Rockborne

Building and Integrating ML Pipelines

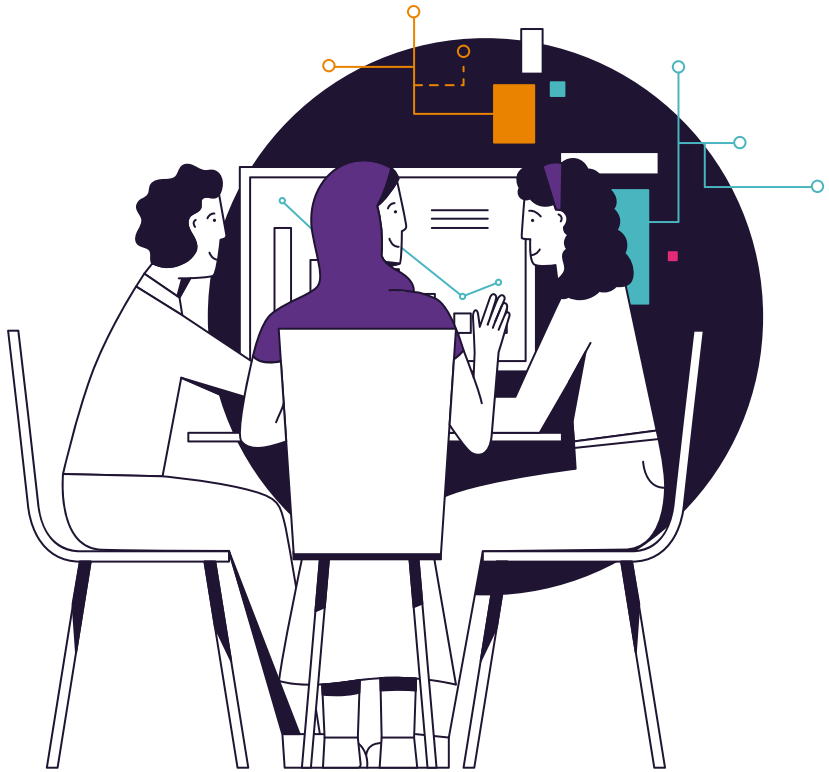
Workshop Agenda

- Testing strategies for Production ML
  - Unit test
  - Integration test
  - Functional and E2E ML Testing
  - Unittest vs PyTest
- Pipeline Orchestration & Automation
  - From notebooks to production pipelines: best practices for refactoring
  - Workflow automation tools (e.g., MLflow)
  - CI/CD for ML
  - Basic Docker usage and best practices for ML projects

Intros

Icebreaker Questions:

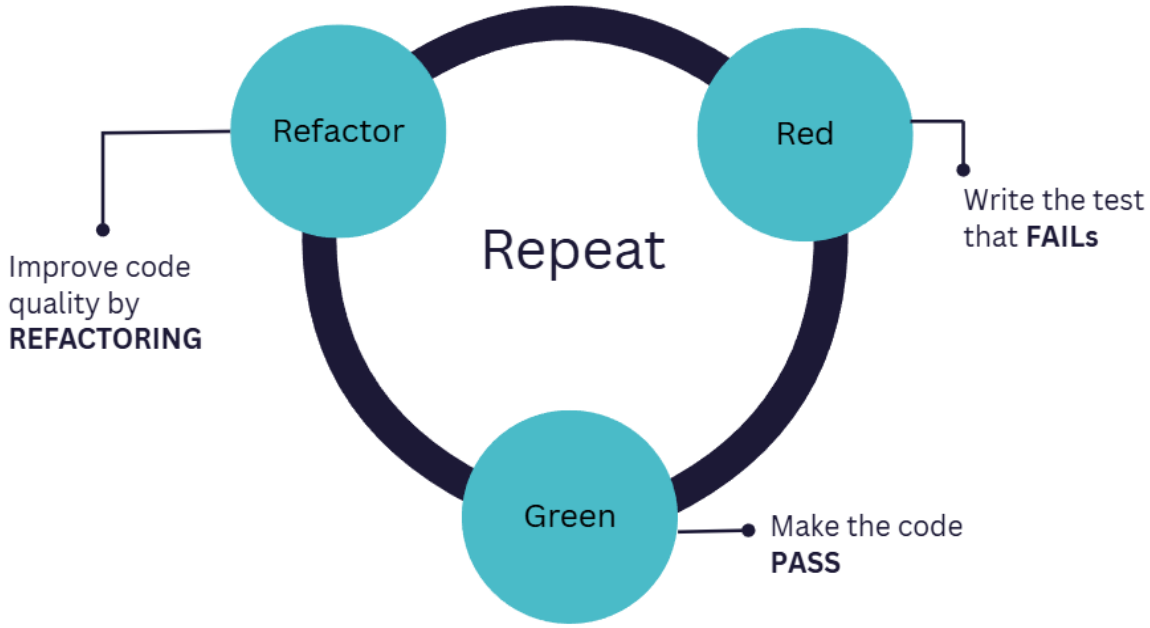
- What is your name?
- What is your current job position?
- What are your expectations for the workshop?
- On a scale of 1-10, how comfortable are you currently with MLOps?



Refactoring

Refactoring refers to the process of restructuring existing code without altering its external behaviour. Typically involves:

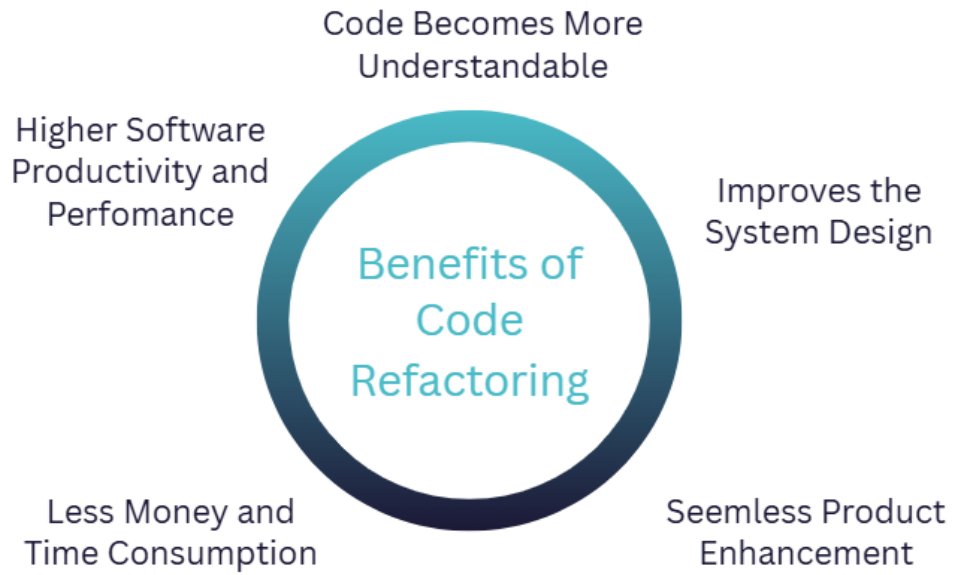
- Organising data loading, transformation, modelling, and configuration into distinct modules.
- Removing redundancy and hard-coded logic.
- Enhancing readability, scalability, and reproducibility.
- Establishing a clean base for collaboration, CI/CD, and model monitoring.



Refactoring

Why refractor?

- Maintainability: Simplifies future edits and debugging
- Reusability: Functions can be shared across scripts, notebooks, or services
- Readability: Clear logic separation improves onboarding and peer reviews
- Scalability: Supports automation, reproducibility, and experimentation at scale.
- Team Collaboration: Facilitates handovers and parallel development



Refactoring

Module	Responsibility
data_loader.py	Reading and basic cleaning of raw data. Initial splitting(e.g., sampling on large datasets).
preprocessing.py	Data transformations and feature engineering.
model.py	Model training, inference, and saving logic.
pipeline.py	Orchestrating pipeline stages in sequence.
config.py	Central definition of parameters and paths.
main.py	Entry point for triggering the pipeline.

Project Structure:

```
ml_project/
├── src/
│   ├── data_loader.py
│   ├── preprocessing.py
│   ├── model.py
│   ├── pipeline.py
│   └── config.py
├── data/
│   └── raw/
├── models/
├── main.py
├── requirements.txt
└── README.md
```

Refactoring – step by step

1. Set Up Version Control: If not already using it, initialise a Git repository.
2. Establish a Testing Baseline: If you don't have tests, start by writing some basic end-to-end tests for your existing, un-refactored code. This confirms what "correct" behaviour looks like.
3. config.py: Extract all hardcoded parameters into config.py. This provides a stable base for the rest of your refactoring.
4. data_loader.py: Focus on clean, modular functions for data ingestion and initial cleaning. Ensure this module outputs consistent data frames.
5. preprocessing.py: Implement transformations as functions or scikit-learn compatible transformers, ensuring correct handling of state and data leakage.
6. model.py: Isolate model instantiation, training, and prediction logic.

Refactoring – step by step

8. pipeline.py: Assemble the refactored components into clear, executable workflows for training and prediction.

9. main.py: This becomes the primary entry point for your application. It will handle argument parsing (e.g., to select train or predict mode), load configuration, and then call the relevant top-level functions from pipeline.py. Keep its logic minimal; it orchestrates, doesn't compute.

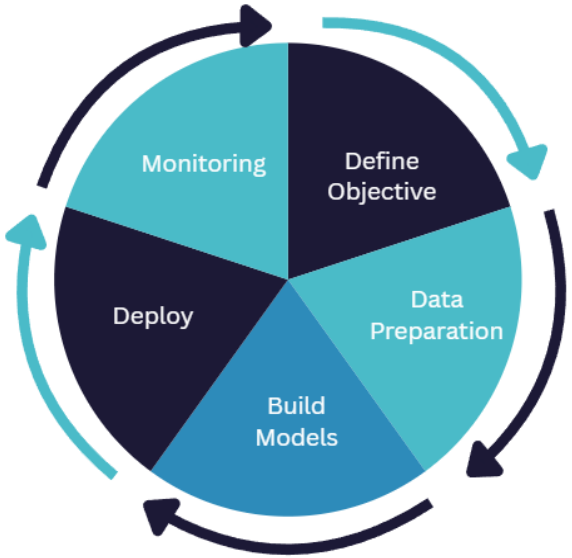
10. Run All Tests: After each significant refactoring step, run your tests to ensure no functionality has been broken.

11. Monitor and Iterate: Refactoring is an ongoing process. As your project evolves, revisit these modules to keep them clean and efficient.

Hands on - Churn Prediction Project

Builds and evaluates a machine learning model to predict customer churn. Follows a standard ML lifecycle:

- **Data Loading:** Ingests raw customer data.
- **Data Cleaning & Preprocessing:** Handles missing values, transformation (e.g., encoding).
- **Feature & Target Separation:** Splits the dataset into features and the target variable .
- **Data Splitting:** Divides data into training and testing sets, ensuring stratification.
- **Model Training:** Trains a classification model (Logistic Regression by default).
- **Model Evaluation:** Calculates and reports key performance metrics.
- **Model Persistence:** Saves the trained model and logs run details.



Hands on - Churn Prediction Project

The dataset:

- **Source:** Commonly found on Kaggle , representing a typical business problem.
- **Context:** Contains customer data from a telecommunications company.
- **Objective:** To predict whether a customer will 'churn' (cancel their service).
- **Features:** A mix of demographic information, service subscriptions, and billing details.
- **Target Variable:** Churn (Binary: Yes/No).
- **Size:** The print statements suggest a dataset with several thousand samples and numerous features.
- **Challenges:** Potential for missing values, encoding needed, class imbalance.

Hands on - Churn Prediction Project

Let’s analyse the code, and following the best practices mentioned before and refractor it to:

Module	Responsibility
data_loader.py	Reading and basic cleaning of raw data. Initial splitting(e.g., sampling on large datasets).
preprocessing.py	Data transformations and feature engineering.
model.py	Model training, inference, and saving logic.
pipeline.py	Orchestrating pipeline stages in sequence.
config.py	Central definition of parameters and paths.
main.py	Entry point for triggering the pipeline.

Project Structure:

```
ml_project/
├── src/
│   ├── data_loader.py
│   ├── preprocessing.py
│   ├── model.py
│   ├── pipeline.py
│   └── config.py
├── data/
│   └── raw/
├── models/
├── main.py
├── requirements.txt
└── README.md
```

Refactoring – additional files

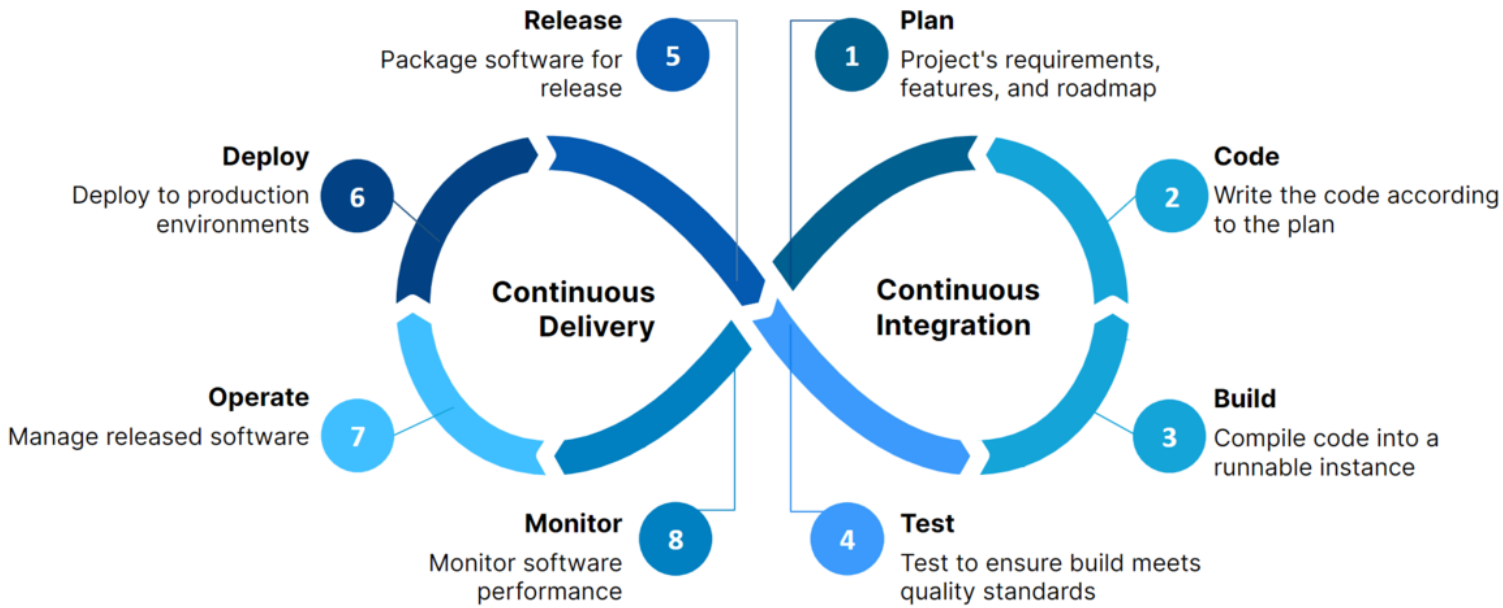
When refactoring code, the following files are essential to include:

Requirements.txt - Think of this file as a blueprint for your project's dependencies. It includes details of libraries (e.g. scikit-learn==1.4.2) which enable your code to run. Using this file guarantees reproducibility and simplifies the environment setup for anyone using the refactored code.

README.md - This document acts as your project's manual, providing crucial information like setup instructions, usage examples, and an overview of the refactored code's purpose and structure, making it understandable and accessible to others.

CI/CD Pipeline

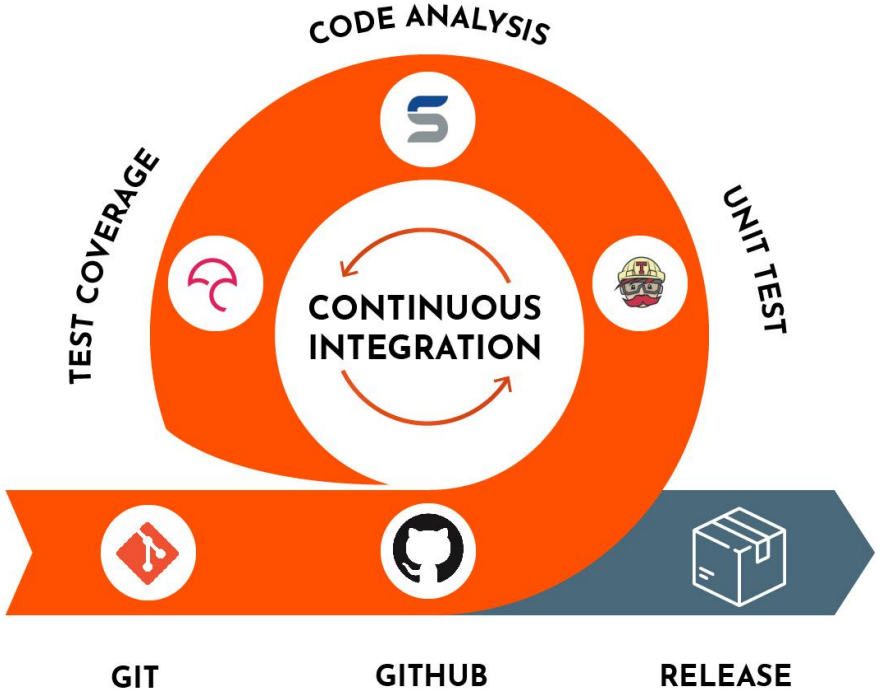
Standing for **Continuous Integration** and **Continuous Delivery/Deployment**, is a set of practices originating from traditional software development. It's all about automating and streamlining the process of getting changes from development to production reliably and efficiently.



CI/CD Pipeline – In ML

In Machine Learning:

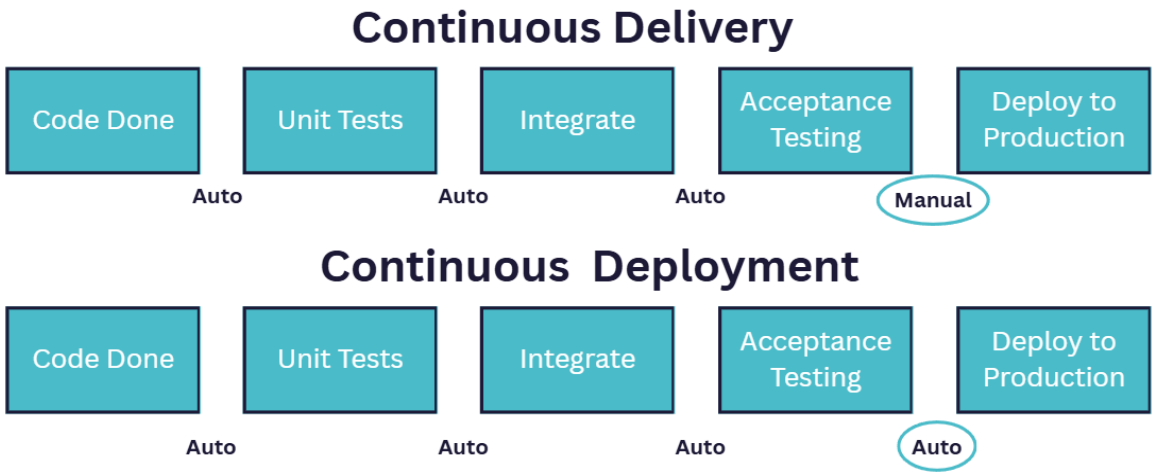
- **Continuous Integration (CI):** whenever a data scientist or ML engineer makes a change to the code, perhaps a new feature engineering step, a different model architecture, or even just fixing a bug, those changes are automatically integrated and tested. This includes checks on the code itself, but also on the data pipelines and the basic functionality of the model training process.



CI/CD Pipeline – In ML

In Machine Learning:

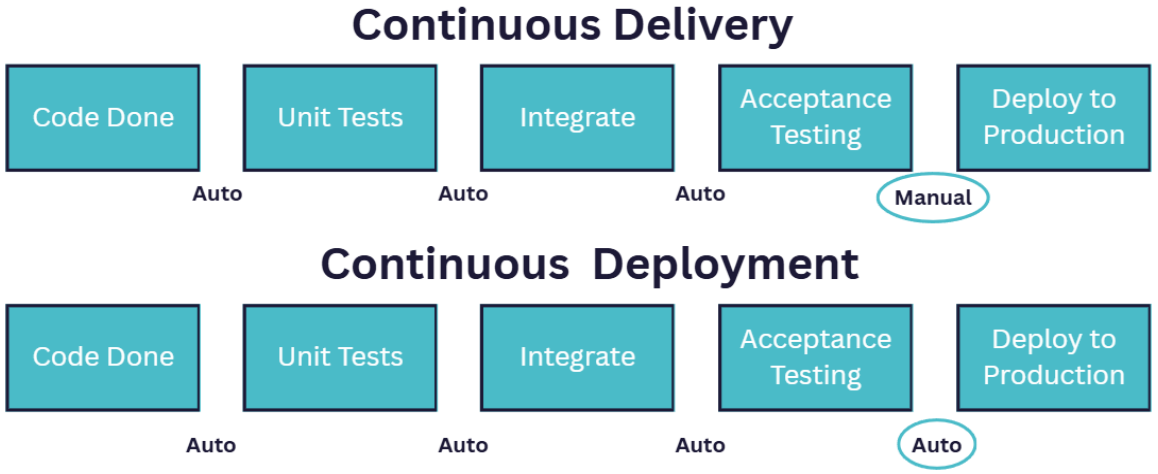
- **Continuous Delivery (CD):** once your updated model code and data pipeline have passed all the automated tests in CI, the system is ready to package and deploy that new model to a staging or production environment. It's in a "deployable" state but might still require a manual trigger.



CI/CD Pipeline – In ML

In Machine Learning:

- **Continuous Deployment (CD - the second 'D'):** this implies that a new model, having passed all its automated tests, including rigorous performance evaluations and data validation, is automatically deployed into the live production environment, serving predictions to users. This is particularly relevant for models that need frequent updates due to changing data patterns.



CI/CD Pipeline - Benefits

- **Faster Iteration and Deployment:** CI/CD automates model updates, enabling quick responses to evolving business needs.
- **Improved Reliability and Consistency:** Automating deployments reduces human error, ensuring models are consistently robust in production.
- **Enhanced Reproducibility:** CI/CD, with version control, guarantees that models can be reliably rebuilt and re-trained for auditing and debugging.
- **Better Collaboration:** It streamlines team efforts by automatically integrating contributions, ensuring everyone uses validated components.
- **Reduced Risk:** Deploying smaller, tested changes more often significantly lowers deployment risks and simplifies issue resolution.

CI/CD Pipeline - GitHub

GitHub Actions is GitHub's built-in CI/CD platform that allows you to automate tasks directly within your repositories. Key elements:

- **Workflows:** At the heart of GitHub Actions are Workflows. These are automated processes that you set up to run specific jobs, such as training a machine learning model, running tests, or deploying a new version of your model.
- A workflow is typically triggered by events, like pushing new code to your repository, creating a pull request, or on a scheduled basis.

CI/CD Pipeline - GitHub

- **YAML Files:** These workflows are defined using YAML files (specifically, .yml or .yaml files). You create these files within a .github/workflows directory in your project's repository.
- These YAML files describe the sequence of steps, or "jobs," that your workflow will execute. For instance, one job might be dedicated to data preprocessing and model training, while another handles model evaluation and deployment.

In summary, you define your automated processes in YAML files using various Actions as building blocks, and then you monitor and manage the execution of these Workflows directly from the "Actions" tab in your GitHub repository.

```yaml
name: Churn Classification APP CI/CD (Unittest)

on:
  push:
    branches: [main]
    paths:
      - '02 Building & Integrating ML Pipelines/churn_classification_app_uni
  pull_request:
    branches: [main]
    paths:
      - '02 Building & Integrating ML Pipelines/churn_classification_app_uni

jobs:
  test:
    name: Run Unit, Integration, Functional, and E2E Tests
    runs-on: ubuntu-latest

    defaults:
      run:
        # Set the working directory to the project root within the repositor
        working-directory: "02 Building & Integrating ML Pipelines/churn_cla

    steps:
    - name: Checkout Repository
      uses: actions/checkout@v3

    - name: Set up Python 3.11
      uses: actions/setup-python@v4
      with:
        python-version: '3.11'

    - name: Install Dependencies
      run: |
        pip install -r requirements.txt
```

Testing

Here are the core reasons why testing your code is essential:

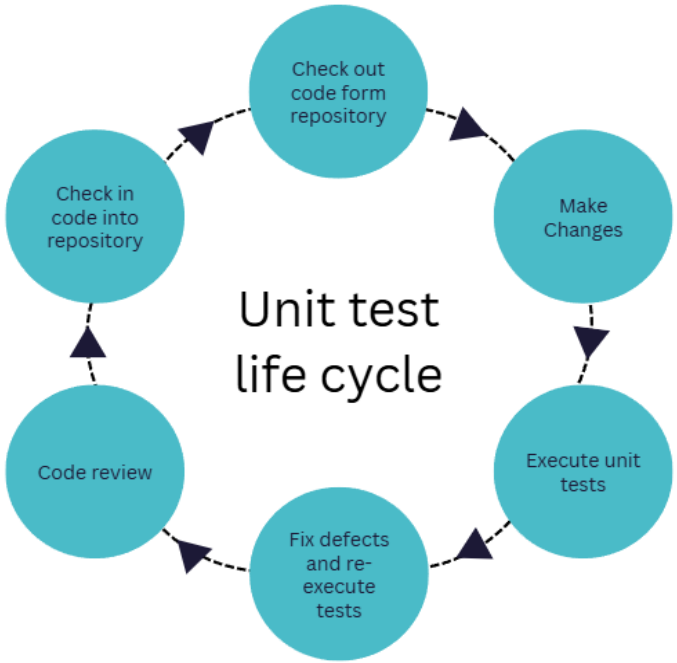
- **Ensures Correctness & Reliability:** Testing catches bugs early and verifies your code functions as intended, preventing silent failures, especially critical in machine learning.
- **Facilitates Refactoring & Maintenance:** Tests act as a safety net for changes, giving you the confidence to refactor and modify code without introducing new errors.
- **Improves Code Quality & Design:** Writing tests naturally encourages modular and testable code, leading to cleaner design and acting as living documentation.
- **Boosts Confidence & Saves Time:** Passing tests build developer confidence and, by finding issues early, save significant time and resources in the long run.

Testing Unit

Unit testing is a fundamental software testing technique focused on verifying the correctness of individual, isolated components (or "units") of code. These units are usually the smallest logical elements, such as individual functions or methods, tested independently from external dependencies.

Why it matters in MLOps:

- Ensures reliability of feature engineering functions, data validation scripts, and custom ML utilities.
- Prevents early bugs from leaking into training or production pipelines.



Testing Unit

Benefits of Unit Testing:

Isolation & Granularity
Tests small code blocks independently for precise bug tracking and faster runs

Improved Design
Encourages modular, testable, and reusable components

Early Bug Detection
Catches logic errors before integration, reducing downstream failures

Regression Prevention
Ensures new code doesn't break existing logic

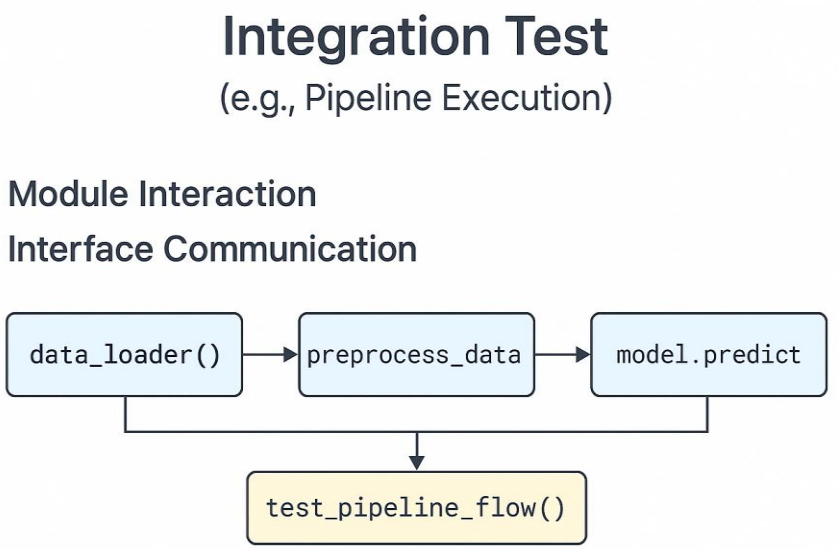
Automation & Speed
Enables fast, frequent, automated checks during development

Testing Integration

Integration testing ensures that multiple components or modules of an ML system function correctly when combined. It validates how they interact, share data, and pass control between each other — going beyond what unit tests can detect.

Common Targets in MLOps:

- Data ingestion modules connecting to transformation scripts
- Model prediction pipelines interfacing with APIs
- Workflow orchestration between training, evaluation, and logging components



Testing Integration

Key aspects of integration testing:

Interaction Verification: Ensures multiple components share and process data correctly

Interface Validation Detects data mismatches or errors in input/output structure

Early Bug Detection Catches integration errors not visible in isolated unit tests

Simulates Real-World Flows Tests end-to-end sequences across pipeline stages

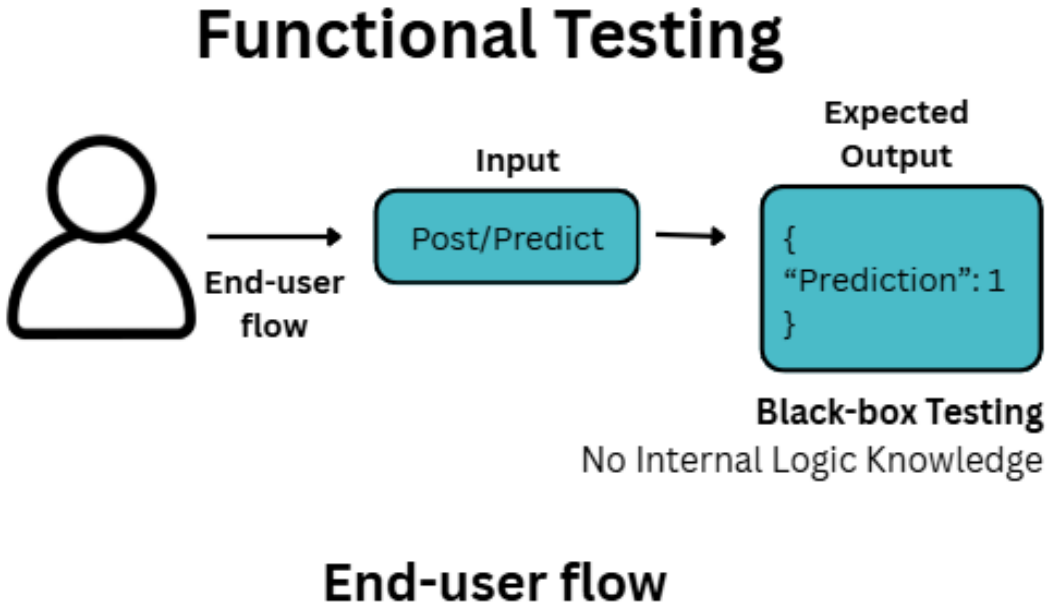
Uses Real Dependencies Includes components like databases, APIs, and services

Testing Functional

Functional testing verifies that software behaves as expected based on user requirements — focusing on **what the system should do**, not how it does it. It's a black-box technique that checks if full workflows (e.g., data input to prediction output) work from an **end-user perspective**.

In MLOps Context:

- Validates that data pipelines, API endpoints, and model outputs work together as specified
- Emulates how users or systems will interact with the ML system



Testing Functional

Key aspects of functional testing:

User Perspective
Tests without needing to know how the system is implemented.

Requirements Validation
Ensures system meets business specifications.

Black-Box Approach
Tests based on input/output behavior.

Feature Completeness
Confirms all features work as intended.

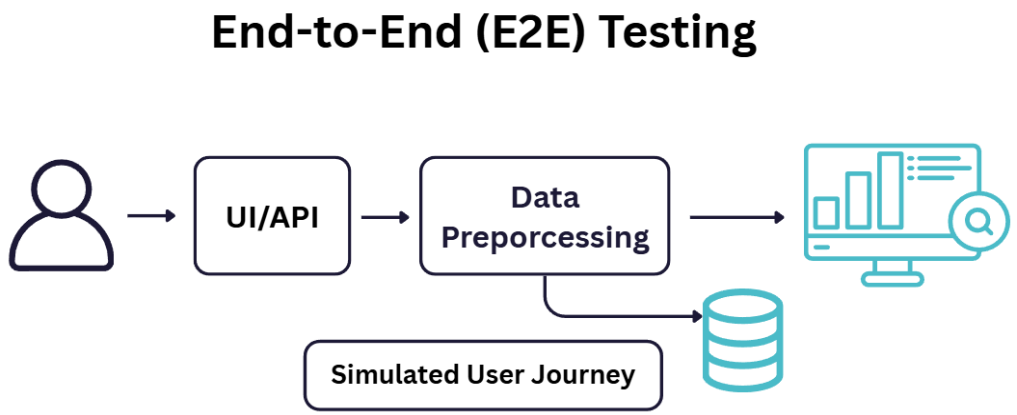
End-to-End Flow Coverage
Covers workflows like model scoring to logging to dashboard display.

Testing E2E

E2E testing simulates a **real user's full workflow** through the entire ML system — from data input to output presentation. It validates that all components (frontend, backend, models, databases, APIs) work **together correctly** across the full stack.

In MLOps Context:

- Simulates real-world use cases: e.g., uploading data → triggering training → viewing prediction/report
- Involves live services: databases, dashboards, cloud storage, etc.



Testing E2E

Key aspects of E2E testing:

Real User Simulation
Executes entire workflows across all layers.

System-Wide Validation
Checks the end-to-end behavior, including infrastructure and external services.

Highest Confidence
Mimics production to catch what other tests miss.

Exposes Integration Issues
Captures hidden defects from system interactions

Comprehensive Coverage
Tests from UI to Logic to DB to Response

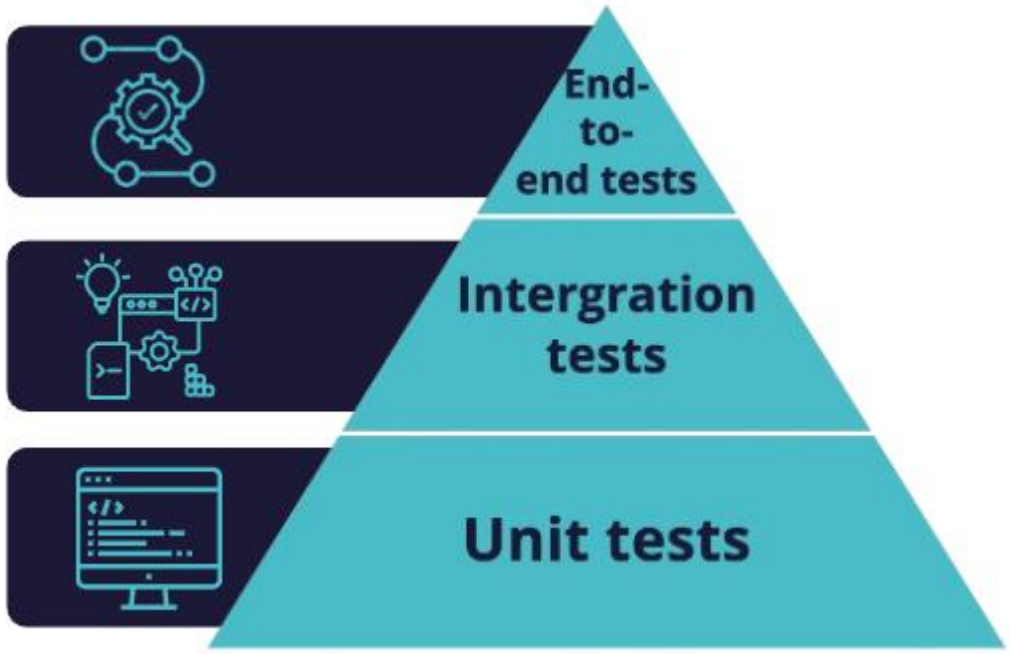
Testing Pyramid

Why Use the Testing Pyramid?

- **Efficiency**
- **Cost-Effectiveness**
- **Maintainability**
- **Reliability**

The testing pyramid is a framework that visualises tests as categories, suggesting how many of each type should be used.

The idea is to have many fast, isolated tests at the bottom, and fewer, slower, more integrated tests at the top.



Testing Recap

Feature	Unit Testing	Integration Testing	Functional Testing	End-to-End (E2E) Testing
What it Tests	Smallest isolated code units (functions, methods).	Interactions and interfaces between combined modules.	Specific features/functions against requirements.	Entire application, including external systems, simulating user flow.
Focus	Correctness of individual components.	Data flow and communication between modules.	"What" the system does for the user.	Full user journey and overall system health.
Speed	Very Fast.	Fast to Medium.	Medium.	Slowest.
When to Use	Early development, during coding, refactoring.	After unit testing, combining modules.	After integration, verifying features.	Late in development, pre-release, critical flows.
Purpose	Catch bugs early, ensure component health.	Verify module collaboration, expose interface issues.	Confirm requirements are met, features work.	Validate complete system readiness and user experience.
Dependencies	Mocks/stubs used to isolate.	Often uses real or mocked dependencies.	Uses real dependencies/environment.	Uses real dependencies and a production-like environment.

Testing - Profiles

Who does what?

- **Data Scientists**: Profile unit-level code, feature transformations, and model training efficiency.
- **ML Engineers**: Focus on training time, batch processing, and hardware utilisation.
- **MLOps Engineers**: Handle integration, deployment, and API latency profiling.
- **DevOps/SREs**: Monitor system-level performance in production (CPU, memory, I/O).

Designing Machine Learning Systems – Chip Huyen, MLOps Guide - Google

Testing Libraries - Unittest

'Unittest' is a built-in Python **testing framework** used to test code logic. It helps ensure your functions, models, and data pipelines behave as expected.

How to Use 'unittest'

1. Import the module: `import unittest` and define
2. Create a test class inheriting from '`unittest.TestCase`'
3. Define test methods that begin with '`test_`'
4. Use assertion methods to validate outputs
5. Run tests using:
    - CLI: `python '-m unittest test_script.py'`
    - Programmatically: '`unittest.main()`'

Example Script

```python
import unittest  # 1. Import

def add(x, y):    # 2. Code to test
    return x + y

class TestMath(unittest.TestCase):  # 3. Test class
    def test_add(self):             # 4. Test method
        self.assertEqual(add(2, 3), 5)  # 5. Assertions

if __name__ == '__main__':          # 6. Test runner
    unittest.main()
```

Testing Libraries – Unittest Cheatsheet

Assertion Method	What It Does	Example
assertEqual(a, b)	Check if a == b	self.assertEqual(2 + 2, 4)
assertNotEqual(a, b)	Check if a != b	self.assertNotEqual(2 * 2, 5)
assertTrue(x)	Check if x is True	self.assertTrue(3 < 5)
assertFalse(x)	Check if x is False	self.assertFalse(5 < 3)
assertIs(a, b)	Check if a is b (same object)	self.assertIs(obj1, obj1)
assertIsNot(a, b)	Check if a is not b	self.assertIsNot(obj1, obj2)
assertIsNone(x)	Check if x is None	self.assertIsNone(result)
assertIsNotNone(x)	Check if x is not None	self.assertIsNotNone(data)
assertIn(a, b)	Check if a in b	self.assertIn(3, [1, 2, 3])
assertNotIn(a, b)	Check if a not in b	self.assertNotIn(4, [1, 2, 3])
assertRaises(Exception)	Check that an exception is raised	with self.assertRaises(ValueError): int('x')

Testing – Hands-On

- Unit
- Integration
- Functional
- E2E
- CI/CD pipeline

CI/CD Pipeline – GitHub - Hands On

On this section we will:

- Analyse the solution that we are planning to deploy.
- Configure our project to define the MLOps pipeline on GitHub
- Develop the YAML file for our project
- Push the changes to the repository
- Monitor the process on GitHub
- Evaluate the results

Testing Libraries – Unittest vs PyTest

Pytest is a **popular third-party** testing framework for Python. It is considered more **concise, readable**, and **feature-rich** than unittest, supporting **parametrised tests**, **fixtures**, **plugins**, and **detailed output**.

PyTest: Testing Palindrome

```python
def is_palindrome(word):
    word = word.lower().replace(" ", "")
    return word == word[::-1]

def test_simple_palindrome():
    assert is_palindrome("madam")

def test_not_palindrome():
    assert not is_palindrome("hello")
```

Unittest: Testing Palindrome

```python
import unittest

def is_palindrome(word):
    word = word.lower().replace(" ", "")
    return word == word[::-1]

class TestPalindrome(unittest.TestCase):
    def test_simple_palindrome(self):
        self.assertTrue(is_palindrome("madam"))

    def test_not_palindrome(self):
        self.assertFalse(is_palindrome("hello"))

if __name__ == '__main__':
    unittest.main()
```

Testing Libraries – Unittest vs PyTest

Aspect	unittest	pytest
Framework Type	Built-in (no install required)	External (install with pip install pytest)
Test Structure	Class-based (class Test...)	Function-based (no class required)
Assertions	Verbose: self.assertTrue(), self.assertFalse()	Clean: assert ... / assert not ...
Boilerplate	More (import, class, main block)	Minimal (just write test functions)
Readability	Verbose, more structured	Cleaner and more Pythonic
Setup/Teardown	setUp() / tearDown()	@pytest.fixture
Output & Debug Info	Basic error output	Rich tracebacks, detailed failure reporting
Test Discovery	Manual or by naming convention	Automatic by naming (test_*.py, test_*)
Advanced Features	Limited	Parametrisation, fixtures, plugins, markers

Docker

Think of Docker like a **shipping container for your code**. Docker containers package your ML code, dependencies, and environment so they run identically everywhere, your laptop, cloud servers, or production systems.

Container: A lightweight, isolated environment running your application. Like having a mini-computer dedicated just to your ML model.

Image: The blueprint or recipe for creating containers. Think of it as a snapshot of everything needed to run your code - like a frozen meal with all ingredients included.

Dockerfile: The instruction manual that tells Docker how to build your image. It's like a recipe that says, "start with Python 3.11, add these libraries, copy this code, set these configurations."

Docker – Hands-On

We are going to deploy a Churn Prediction APP. For this we are going to:

- Initial testing on the APP
- Create the Dockerfile
- Generate an image from the Dockerfile
- Create a container and test the APP locally

Docker – Hands-On

- Docker
- Testing process on an App (unit, integration, functional, E2E)
- CI/CD

MLFlow

- An open-source platform, purpose-built to assist machine learning practitioners and teams in handling the complexities of the machine learning process. https://mlflow.org/docs/latest/ml/
- Focuses on the full lifecycle for machine learning projects, ensuring that each phase is manageable, traceable, and reproducible.
- Benefits:
  - Reproducibility: Easily recreate past experiments and models.
  - Model Lifecycle Management: From packaging, versioning, and deploying models from development to production.
  - Library Agnostic: Works with any ML library and programming language (Python, R, Java).

MLFlow – Hands-On

- MLFlow installation, configuration and walkthrough.

  Set experiments to test and deploy MLFlow.

  - Simple ML solution
  - Data preprocessing
  - Multiple models comparison
  - Using trained models on new data
  - Deploying trained model with FastAPI

Rockborne

Deployment & Productionising ML Models