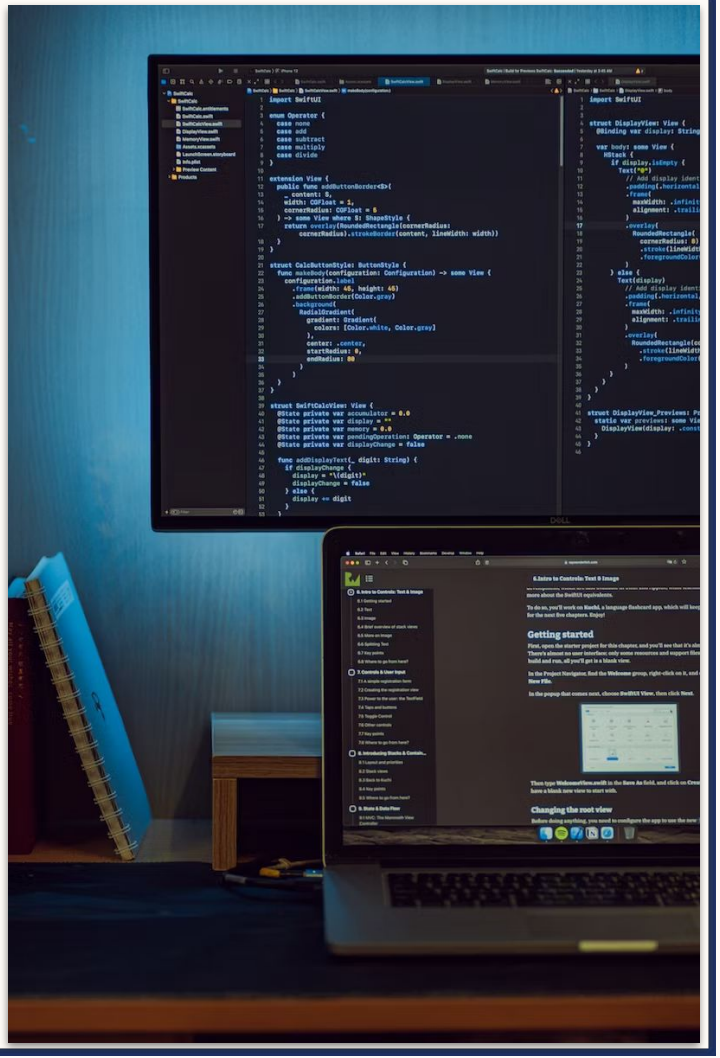
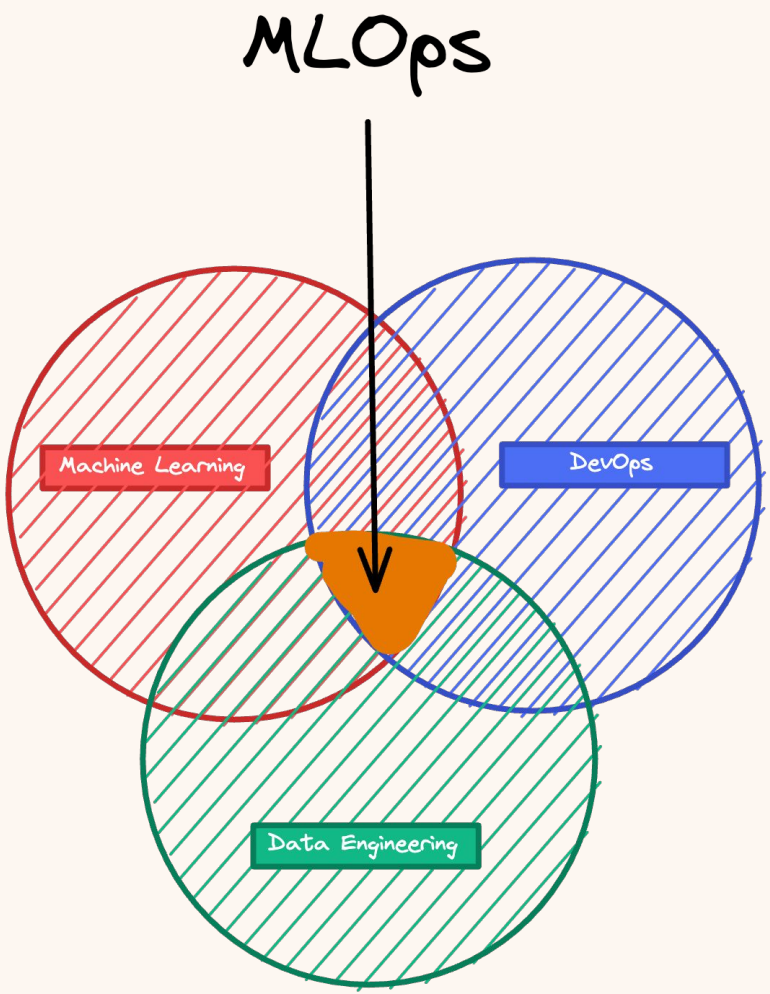
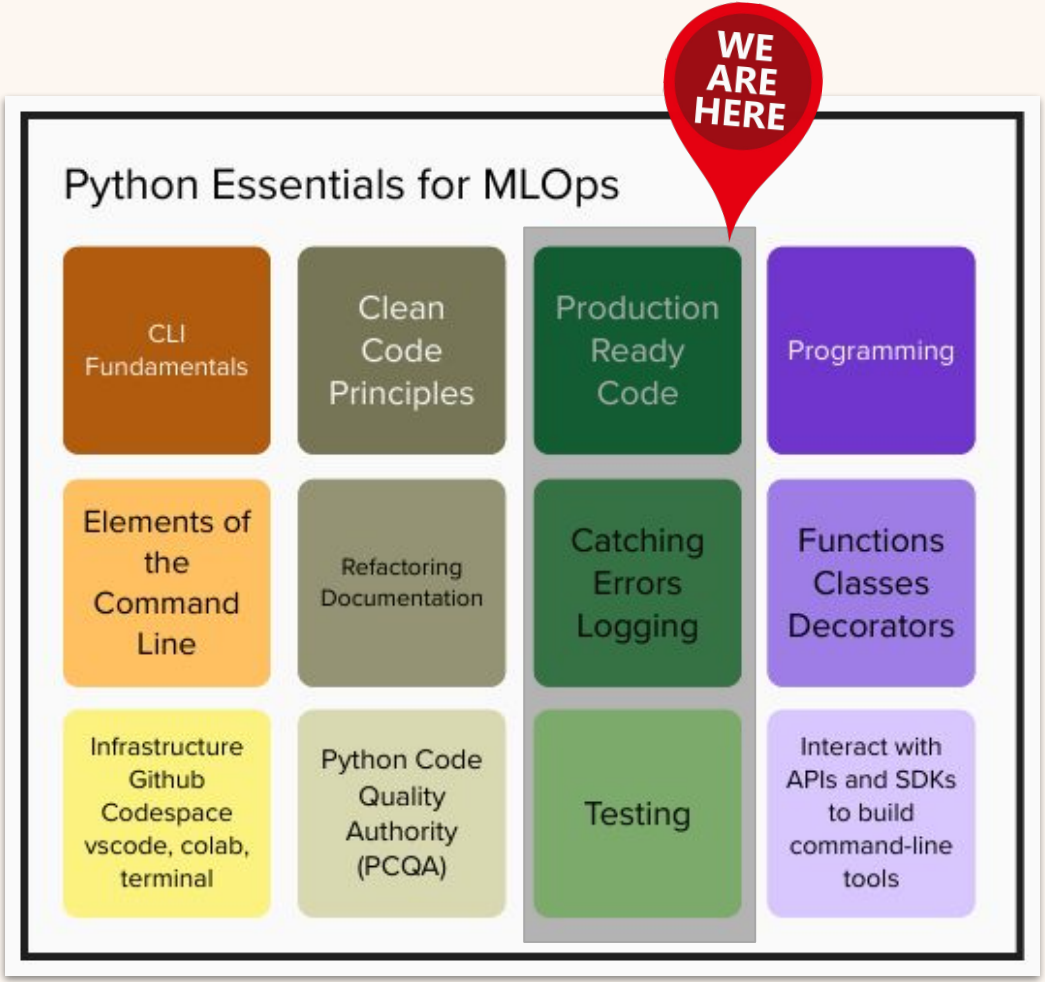
Production Ready Code

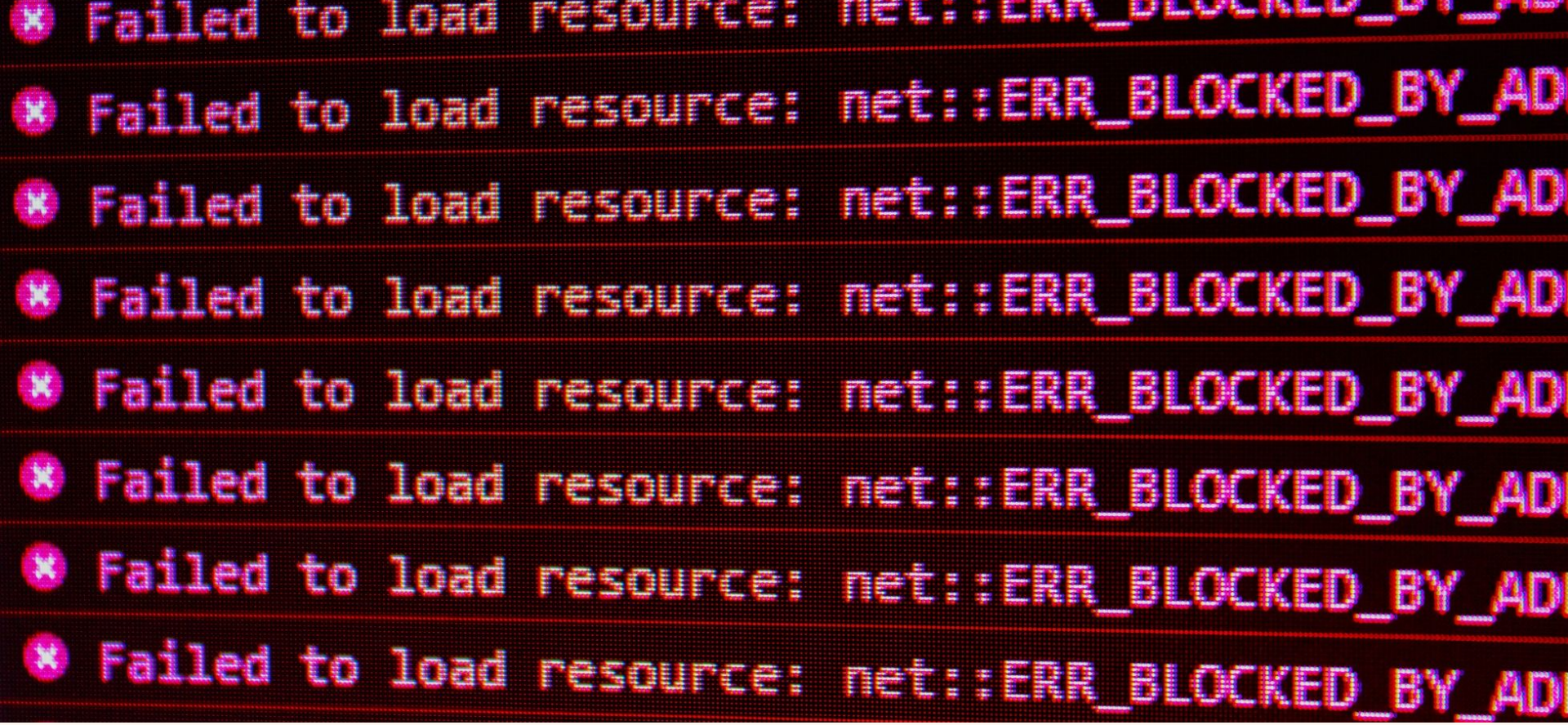
Handling Errors, Writing Tests and Logs

DCA0305
ivanovitch.silva@ufrn.br



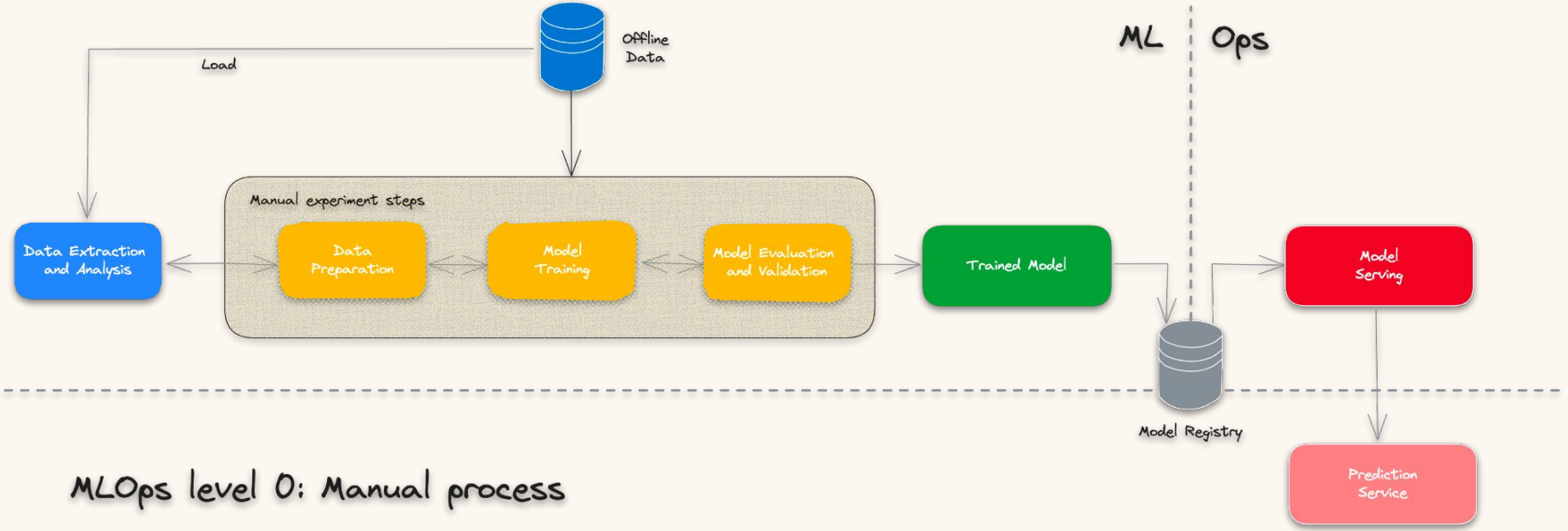

MLOps
Machine Learning
DevOps
Data Engineering


WE ARE HERE
Python Essentials for MLOps
CLI Fundamentals
Clean Code Principles
Production Ready Code
Programming
Elements of the Command Line
Refactoring Documentation
Catching Errors Logging
Functions Classes Decorators
Infrastructure Github Codespace vscode, colab, terminal
Python Code Quality Authority (PCQA)
Testing
Interact with APIs and SDKs to build command-line tools



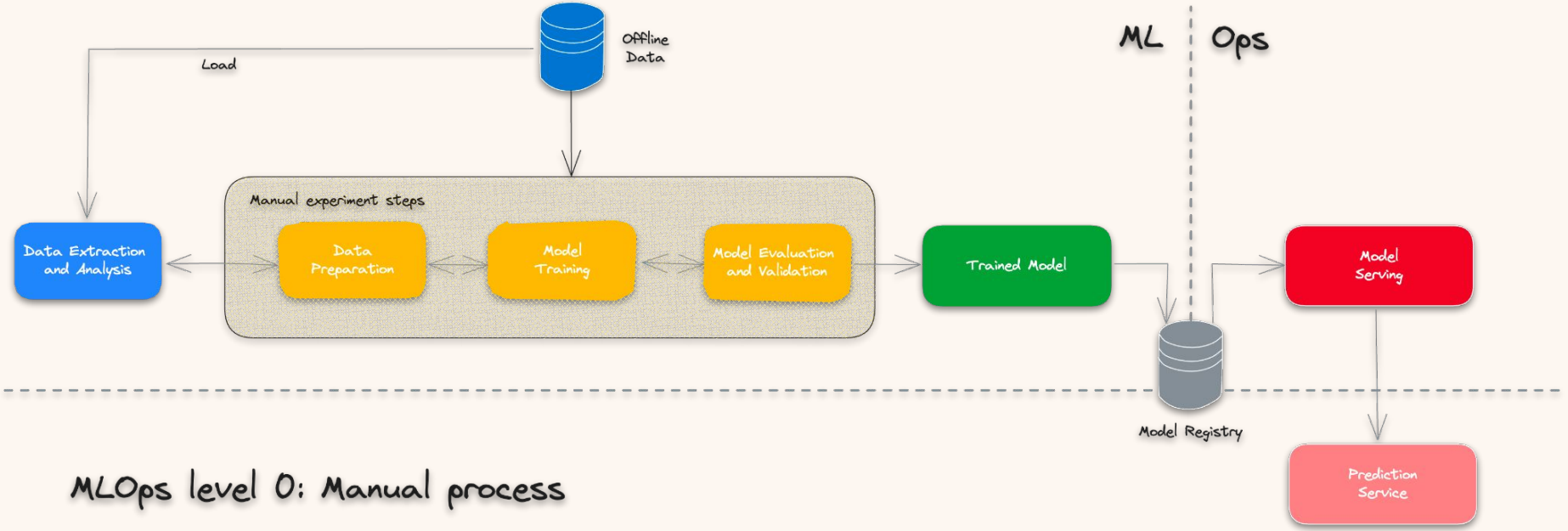
Why is Error Handling Essential in MLOps?

Data Loading Failures: Imagine an MLOps pipeline that trains a new model every day based on newly ingested data. One day, the data source changes its format without notice, and the pipeline fails to handle this change gracefully. As a result, no model is trained for that day, impacting various downstream applications.

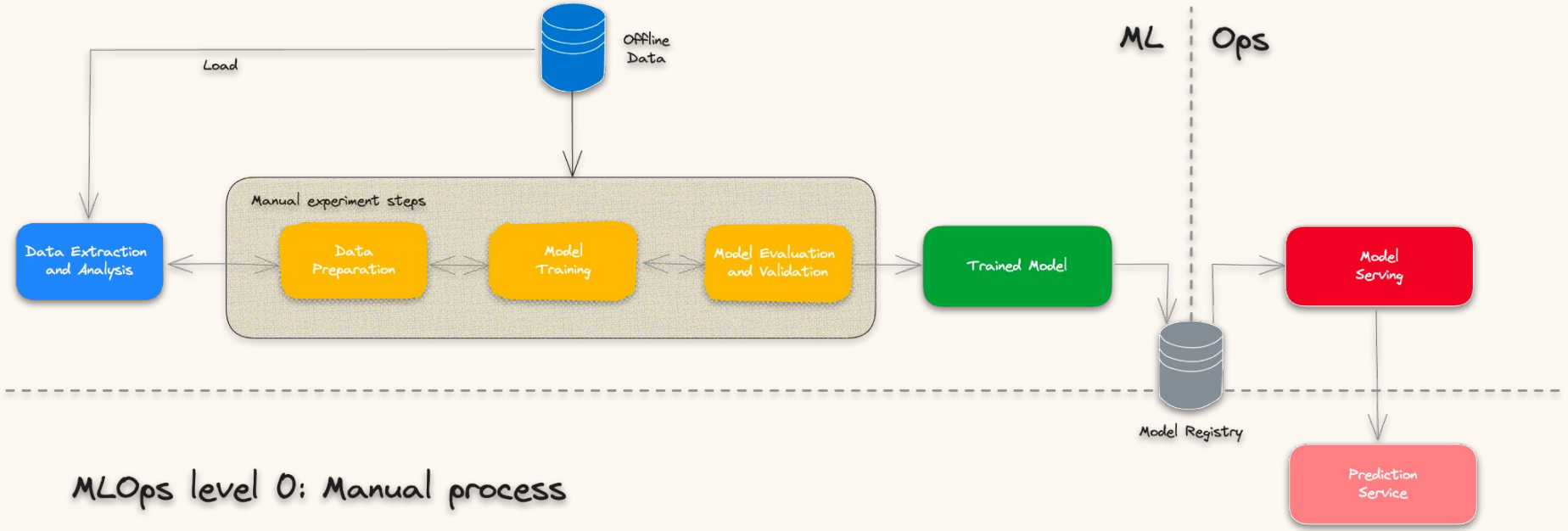


https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning

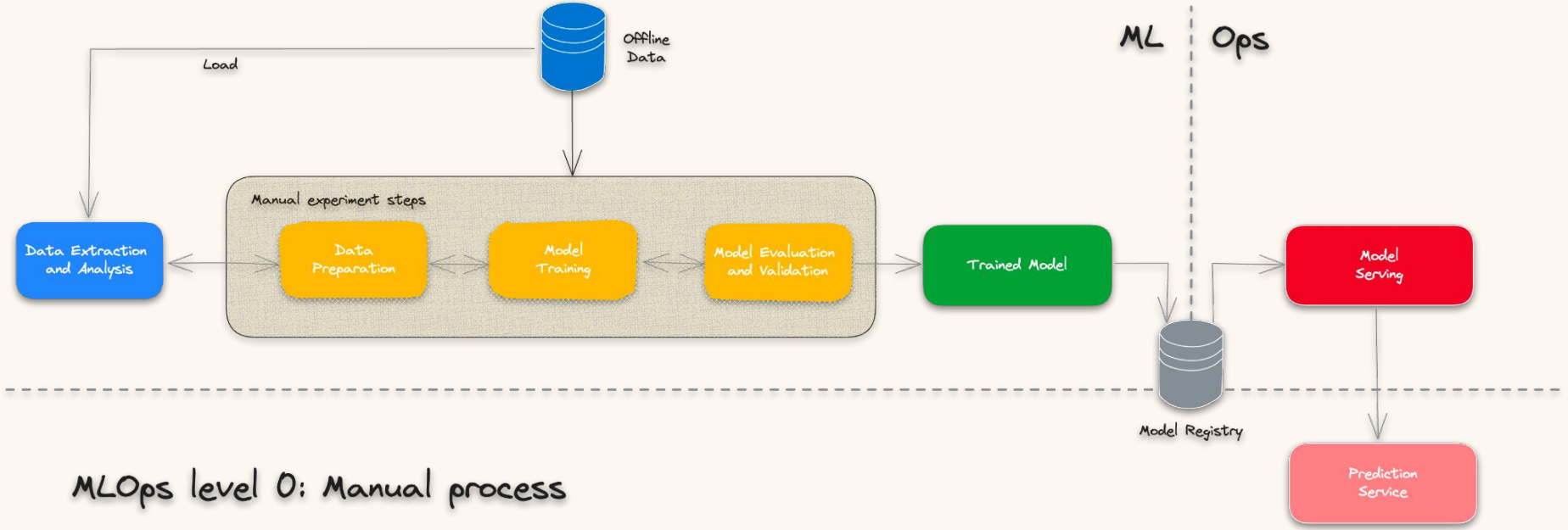
API Rate Limiting: Suppose your MLOps pipeline pulls data from an external API. If the API imposes rate limiting and your system doesn't handle this exception, the pipeline might crash or get stuck in an infinite loop, causing a series of failures downstream.



Resource Exhaustion: Imagine a machine learning model that's part of a recommendation engine for an e-commerce website. Poor error handling results in memory leaks, eventually crashing the recommendation service during peak sales hours, leading to significant loss of revenue.



Timeout Errors: In a complex MLOps setting, various services might depend on each other. Poor handling of timeout errors can cascade, causing a failure in multiple dependent systems.



```
try:
    x = int(input("Enter a number: "))
    y = int(input("Enter another number: "))
    result = x / y
except ZeroDivisionError:
    print("You can't divide by zero!")
except ValueError:
    print("Invalid input. Please enter a number.")
finally:
    print("This will execute no matter what.")
```

What is the output to? and to: and ?

x = 4 and y = 2 x = 4 and y = 0 x = "a" and y = 2

```python
def divide_numbers(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("You can't divide by zero!")
    except TypeError:
        print("Both arguments must be numbers!")
    else:
        print(f"The result is: {result}")
    finally:
        print("This will execute no matter what.")

# Test the function
divide_numbers(10, 2)
divide_numbers(10, 0)
divide_numbers(10, "a")
```

```python
import pandas as pd

def filter_and_save_csv(input_file_path, output_file_path):
    temp_df = None  # Initialize variable to hold the temporary DataFrame

    try:
        # Attempt to read the CSV into a DataFrame
        temp_df = pd.read_csv(input_file_path)

        # Filter rows where the value in the 'age' column is greater than 21
        filtered_df = temp_df[temp_df['age'] > 21]

        # Save the filtered DataFrame to a new CSV file
        filtered_df.to_csv(output_file_path, index=False)

        print(f"Filtered data saved to {output_file_path}")

    except FileNotFoundError:
        print("The input file was not found.")

    except KeyError:
        print("The required column does not exist.")

    finally:
        # Always run this block to delete the temporary DataFrame and free up memory
        print("Cleaning up resources.")
        if temp_df is not None:
            del temp_df
```

Custom Exception

```
# Define a custom exception
class InvalidAgeException(Exception):
    pass

# Using the custom exception
try:
    age = int(input("Enter your age: "))
    if age < 0:
        raise InvalidAgeException("Age cannot be negative")
except InvalidAgeException as e:
    print(e)
```

Custom Exception

```
import pandas as pd

# Custom Exception for DataFrame Validation
class DataFrameValidationException(Exception):
    def __init__(self, message, missing_columns=None, insufficient_data_columns=None):
        super().__init__(message)
        self.missing_columns = missing_columns
        self.insufficient_data_columns = insufficient_data_columns

# Function to Validate DataFrame
def validate_dataframe(df, required_columns, min_non_null_entries):
    missing_columns = [col for col in required_columns if col not in df.columns]
    insufficient_data_columns = [col for col in required_columns if df[col].count() < min_non_null_entries]

    if missing_columns or insufficient_data_columns:
        raise DataFrameValidationException(
            "DataFrame validation failed.",
            missing_columns=missing_columns,
            insufficient_data_columns=insufficient_data_columns
        )
```

8. Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

8.1. Syntax Errors

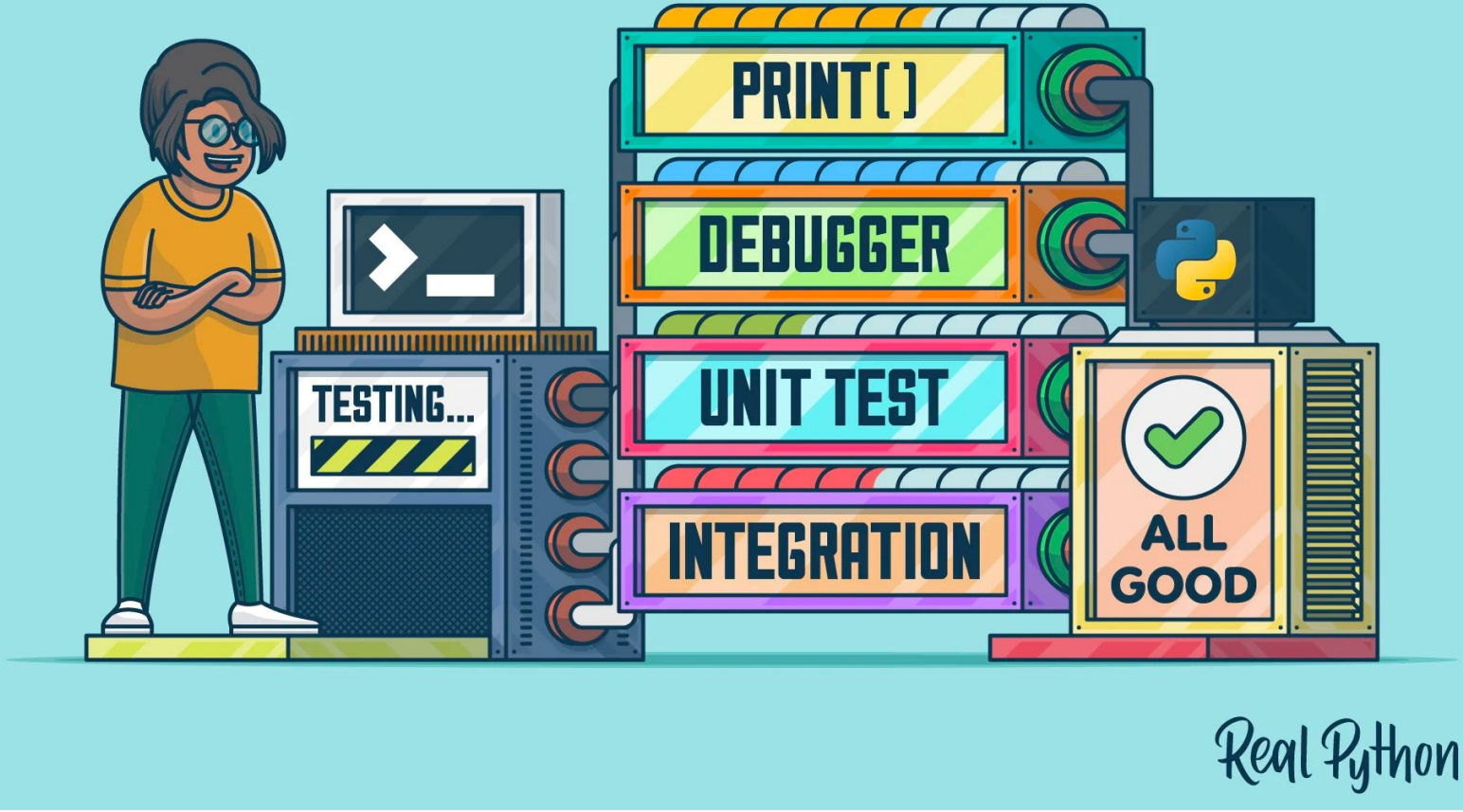
Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function `print()`, since a colon (`':'`) is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

8.2. Exceptions

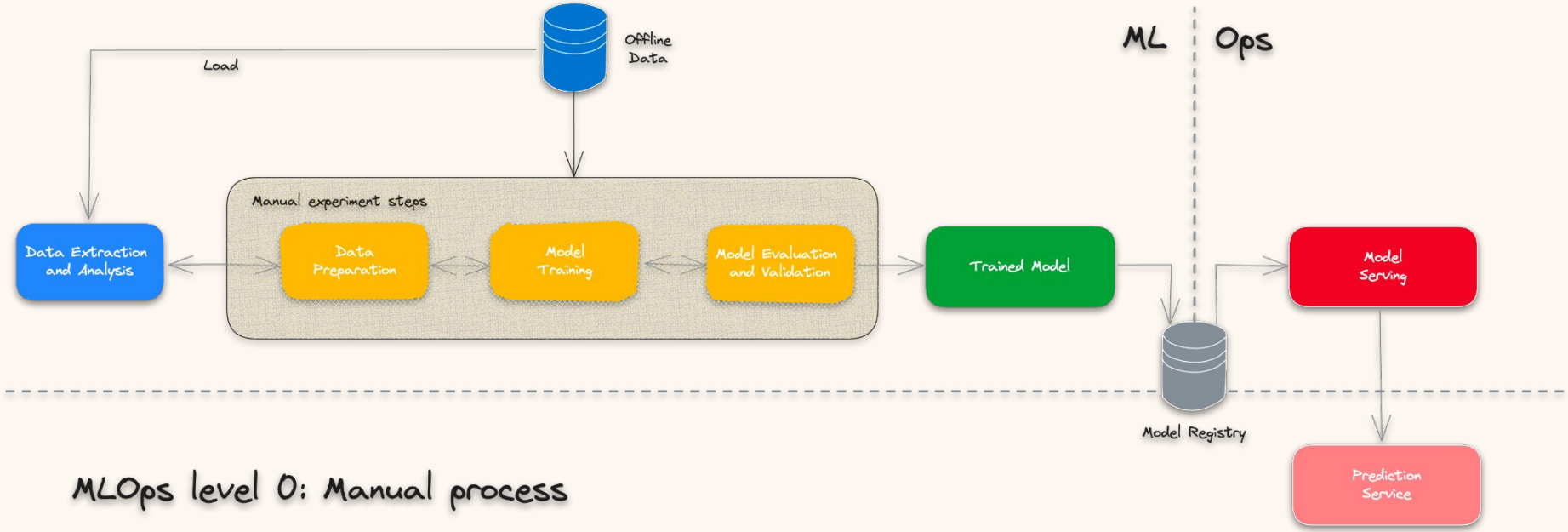
Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:



Debugging is hard, testing is easy (...)

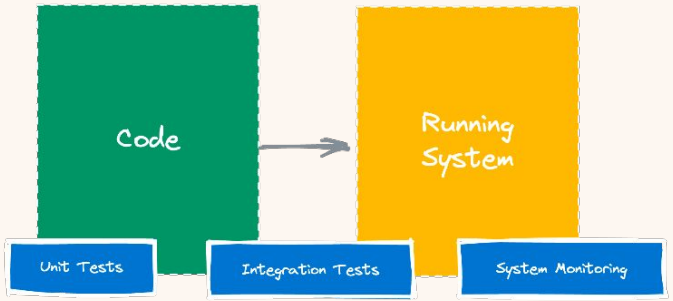
a) Rapid Application Development (RAD) tools
b) Manual & Script-Driven
c) Disconnection between ML and operation
d) Infrequent release iterations
e) No Continuous Integration (CI)
f) No Continuous Delivery (CD)
g) No Continuous Training (CT)

h) Deployment refers to the prediction service (e.g a microservice with a REST API), rather than deploying the entire ML system
i) Lack of active performance monitoring

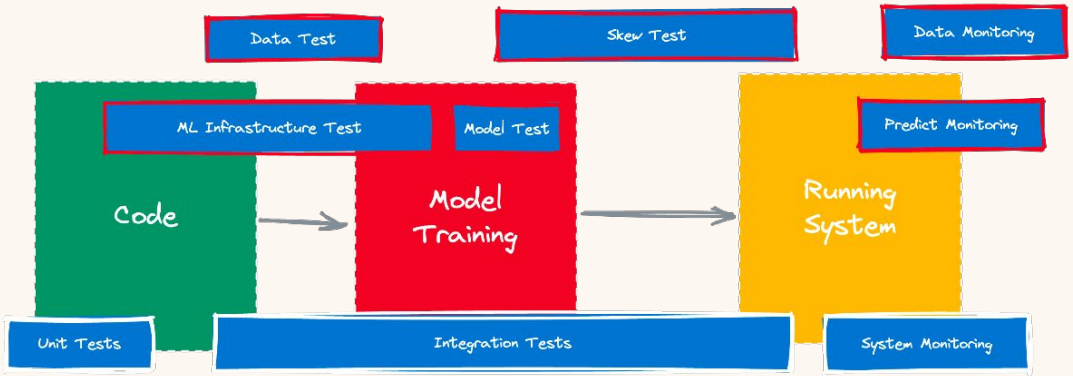


MLOps level 0: Manual process

The complete development pipeline includes three essential components, **data pipeline, ML model pipeline**, and **application pipeline**. In accordance with this separation we distinguish three scopes for testing in ML systems: **tests for features and data**, **tests for model development**, and **tests for ML infrastructure**.

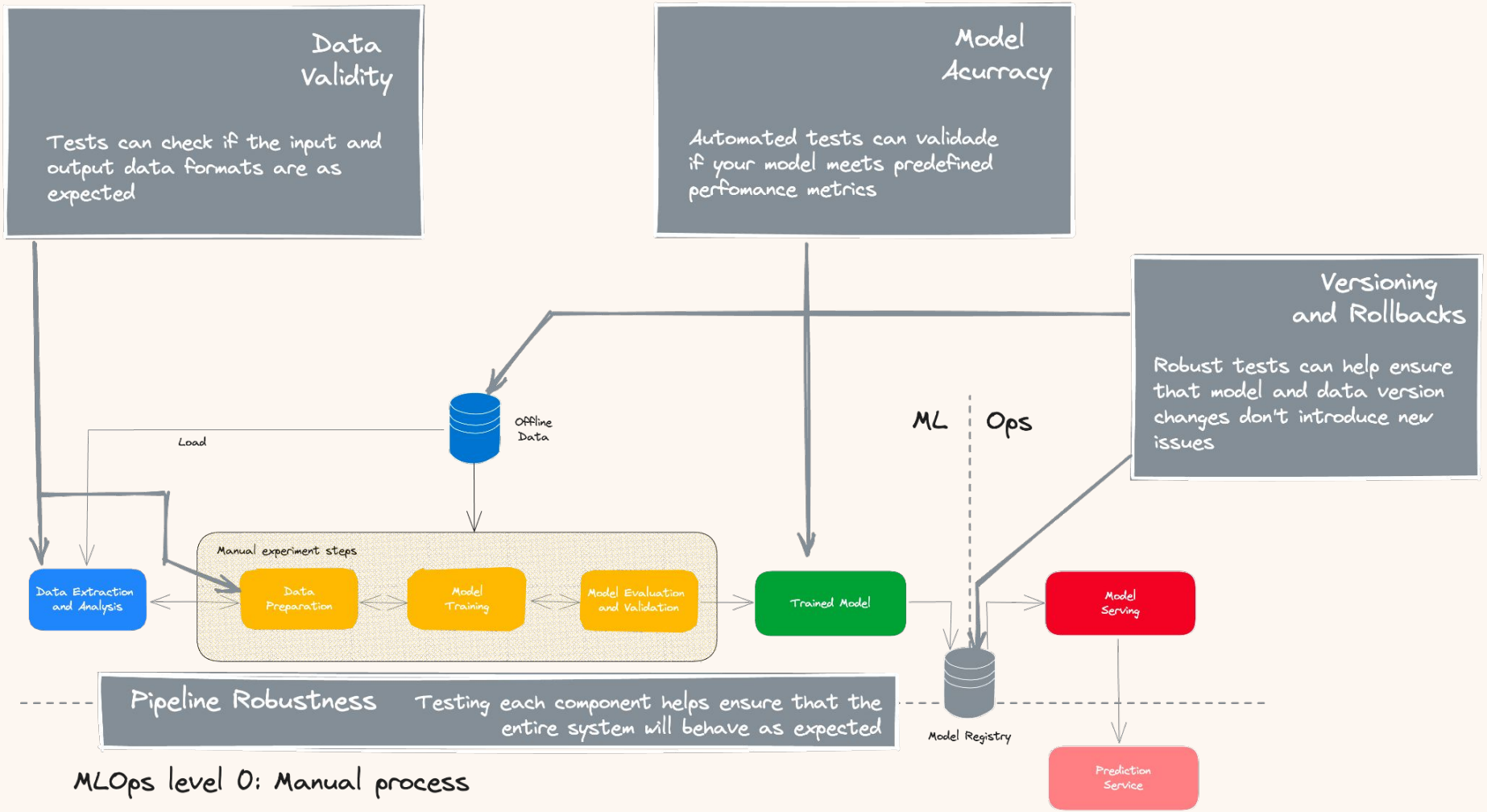


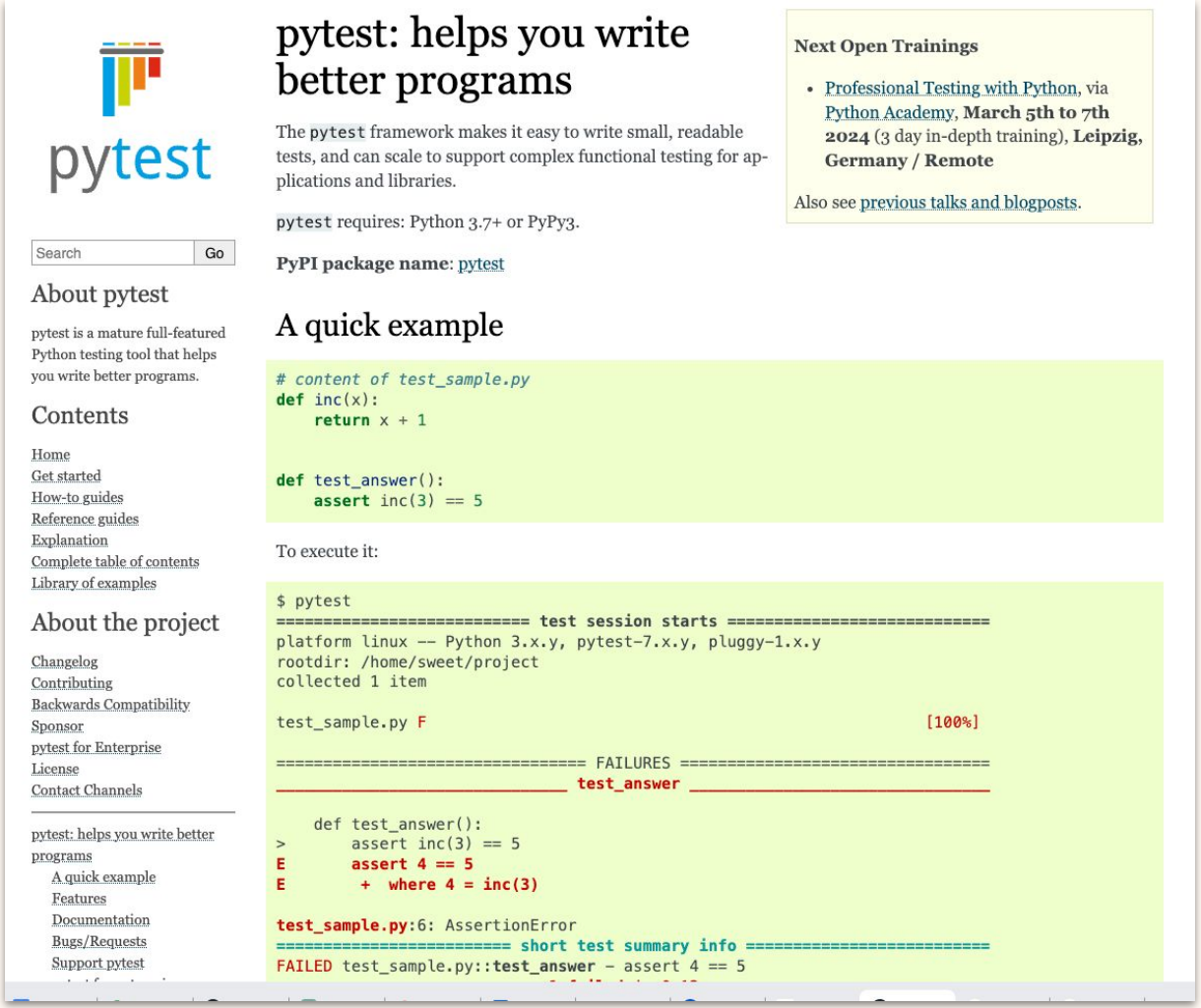
Traditional System Testing and Monitoring



ML-Based System Testing and Monitoring

https://ml-ops.org/content/mlops-principles


Data Validity
Tests can check if the input and output data formats are as expected
Model Acurracy
Automated tests can validade if your model meets predefined perfomance metrics
Versioning and Rollbacks
Robust tests can help ensure that model and data version changes don't introduce new issues
Offline Data
Load
ML
Ops
Manual experiment steps
Data Extraction and Analysis
Data Preparation
Model Training
Model Evaluation and Validation
Trained Model
Model Serving
Model Registry
Prediction Service
Pipeline Robustness
Testing each component helps ensure that the entire system will behave as expected
MLOps level 0: Manual process



pytest

Search Go

About pytest

pytest is a mature full-featured Python testing tool that helps you write better programs.

Contents

About the project

pytest: helps you write better programs

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

`pytest` requires: Python 3.7+ or PyPy3.

PyPI package name: pytest

Next Open Trainings

- Professional Testing with Python, via Python Academy, **March 5th to 7th 2024** (3 day in-depth training), **Leipzig, Germany / Remote**

Also see previous talks and blogposts.

A quick example

```
# content of test_sample.py
def inc(x):
    return x + 1


def test_answer():
    assert inc(3) == 5
```

To execute it:

```
$ pytest
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-7.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 1 item

test_sample.py F                                                  [100%]

================================ FAILURES ================================
_____________________________ test_answer _____________________________

    def test_answer():
>       assert inc(3) == 5
E       assert 4 == 5
E        +  where 4 = inc(3)

test_sample.py:6: AssertionError
======================== short test summary info ========================
FAILED test_sample.py::test_answer - assert 4 == 5
```

```
# math_operations.py

def add(a, b):
    return a + b
```

```
# test_math_operations.py

from math_operations import add

def test_add():
    result = add(3, 5)
    assert result == 8
```

```
pytest test_math_operations.py
```

```
pip install pytest pytest-sugar
```

Pytest Fixtures

```python
# ml_operations.py

def normalize(numbers):
    max_num = max(numbers)
    min_num = min(numbers)
    return [(x - min_num) / (max_num - min_num) for x in numbers]
```

```python
# test_ml_operations.py

import pytest
from ml_operations import normalize

@pytest.fixture
def sample_data():
    return [5, 10, 15, 20, 25]

def test_normalize(sample_data):
    result = normalize(sample_data)
    assert result == [0.0, 0.25, 0.5, 0.75, 1.0]
```

```
pytest test_ml_operations.py
```

Pytest Fixtures: using a fixture across multiple test cases

```python
# conftest.py

import pytest

@pytest.fixture
def sample_data():
    return [5, 10, 15, 20, 25]
```

```python
# test_ml_operations2.py

from ml_operations import normalize

def test_normalize_again(sample_data):
    result = normalize(sample_data)
    assert result == [0.0, 0.25, 0.5, 0.75, 1.0]
```

```
pytest test_ml_operations2.py
```

Once the ML has been deployed, it need to be monitored

logging — Logging facility for Python

Source code: Lib/logging/__init__.py

This module defines functions and classes which implement a flexible event logging system for applications and libraries.

The key benefit of having the logging API provided by a standard library module is that all Python modules can participate in logging, so your application log can include your own messages integrated with messages from third-party modules.

Important

This page contains the API reference information. For tutorial information and discussion of more advanced topics, see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

The simplest example:

```
>>> import logging
>>> logging.warning('Watch out!')
WARNING:root:Watch out!
```

The module provides a lot of functionality and flexibility. If you are unfamiliar with logging, the best way to get to grips with it is to view the tutorials (**see the links above and on the right**).

The basic classes defined by the module, together with their functions, are listed below.

- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- Formatters specify the layout of log records in the final output.

```python
import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug("This is a debug message") # This will be printed
logging.info("This is an info message") # This will be printed
logging.warning("This is a warning message") # This will be printed
logging.error("This is an error message") # This will be printed
logging.critical("This is a critical message") # This will be printed
```

```python
import logging

logging.basicConfig(level=logging.WARNING)

logging.debug("This is a debug message") # Won't be printed
logging.info("This is an info message") # Won't be printed
logging.warning("This is a warning message") # This will be printed
logging.error("This is an error message") # This will be printed
logging.critical("This is a critical message") # This will be printed
```

DEBUG: detailed information for debugging purposes
INFO: general confirmations that things are working as expected
WARNING: an indication that something unexpected happened or may happen soon but the program still works as expected
ERROR: due to a severe problem, the software has not been able to perform some function
CRITICAL: a very severe error that will likely lead to the application terminating

When you use %%python in a Google Colab cell, it runs the code in that cell in a separate Python process. This new process does not inherit the logging configuration from the main Colab environment. Therefore, the logging starts with its default configuration.

```
%%python
import logging

logging.basicConfig(level=logging.WARNING)

logging.debug("This is a debug message") # Won't be printed
logging.info("This is an info message") # Won't be printed
logging.warning("This is a warning message") # This will be printed
logging.error("This is an error message") # This will be printed
logging.critical("This is a critical message") # This will be printed
```

Example 1: Logging During Data Preprocessing

```
import logging
import pandas as pd

logging.basicConfig(filename='./results.log',
                    level=logging.INFO,
                    filemode='w',
                    format='%(asctime)s - %(name)s - \
                    %(levelname)s - %(message)s',
                    datefmt='%Y-%m-%d %H:%M:%S'
                   )

try:
    # Load data
    df = pd.read_csv("data.csv")
    logging.info("ℹ️ Data loaded successfully.")
except FileNotFoundError:
    logging.error("❌ File not found.")

# Some more preprocessing code...
```

Example 2: Logging During Model Training

```
import logging
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification

logging.basicConfig(
    filename='./results.log',
    level=logging.INFO,
    filemode='w',
    format='%(name)s - %(levelname)s - %(message)s')

try:
    X, y = make_classification()
    clf = RandomForestClassifier()

    logging.info("ℹ️ Starting model training.")
    clf.fit(X, y)
    logging.info("ℹ️ Model training complete.")

except Exception as e:
    logging.critical(f"💥 Training failed: {e}")
```

Example 3: Logging During API Calls

```
import logging
import requests

logging.basicConfig(
    filename='./results.log',
    level=logging.INFO,
    filemode='w',
    format='%(name)s - %(levelname)s - %(message)s')

try:
    response = requests.get("http://api.example.com/data")
    if response.status_code == 200:
        logging.info("ℹ️ Successfully fetched data from API.")
    else:
        logging.warning(f"⚠️ Unexpected status code: {response.status_code}")

except requests.RequestException as e:
    logging.error(f"❌ API request failed: {e}")
```