Dev(ML)Ops Engineering Sessions 3

Bachelors in Data and Business Analytics

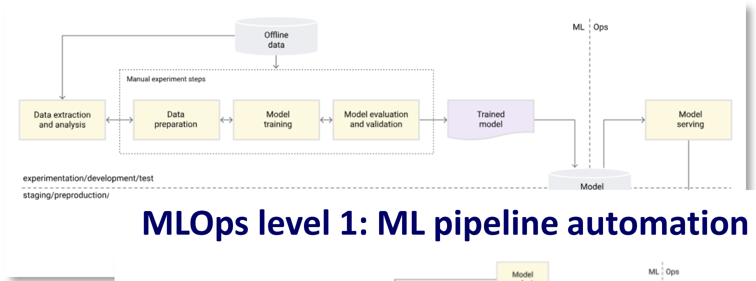
Madrid, Sep 2023

ie UNIVERSITY

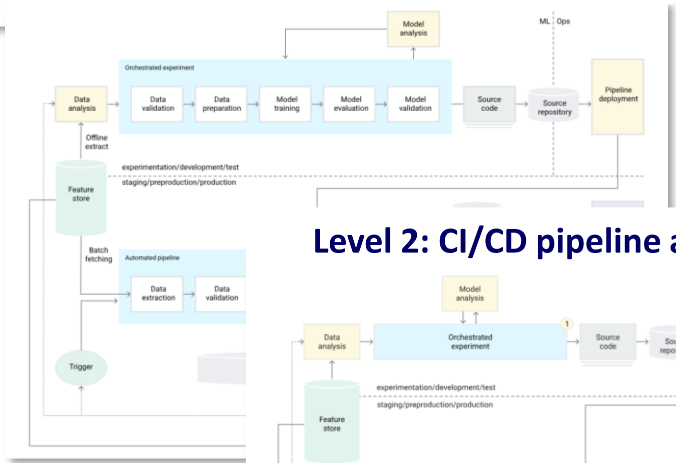
Additional material: MLOps: Continuous delivery and automation pipelines

MLOps is not a destination, but a journey. Involving people, processes, tools, data and governance

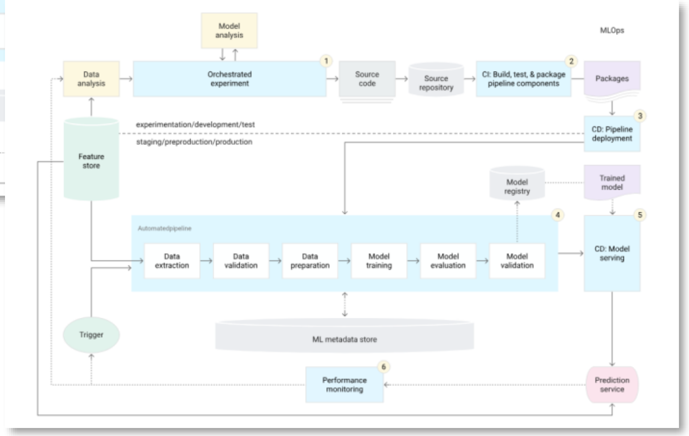
Level 0: Manual process

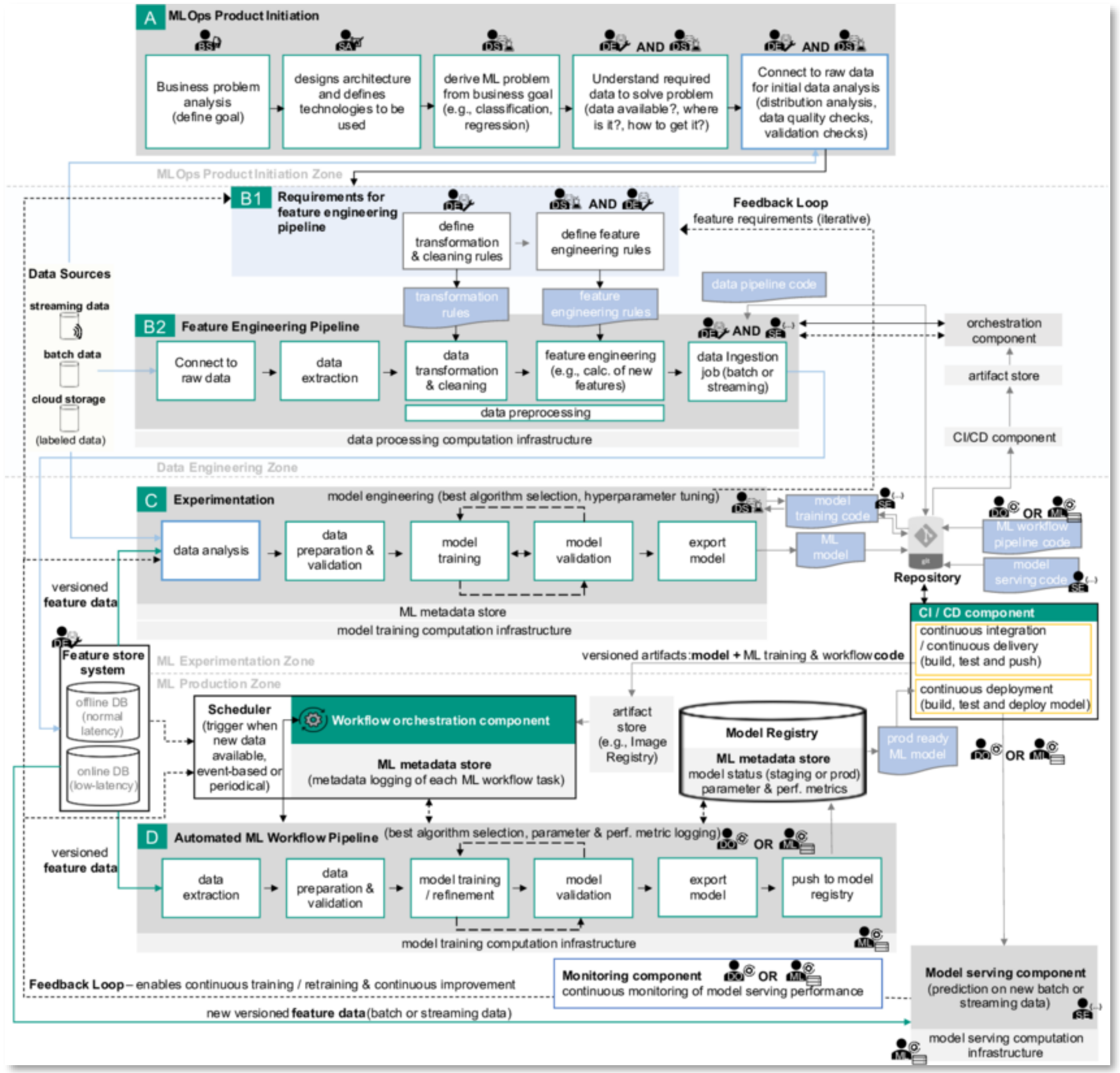


MLOps level 1: ML pipeline automation



Level 2: CI/CD pipeline automation

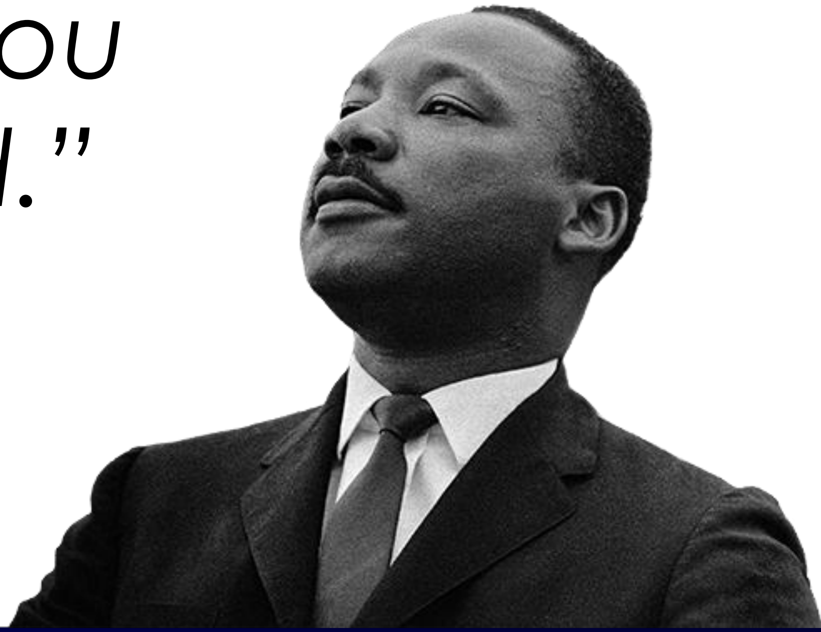



A ML Ops Product Initiation
Business problem analysis (define goal)
designs architecture and defines technologies to be used
derive ML problem from business goal (e.g., classification, regression)
Understand required data to solve problem (data available?, where is it?, how to get it?)
Connect to raw data for initial data analysis (distribution analysis, data quality checks, validation checks)
AND
MLOps Product Initiation Zone
B1 Requirements for feature engineering pipeline
define transformation & cleaning rules
define feature engineering rules
Feedback Loop
feature requirements (iterative)
Data Sources
streaming data
batch data
cloud storage
(labeled data)
transformation rules
feature engineering rules
data pipeline code
B2 Feature Engineering Pipeline
Connect to raw data
data extraction
data transformation & cleaning
feature engineering (e.g., calc. of new features)
data ingestion job (batch or streaming)
data preprocessing
data processing computation infrastructure
orchestration component
artifact store
CI/CD component
Data Engineering Zone
C Experimentation
model engineering (best algorithm selection, hyperparameter tuning)
data analysis
data preparation & validation
model training
model validation
export model
ML metadata store
model training computation infrastructure
model training code
ML model
ML workflow pipeline code
model serving code
OR
Repository
versioned feature data
CI / CD component
continuous integration / continuous delivery (build, test and push)
continuous deployment (build, test and deploy model)
Feature store system
offline DB (normal latency)
online DB (low-latency)
ML Experimentation Zone
ML Production Zone
versioned artifacts: model + ML training & workflow code
Scheduler (trigger when new data available, event-based or periodical)
Workflow orchestration component
ML metadata store (metadata logging of each ML workflow task)
artifact store (e.g., Image Registry)
Model Registry
ML metadata store
model status (staging or prod) parameter & perf. metrics
prod ready ML model
D Automated ML Workflow Pipeline
(best algorithm selection, parameter & perf. metric logging)
data extraction
data preparation & validation
model training / refinement
model validation
export model
push to model registry
model training computation infrastructure
Feedback Loop – enables continuous training / retraining & continuous improvement
Monitoring component
continuous monitoring of model serving performance
Model serving component (prediction on new batch or streaming data)
model serving computation infrastructure
new versioned feature data (batch or streaming data)

Source: Google

"If you can't fly, run; if you can't run, walk; if you can't walk, crawl; but whatever you do you have to keep moving forward."

— Martin Luther King, Jr.



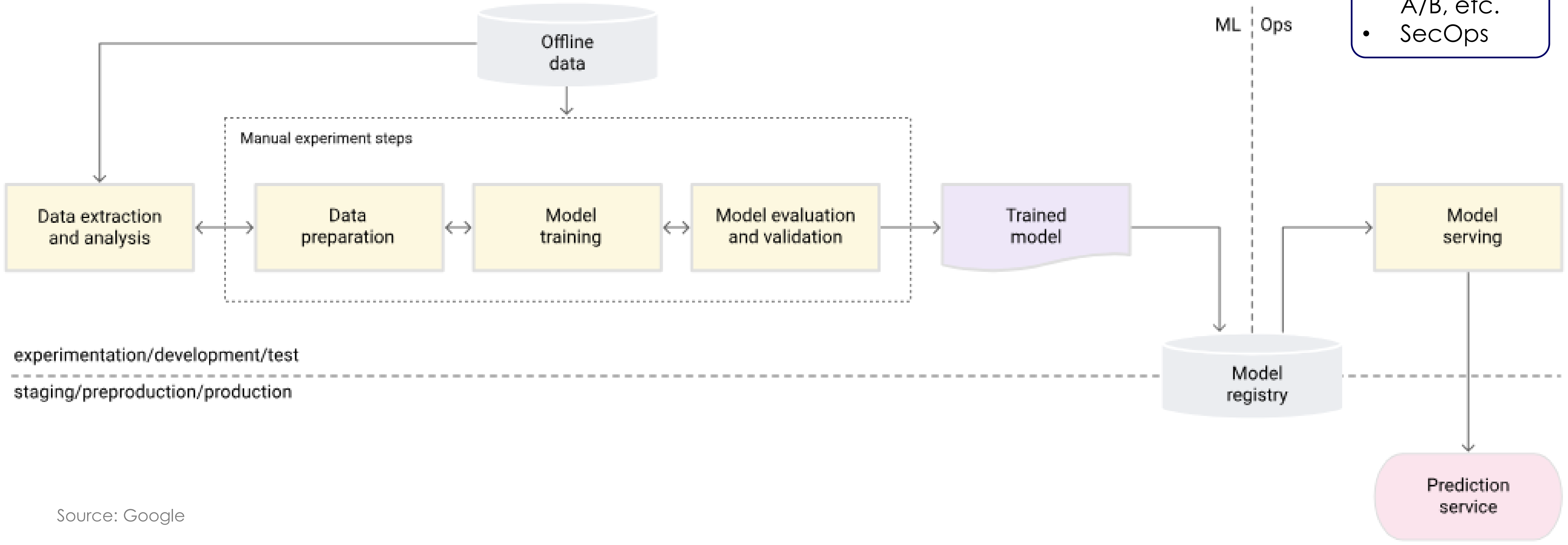
Implementing digital transformations requires a phased approach

	CRAWLING Try!	**WALKING** Learn!	**RUNNING** Grow!
Strategy & Governance	– What is our digital **vision**, and how does it align with our **business objectives**?	– How are we refining our **digital roadmap** based on **initial learnings**?	– How is digital strategy **integrated** into our **overall corporate strategy**?
Tech & Infrastructure	– Do we have the **foundational technology** to **support** initial digital initiatives?	– How are we **scaling our technology** to support broader digital efforts?	– Are our technology platforms **optimized** for **agility and innovation**?
Data, Analytics & AI	– What **critical data** do we need, and what **basic analytics** capabilities do we have?	– How are we enhancing **data quality** and **integration** across systems, and are we leveraging **predictive analytics** to inform decisions?	– Is our data infrastructure enabling **real-time insights** and **decision-making**, and can we use **AI to drive innovation** and competitive advantage?
Process & Delivery	– What **processes need** to be **digitized first** for quick wins?	– How are we **standardizing** and **automating** digital processes?	– Are our processes **agile** and **responsive** to market changes?
Talent & Culture	– Do we have the **necessary skills and mindset** for digital transformation?	– How are we fostering **a culture of continuous learning** and **innovation**?	– Is our organizational culture fully embracing **digital as a core competency**?

Manual ML workflows hinder scalability and reliability, highlighting the need for MLOps automation to streamline operations and reduce errors

Business Problem/ Opportunity definition

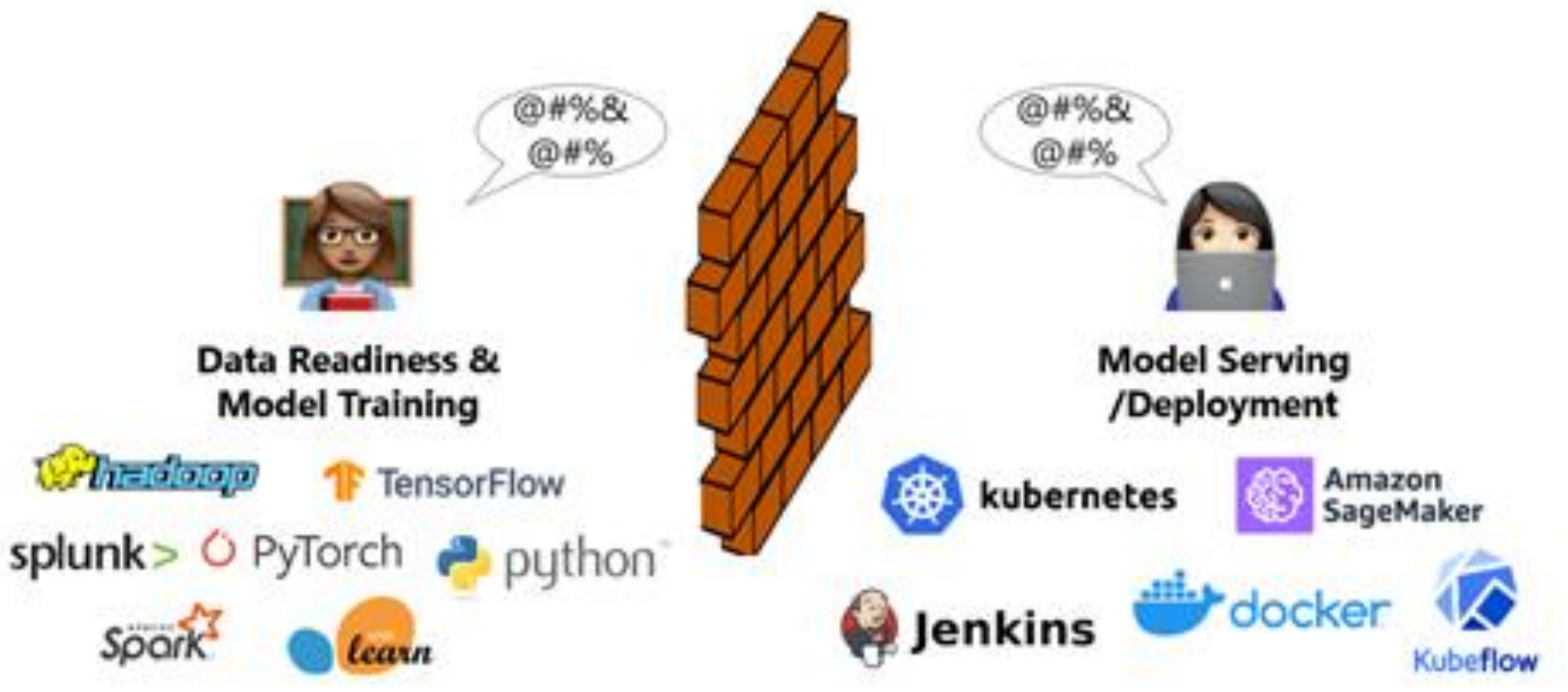
- RestAPIs
- Testing suit, Canary, A/B, etc.
- SecOps



Source: Google

Root cause of all evil

“Throwing it over the wall” paradigm



ML / OPs

Good code isn't just easier to read, it's essential for a successful ML project

Coding best practices

- Writing clean and **modular** code
- **Refactoring** code
- **Optimizing** code for efficiency
- Writing **documentation**
- Following **PEP8** guidelines and **linting**

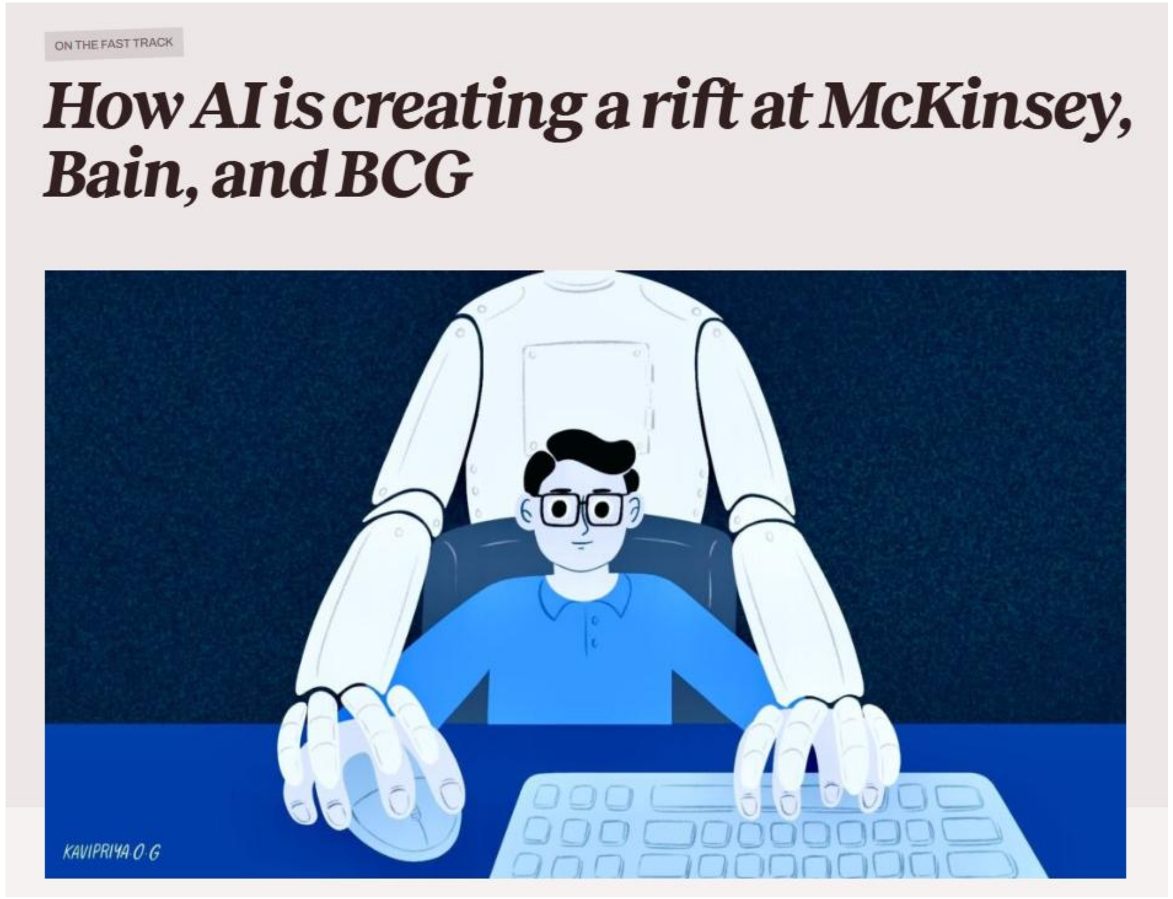
Production ready

- **Unit test** and I/O **data validation**
- **Logging** and **tracking**
- **Command line** arguments

GitHub collaboration

- Review of the add, commit, and push
- Branching
- Common version control workflows
- Code reviews

Source: Based on Udacity framework

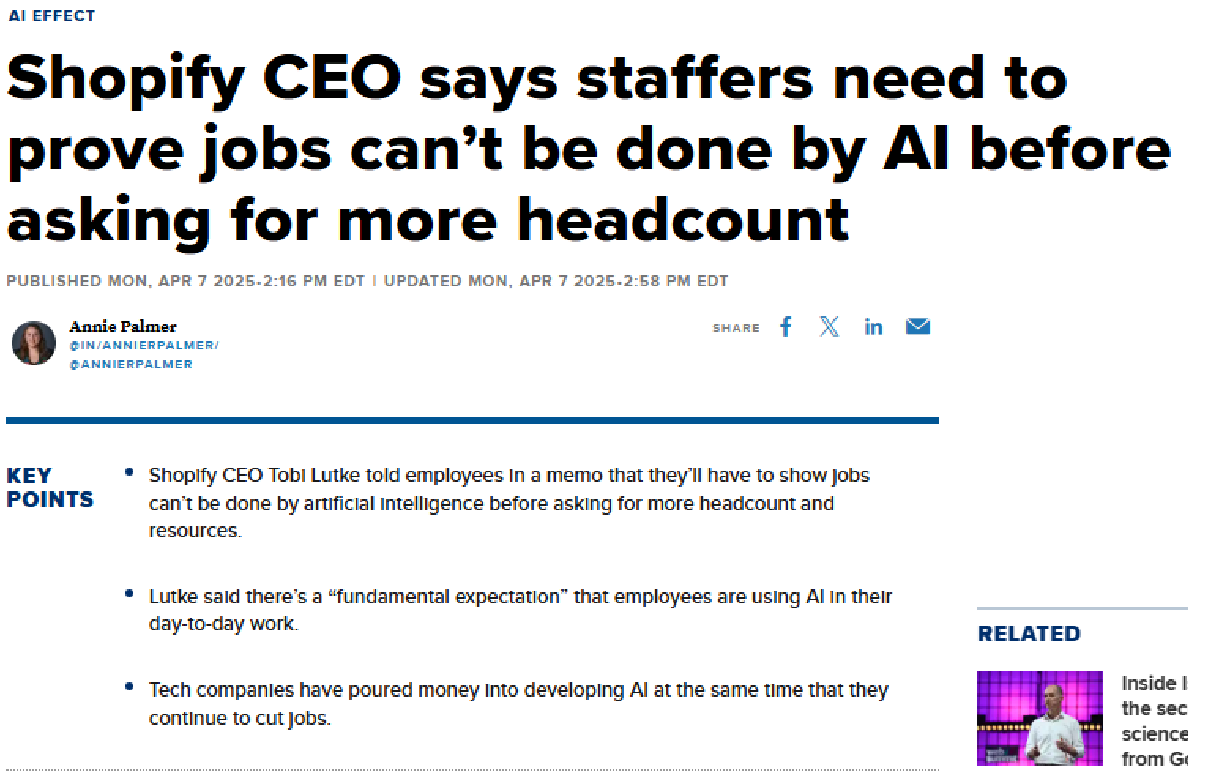


ON THE FAST TRACK

How AI is creating a rift at McKinsey, Bain, and BCG

"a senior partner .... thought they could use an AI chatbot to parse data and ready the ***assessment in a day****… The same thing would have* ***earlier taken two to three weeks****"*

*"****Before asking for more Headcount*** *and resources, teams must* ***demonstrate*** *why they cannot get what they want* ***done using AI****…"*



AI EFFECT

Shopify CEO says staffers need to prove jobs can't be done by AI before asking for more headcount

PUBLISHED MON, APR 7 2025-2:16 PM EDT | UPDATED MON, APR 7 2025-2:58 PM EDT

Annie Palmer
@IN/ANNIEPALMER/
@ANNIEPALMER

SHARE

KEY POINTS

- Shopify CEO Tobi Lutke told employees in a memo that they'll have to show jobs can't be done by artificial intelligence before asking for more headcount and resources.
- Lutke said there's a "fundamental expectation" that employees are using AI in their day-to-day work.
- Tech companies have poured money into developing AI at the same time that they continue to cut jobs.

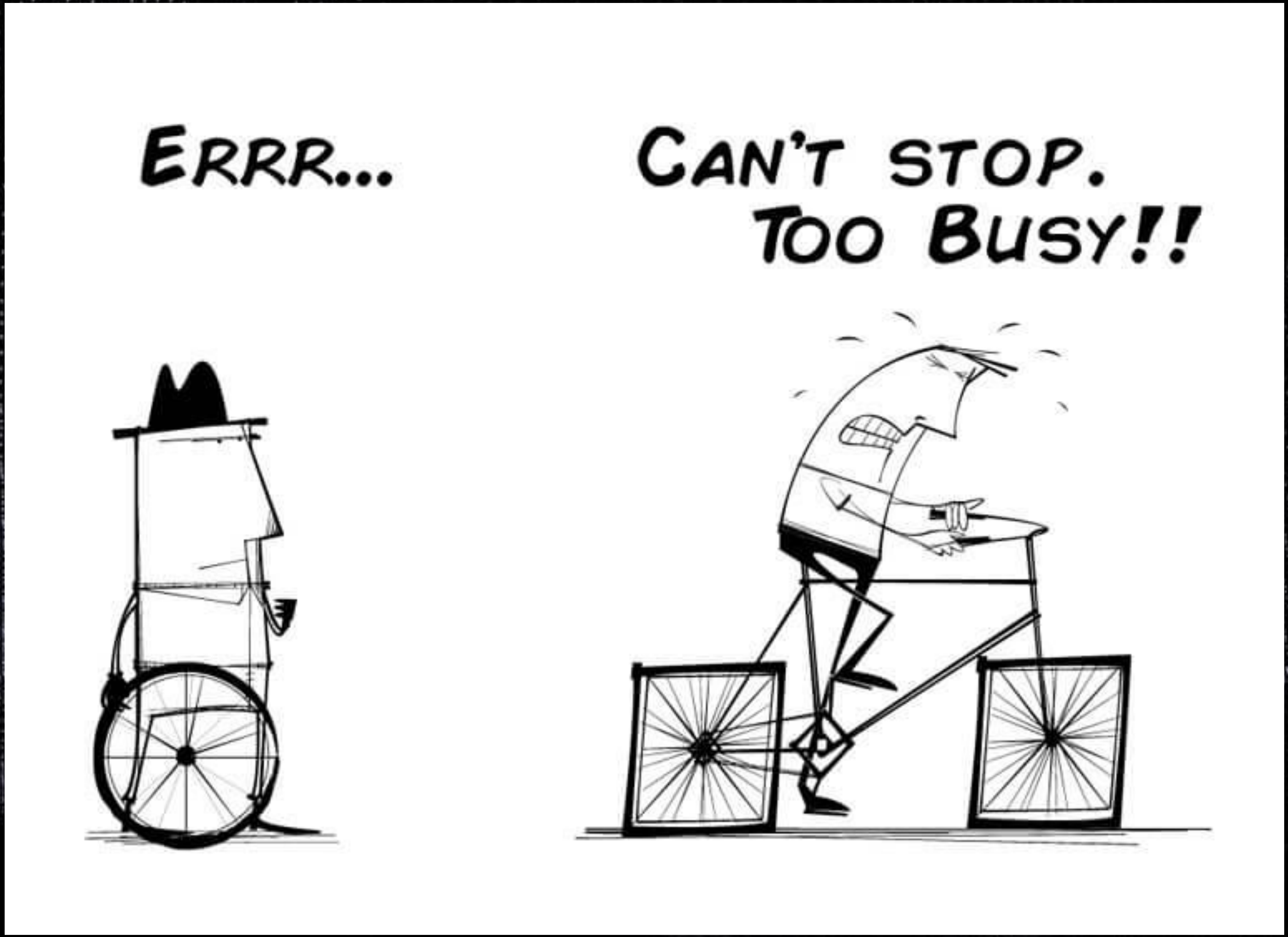
RELATED

Inside I
the sec
science
from G

These “least-expected-standards” when working in a professional environment, require time and commitment

Clean code	**Readable, simple, and concise**; crucial for collaboration and maintainability e.g. Pythonic, Code Reviews – **Linting** Automated checks for syntax, style, and logic errors e.g. pylint – **PEP8** Python style guide for clean code e.g. autopep8
Modular code	Logically split into **functions/modules** for organization, efficiency, and reuse e.g. cookiecutter.io
Module	A .py file enabling **reusable code** via import (`python main.py -h`)
Refactoring	Rewriting code to **improve speed, readability, and maintainability** without changing functionality e.g. 20% tech debt (Be aware of tech debt!)
Optimization	Enhancing code efficiency e.g. execution time
Documentation	Written explanations of software components e.g. markdown

TECHNICAL DEBT
20% OF NEXT SPRINT
imgflip.com


ERRR...
CAN'T STOP. TOO BUSY!!

Structure main.py to isolate execution logic, enable reuse, handle inputs cleanly

- Define a **main()** function to encapsulate the program's entry point
- Use **if __name__ == "__main__":** to ensure code executes only when the script is run directly
- Keep the main() **function concise**; **delegate tasks** to other functions or modules
- Avoid executing code at the top level; place **execution logic within functions**
- Include a **docstring** at the top of the file to describe the script's purpose
- Handle **command-line** arguments e.g. using the **argparse** module for flexibility
- Implement proper **error handling** within the main() function to manage exceptions gracefully
- Ensure that main.py serves as a **clear and minimal entry point**, delegating complex logic to other modules

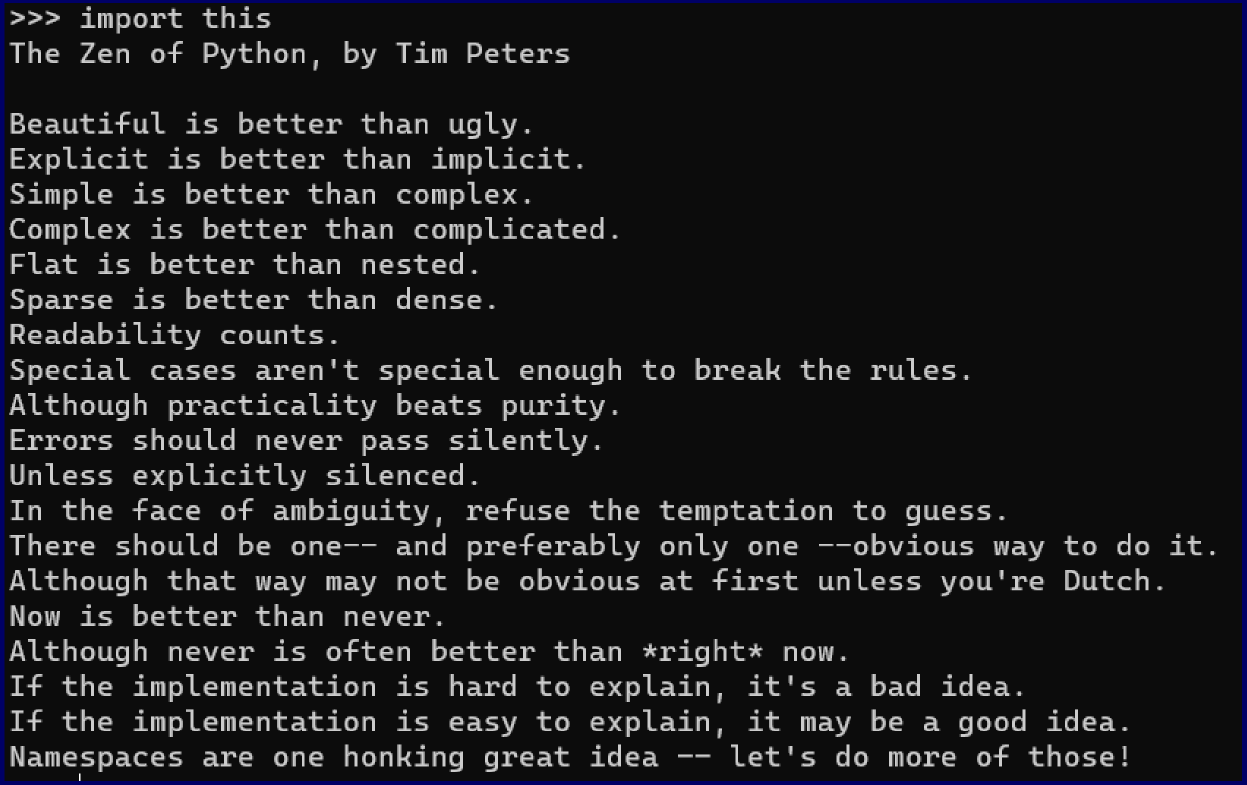
```python
"""
main.py – Illustrates Python main script best practices

This script shows how to:
- Encapsulate execution logic inside a main() function
- Use argparse for flexible input handling
- Delegate tasks to modular functions
- Prevent top-level code execution
- Gracefully handle errors
"""

import argparse


def process_name(name: str) -> str:
    """Return a formatted greeting message."""
    return f"Hello, {name}!"


def main():
    """Main entry point for the script."""
    try:
        parser = argparse.ArgumentParser(description="Greet the user by name.")
        parser.add_argument("name", help="Name of the person to greet")
        args = parser.parse_args()

        message = process_name(args.name)
        print(message)

    except Exception as e:
        print(f"An error occurred: {e}")


if __name__ == "__main__":
    main()
```

Quick & Dirty exercise

1. Create a environment.yml
2. Create a main.py
3. Run the CLI file
4. Create a module.py
5. Create a packagem (__init__.py)

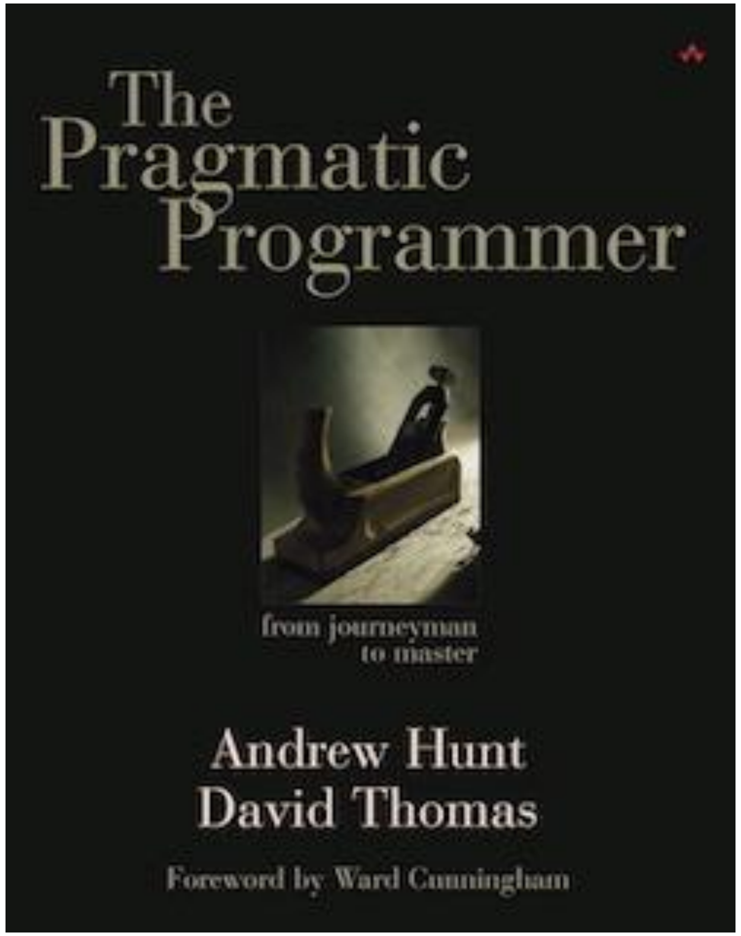
```
conda env create -f environment.yml
python main.py –h
python main.py <argument>
```

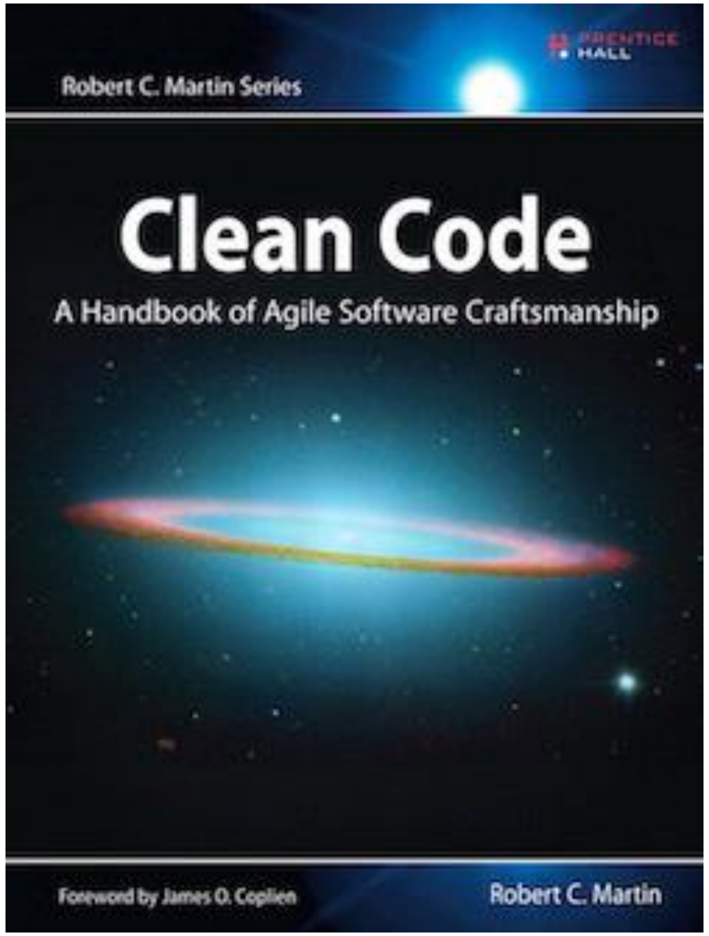
Wisdom distilled into lines, guiding souls to craft code as the river shapes the stone - ChatGPT

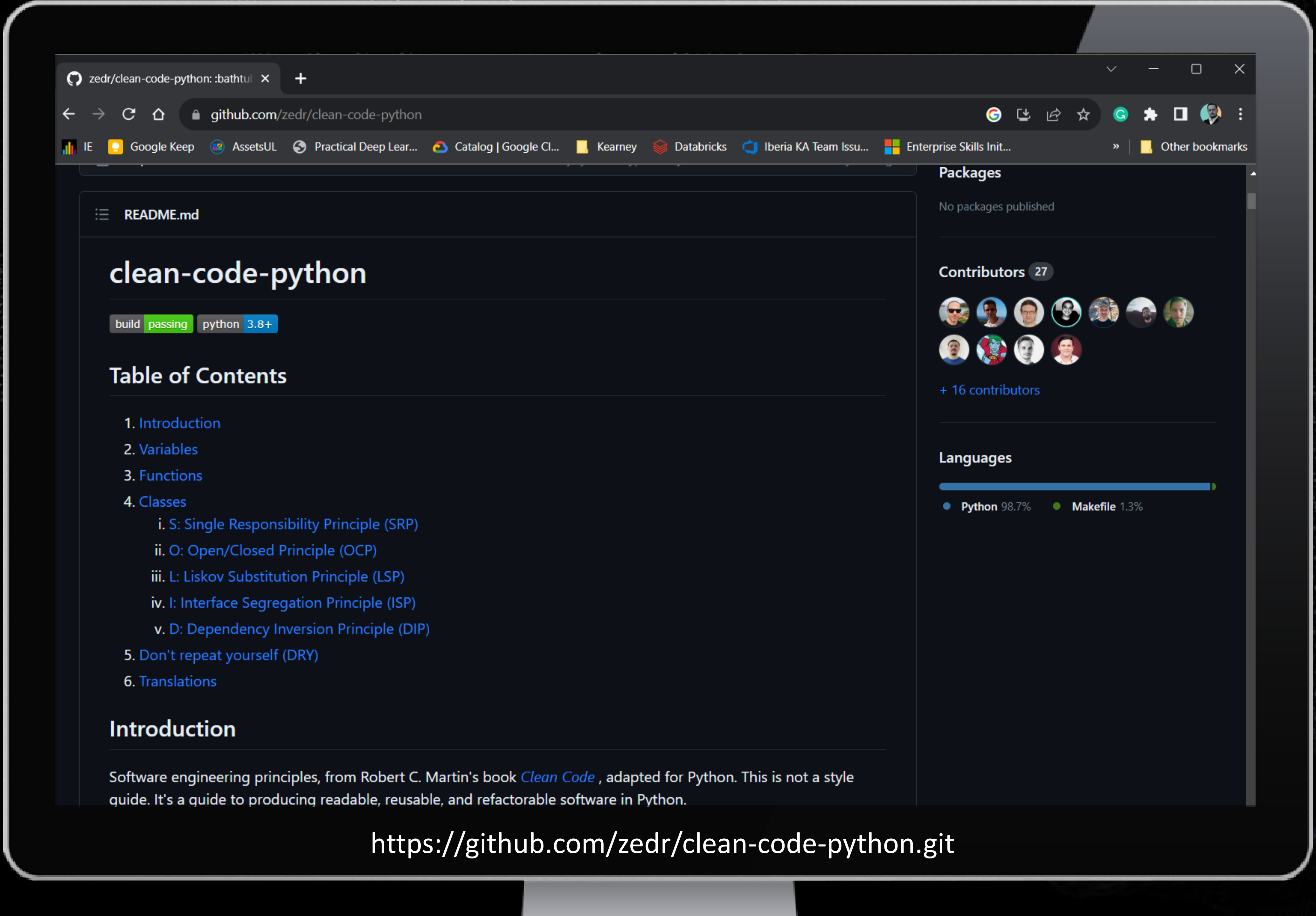


```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Books you should read!




Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin



zedr/clean-code-python: :bathtu

github.com/zedr/clean-code-python

IE | Google Keep | AssetsUL | Practical Deep Lear... | Catalog | Google Cl... | Kearney | Databricks | Iberia KA Team Issu... | Enterprise Skills Init... | Other bookmarks

README.md

clean-code-python

build passing | python 3.8+

Table of Contents

Introduction

Software engineering principles, from Robert C. Martin's book *Clean Code* , adapted for Python. This is not a style guide. It's a guide to producing readable, reusable, and refactorable software in Python.

Packages

No packages published

Contributors 27

+ 16 contributors

Languages

● Python 98.7% ● Makefile 1.3%

https://github.com/zedr/clean-code-python.git

Hands on exercise

Jupyter notebooks limit reproducibility, scalability, and automation, making them unsuitable for production-grade MLOps workflows

Hidden state and execution order
Notebooks remember past runs, so results change if cells run out of order or after edits

Lack of modular structure
Code is rarely split into functions, making it hard to reuse, test, or scale for production

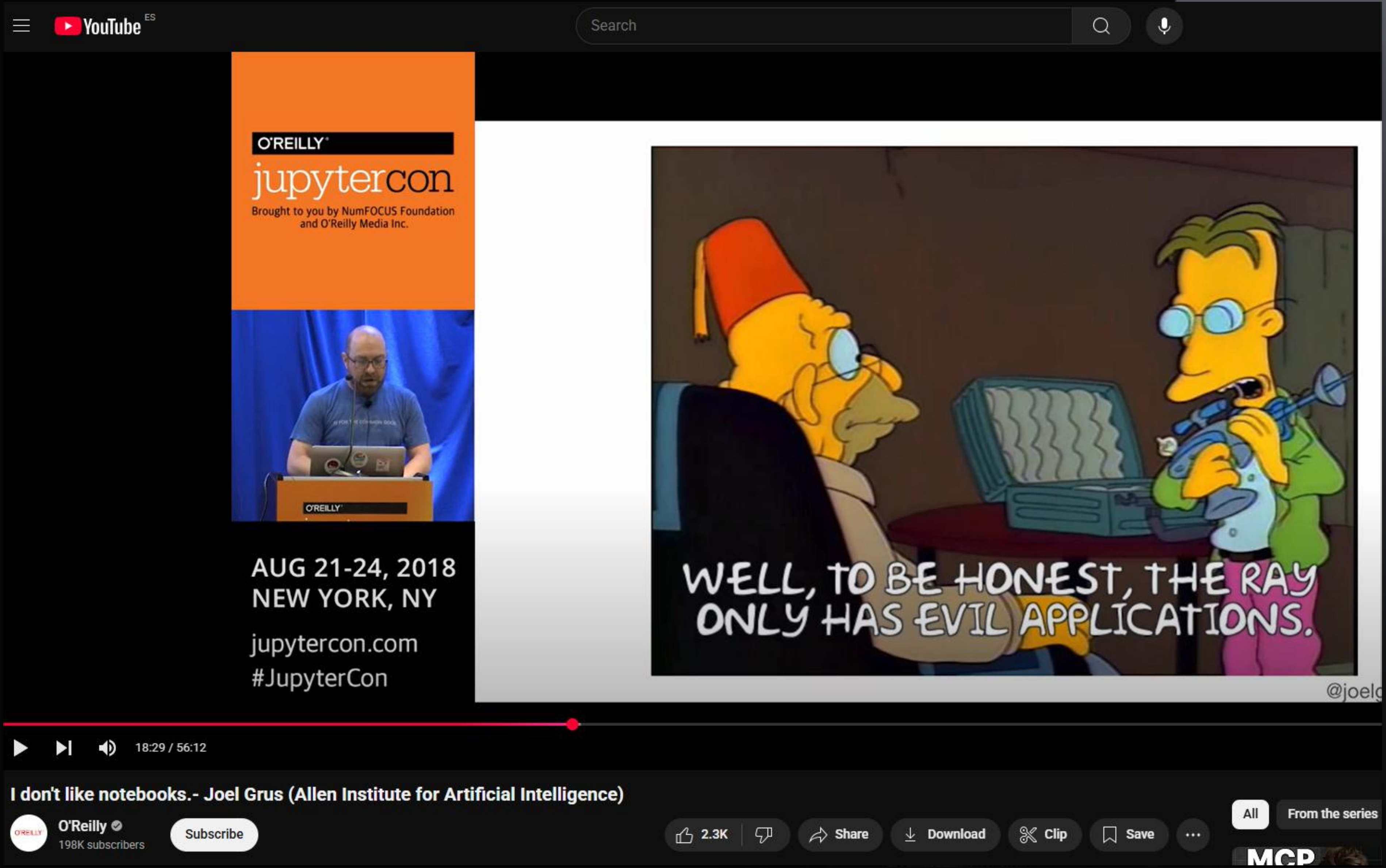
Hard-coded parameters and configuration
Values like file paths or settings are written directly in cells, reducing flexibility and automation

Limited automated testing
Notebooks aren't designed for unit tests, so errors often go unnoticed until late in the process

Difficult version control and collaboration
Notebook files are hard to compare and merge in Git, making teamwork and reviews frustrating

Embedded secrets and security risks
It's common to paste passwords or keys into cells, which can accidentally leak in shared files

ie
UNIVERSITY



I don't like notebooks.- Joel Grus (Allen Institute for Artificial Intelligence)

O'Reilly
198K subscribers
Subscribe

Refactor notebooks into modular, testable code with config separation, logging, and CLI readiness to enable production integration

General best practices

1. **Modularization**: Break down code into smaller, reusable functions or classes
2. **Configuration Management**: Use configuration files or environment variables for settings
3. **Logging and Monitoring**: Add logging and possibly some basic monitoring metrics
4. **Error Handling**: Include comprehensive error handling and data validation
5. **Dependency Management**: List all dependencies and versions for reproducibility
6. **Documentation**: Include sufficient comments and documentation for maintainability
7. **Code Quality**: Ensure the code is clean, readable, and follows PEP 8 standards
8. **Testing**: Include unit tests to validate the functionality
9. **Security**: Remove any sensitive or secret info (.env)

ML Pipeline specific

1. **data_loader.py**: Load data from CSV, databases, or APIs
2. **data_validation.py**: Validate schema, check types, handle missing data
3. **preprocessing.py**: Clean, scale, encode, and transform input data
4. **features.py**: Create and select engineered features
5. **model.py**: Define, train, save, and load ML models
6. **evaluation.py**: Compute and log performance metrics
7. **inference.py:** Apply trained model to new data (predict pipeline)
8. **config.py**: Centralize parameters, paths, and environment configs
9. **utils.py** (optional): Shared utilities (e.g. logging setup, plotting functions)

Steps – Setting up the working environment

1. **Create new conda environment from yml file**
   - conda env create -f environment.yml
   - conda activate <env_name>
2. **Create new project structure (cookiecutter)**
   - cookiecutter -c v1 https://github.com/drivendata/cookiecutter-data-science
3. **Create a GitHub repo**
   - **Locally**
     - git init
     - git add .
     - git commit -m "Initial commit“
   - **On GitHub**
     - ‘New’
     - ‘Create repository’
     - Copy URL
   - **Locally**
     - git remote add origin https://github.com/username/repository.git
     - git branch -M main
     - git push -u origin main

Steps – Refactoring the Jupyter Notebook

4. **Move the Jupyter Notebook to /notebooks**
5. **Run and Check Jupyter Notebook**
6. **Translating the . Ipynb to .py**
   - jupytext --to py notebook.ipynb
   - Pairing Notebooks (optional)
     - jupytext --set-formats ipynb,py notebook.ipynb
7. **Checking formatting standards**
   - Pylint and autopep8 installed on VSC
   - We'll use Copilot for refactoring and code optimization (be careful on what it does to your code!)
8. **Create individual modules for each section (ML step)**
   1. Checking formatting standards
   2. Add logging
   3. Error Handling / Try - Exception
   4. Maintain dependencies updated
   5. Include sufficient comments and documentation
   6. Code Quality: Ensure the code is clean, readable, and follows PEP 8 standards
9. **Generate test suit for each function** (Don not move forward until your module passes your tests)

Project Rubric and expectations (To be updated)

Code Quality

- The code is following the PEP 8 coding style (PEP8 => 8)
  e.g. Meaningful names and syntax; properly commented and formatted; Imports are ordered, etc.
- There's a README.rm providing a general (preliminary) overview of the project, instructions on how to use the code, run tests, etc. [Anyone should be able to run the code from those instructions]
- All functions and files have docstrings, and commenting is used appropierly i.e. Why NOT What

Testing

- Tests cover all functions and relevant artifacts e.g. Inputs are in the expected file format? Outputs are saved in the right places?

Logging

- Each function is complete with logging (INFO or otherwise), and Logs are stored in a .log file (Distinguishing between logs relevant to keep track of, and those to be shown on the terminal)

Config mgmt.

- A config.yml (python_env.yml) file(s) is used for Configuration Management and environment variables for settings (Avoid hardcoding variables)

Dependencies

- Dependencies are managed through a conda.yml file, listing all dependencies and versions for reproducibility

Error handling

- Error Handling thru Try/ Except provides a comprehensive error handling and data validation

Artifacting

- Relevant reports, notebooks, or images of plots from the EDA phase are stored on a respective folder
- A model(s) is properly stored (joblib or pickle) fr future production use in inference phases

- The code components cover and end-to-end ML process e.g.
  1. Data Loading: Create a function to load the data.
  2. Data Exploration: Modularize the code used for data exploration into reusable functions.
  3. Data Preprocessing: Create functions for preprocessing steps like handling missing values, scaling, etc.
  4. Feature Engineering: If applicable, create functions for feature engineering.
  5. Model Training: Create functions for training machine learning models.
  6. Evaluation: Create functions for model evaluation.
  7. Utility Functions: Create utility functions for common operations like plotting.

Version Controlling

- GitHub is used for Versoin Control and development, demosntrating:
  - Proeficiency with basic operations e.g. add, commit, push, etc.
  - Follow best practices (as discussed in class) for:
  1. Project creation
  2. Branching (Check out features, debugging, releases, development vs. main)
  3. Pulls Request reviewing (Open, review and merge)
  4. Taggin and releasing (version control and release)
  5. Collaboration & Communication (Discussions and resolutions)