistency, and reduce "configuration drift" between environments by adopting **Infrastructure as Code** (IaC).

Infrastructure as Code (IaC)

Best practice is to create resources using codes instead of manual steps.

Infrastructure as code (IaC) uses DevOps methodology and versioning to define and deploy infrastructure, such as networks, virtual machines, load balancers, and connection topologies using **codes**.

Just as the same source code always generates the same binary, an IaC model generates the same environment every time it deploys.



IaC benefits

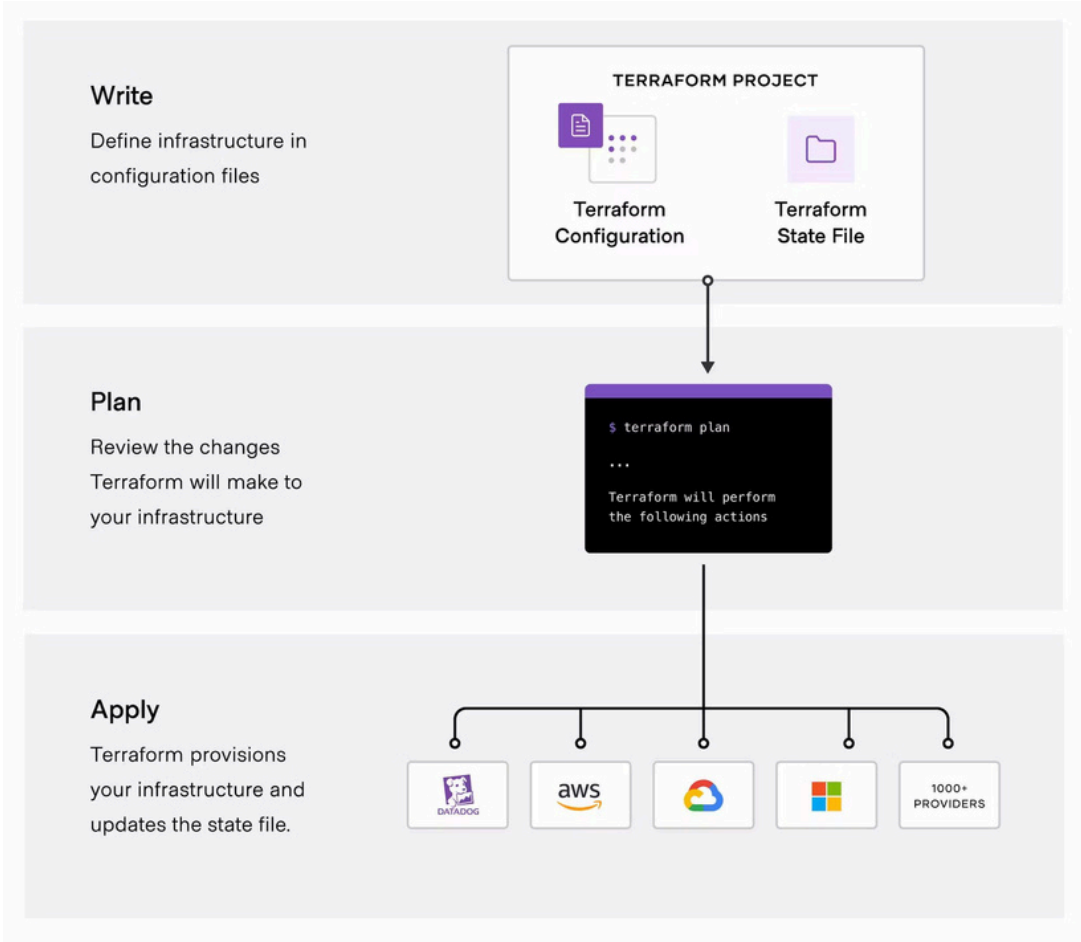
- **Speed** and **efficiency**: Infrastructure as code enables you to *quickly* set up your complete infrastructure *across different environments* by running a script(s).
- **Consistency**: Manual processes result in mistakes and discrepancies… Configurable scripts are much safer.
- **Accountability**: Since you can version IaC configuration files like any source code file, you have full traceability of the changes each configuration suffered.
- **Lower Cost**: Lowering the costs of infrastructure management. Free up developer time from doing repetitive manual tasks

Terraform

Terraform is an **infrastructure as code** tool that lets you define both cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share.

- Terraform can manage low-level components like compute, storage, and networking resources, as well as high-level components like DNS entries and SaaS features.
- Terraform creates and manages resources on cloud platforms and other services through their APIs. Providers enable Terraform to work with virtually any platform or service with an accessible API.

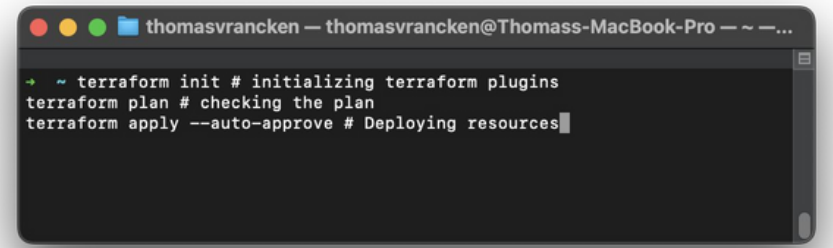
Terraform


Write
Define infrastructure in configuration files
TERRAFORM PROJECT
Terraform Configuration
Terraform State File
Plan
Review the changes Terraform will make to your infrastructure
$ terraform plan
...
Terraform will perform the following actions
Apply
Terraform provisions your infrastructure and updates the state file.
DATADOG
aws
1000+ PROVIDERS

Terraform: Simple example for Cloud Run

```
#main.tf

# Enable the Cloud Run API
resource "google_project_service" "run_api" {
  service = "run.googleapis.com"
  disable_on_destroy = true
}

# Create the Cloud Run service
resource "google_cloud_run_service" "run_service" {
  name = "app"
  location = "us-central1"

  template {
    spec {
      containers {
        image = "us-central1-docker.pkg.dev/someproject-123/docker-repo/fast-api:1.0"
      }
    }
  }

  traffic {
    percent         = 100
    latest_revision = true
  }

  # Waits for the Cloud Run API to be enabled
  depends_on = [google_project_service.run_api]
}
```

```
# output.tf

output "service_url" {
  value = google_cloud_run_service.run_service.status[0].url
}
```



Why Terraform?

Manage any infrastructure:Terraform is versatile and supported by all the main Cloud providers.

Track your infrastructure

- Terraform generates a plan and prompts you for your approval before modifying your infrastructure. It also
- keeps track of your real infrastructure in a state file, which acts as a source of truth for your environment.

Automate changes:

- Terraform configuration files are *declarative* (describe the end state of your infrastructure, not the steps needed to get there)
- Terraform builds a resource graph to determine resource dependencies
  ⇒ creates or modifies non-dependent resources in *parallel*

Standardize configurations: Define reusable *modules* that define configurable collections of infrastructure,

Collaborate: Since your configuration is written in a file, you can commit it to Git and use Terraform Cloud to efficiently manage Terraform workflows across teams

⇒ Share terraform states across a team by *storing them in the Cloud* (e.g. GCS)

Online testing

Online testing

Online testing refers to testing deployed models on live data. By opposition, **offline testing** refers to testing models on a static sample of data.

Youcan receive direct labels from your data (e.g. time series prediction, just wait a bit), or you can use **proxy signals**

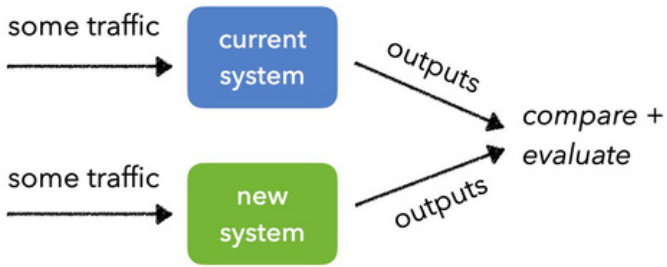
- Manually label data periodically
- Ask users to report wrongly labeled cases and correct the label
- Soft signals such as clicks (e.g. for recommendation engines)

Online testing: A/B testing (1/3)

Deploy both version of your system.

Measure statistical difference of metric/KPI.

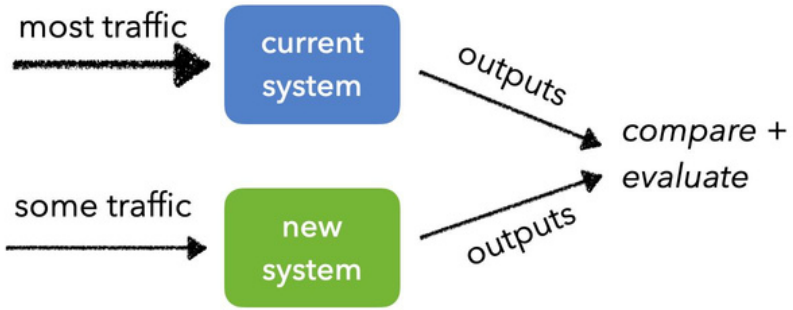
Might need to take into account **novelty effect** (users need time to adapt - think of a new chatbot).
Need enough data to draw a statistical significance.



Online testing: Canary testing (2/3)

Similar to A/B but only redirects parts of the incoming traffic.

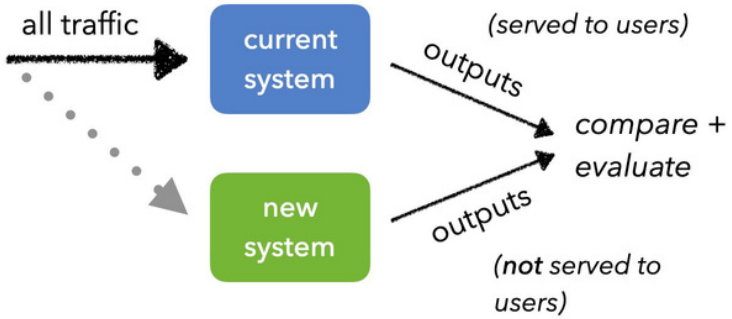
Useful to test out prototypes, to mitigate impact.



Online testing: Shadow testing (3/3)

Send incoming traffic to both systems but this time only send the outputs of the previous system to the users.

Very safe as you are not facing the new system to users.
But does not provide proxy data and feedback from users.



Lab: Github

That’s it for today!