ML END2END PIPELINE

TIPS FROM REAL-WORLD PROJECTS

INTRO / PURPOSE

An end-to-end machine learning project aims to take a model from idea to real-world use, including data pipelines, deployment, monitoring, and updates.

A static project in a notebook usually stops at training and testing the model, without worrying about how it runs in production or keeps working over time.

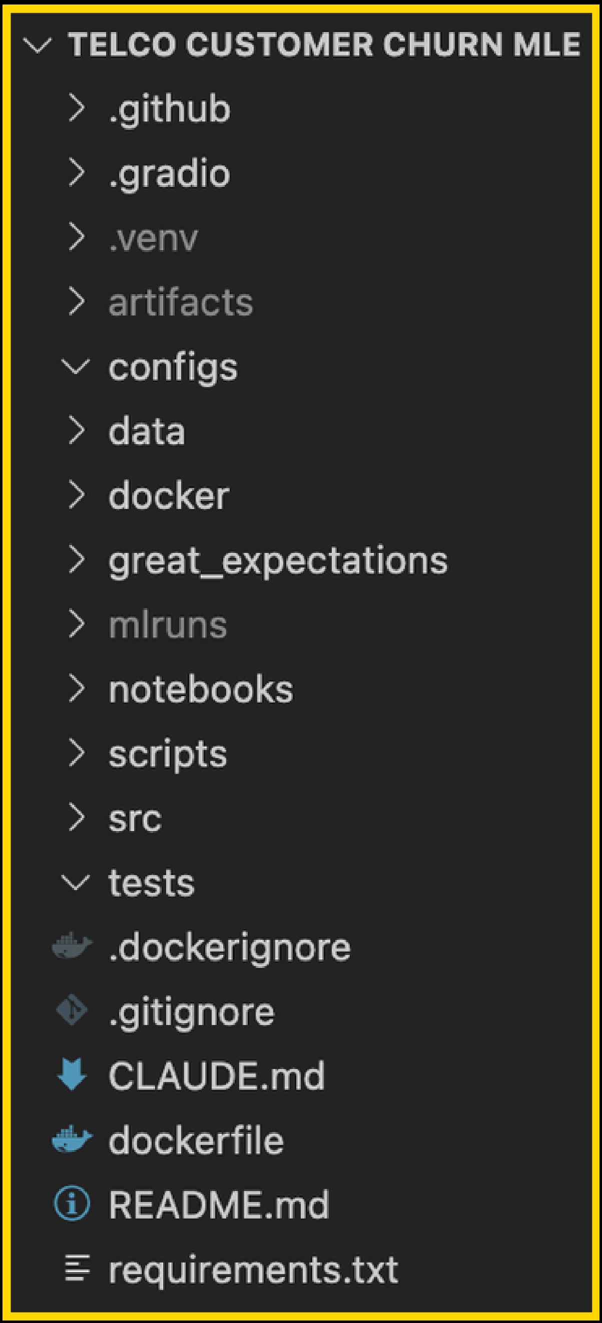
The end-to-end approach makes the work useful outside the notebook.

TABLE OF CONTENT

1/ SET ENVIRONMENT

A PROJECT STRUCTURE

It’s important to set a project structure before starting your project. Some components appear in most ML project, make sure to include them.



```
TELCO CUSTOMER CHURN MLE
  > .github
  > .gradio
  > .venv
  > artifacts
  v configs
  > data
  > docker
  > great_expectations
  > mlruns
  > notebooks
  > scripts
  > src
  v tests
    .dockerignore
    .gitignore
    CLAUDE.md
    dockerfile
    README.md
    requirements.txt
```

```
mkdir -p \
  data/raw data/processed data/external \
  notebooks \
  src/{data,features,models,utils} \
  app \
  configs \
  scripts \
  tests \
  .github/workflows \
  docker \
  great_expectations \
  mlruns \
  artifacts
```

Run this command in project’s terminal to create all the structure for you.

Why? *Because ‘most’ ML projects have a similar structure so no need to keep doing the same thing over and over*

1/ SET ENVIRONMENT

B INSTALL LIBRARIES AND CREATE VIRTUAL ENVIRONMENT

Run this command in project's terminal to create requirements.txt. Include all needed libraries.

```
cat > requirements.txt << 'EOF'
pandas
numpy
scikit-learn
mlflow
fastapi
uvicorn
pydantic
python-dotenv
joblib
great-expectations
pytest
EOF
```

Run this command in project's terminal to create virtual environment and install libraries using 'uv'. uv in this case speeds up the installation.

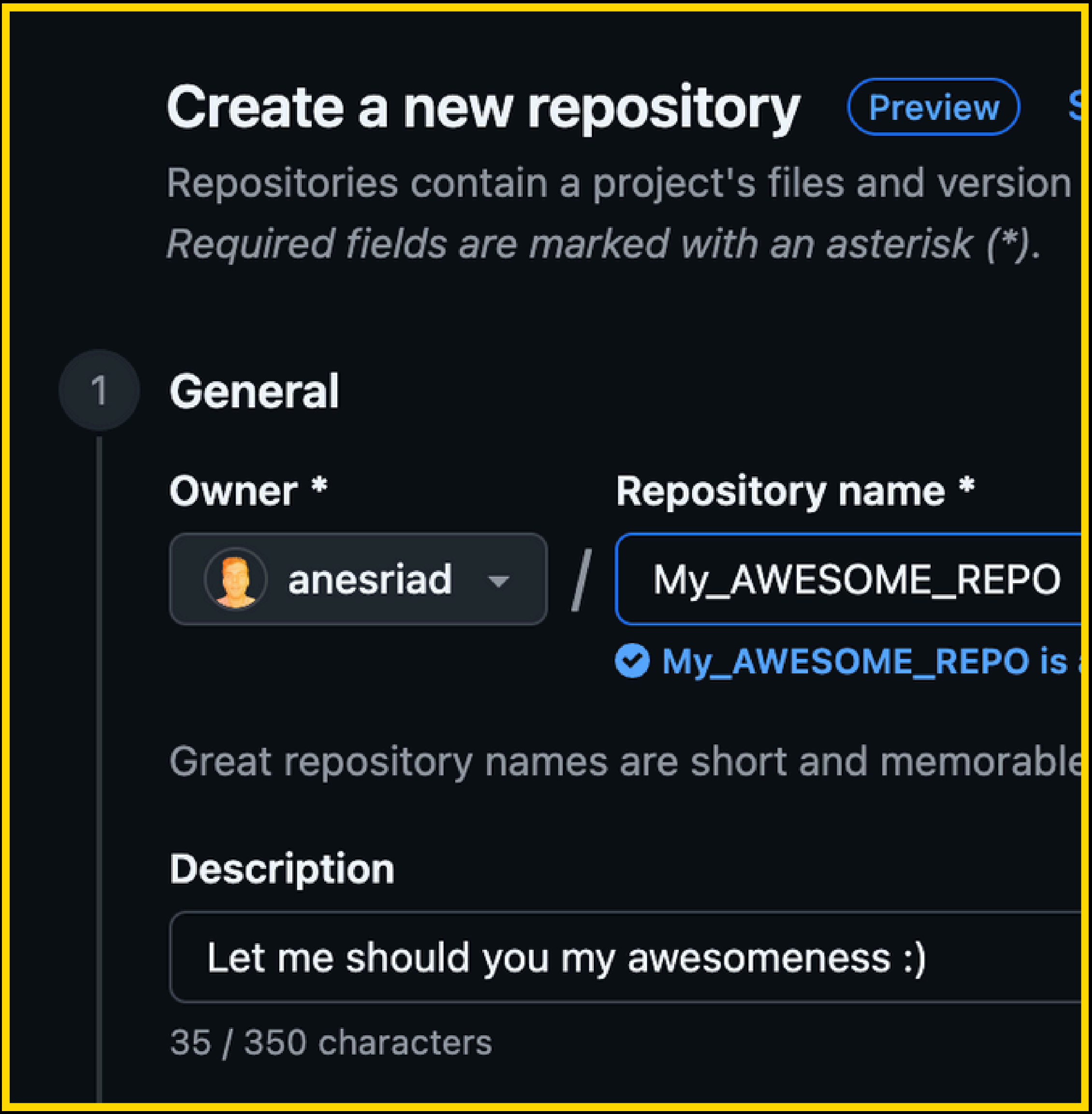
```
python3.11 -m venv .venv
source .venv/bin/activate
pip install --upgrade pip
uv pip install -r requirements.txt
```

Why? *Each project needs a dedicated environment to avoid dependency conflicts.*

1/ SET ENVIRONMENT

C SET GIT & GITHUB

Go to GitHub.com, sign in to your account, and create a new repository.



Create a new repository Preview

Repositories contain a project's files and version

Required fields are marked with an asterisk (*).

1 General

Owner * / Repository name *

anesriad / My_AWESOME_REPO

My_AWESOME_REPO is

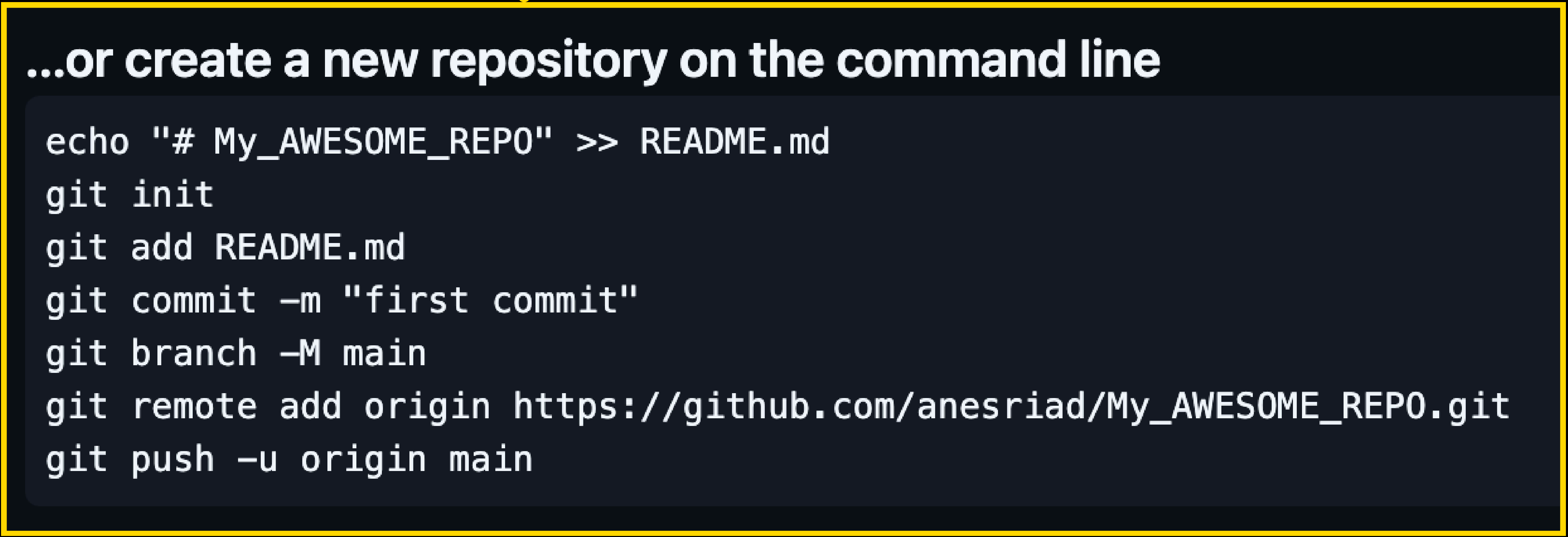
Great repository names are short and memorable

Description

Let me should you my awesomeness :)

35 / 350 characters

Run these commands in project's terminal to initialise GIT, add a Readme file, commit your first change, and link your project to GitHub



...or create a new repository on the command line

```
echo "# My_AWESOME_REPO" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/anesriad/My_AWESOME_REPO.git
git push -u origin main
```

Why? *Each project needs a dedicated environment to avoid dependency conflicts.*

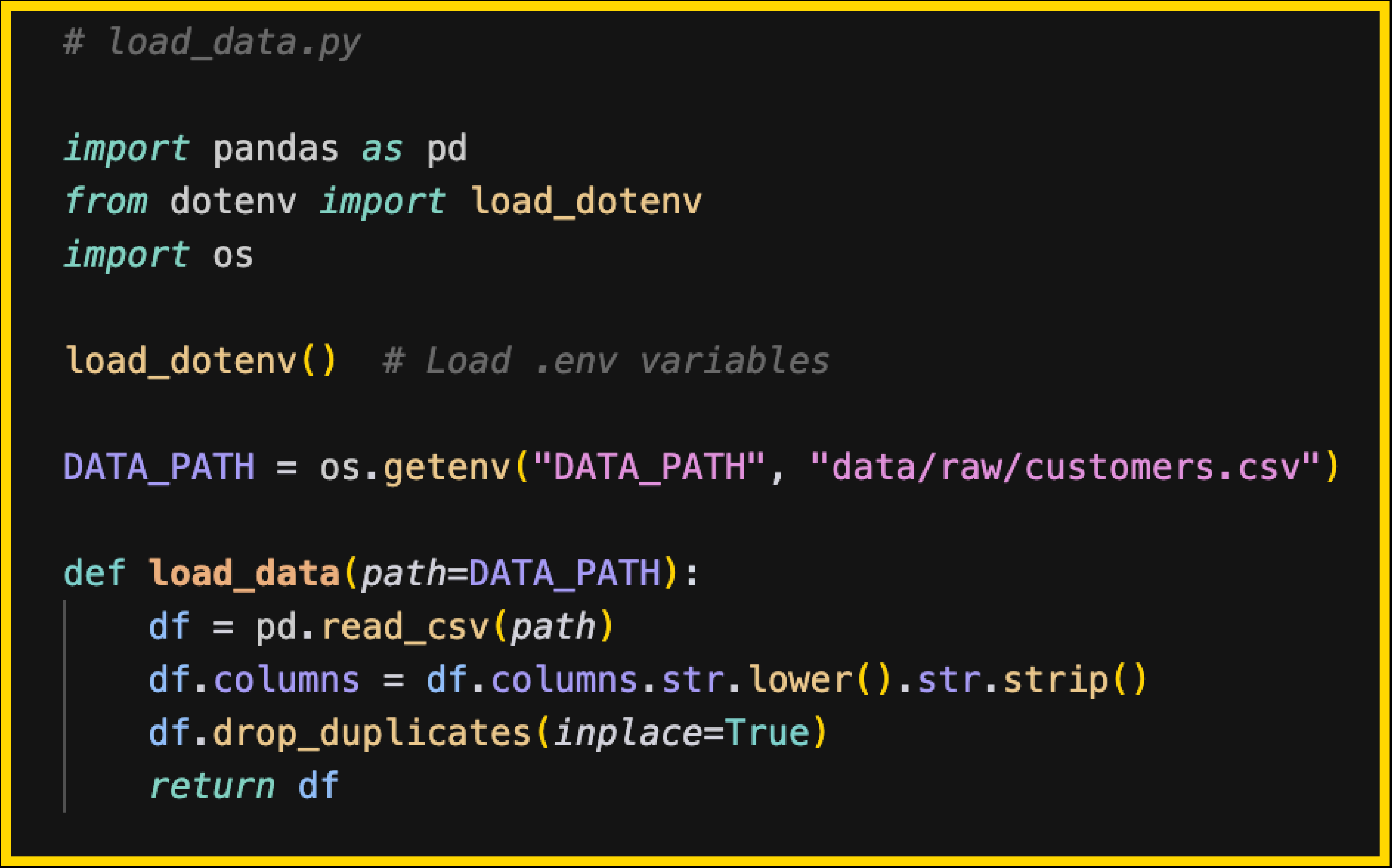
2/ DATA INGESTION

Before you analyze or model anything, you need to load your data reliably and reproducibly.

Where data comes from
- CSVs, SQL/NoSQL databases, APIs, cloud storage

Light preprocessing at this stage
- Rename columns, parse dates, drop duplicates
- Apply consistent schemas or dtypes early

Reproducibility
- Use loading scripts, not manual steps
- Version input data if possible



```python
# load_data.py

import pandas as pd
from dotenv import load_dotenv
import os

load_dotenv()  # Load .env variables

DATA_PATH = os.getenv("DATA_PATH", "data/raw/customers.csv")

def load_data(path=DATA_PATH):
    df = pd.read_csv(path)
    df.columns = df.columns.str.lower().str.strip()
    df.drop_duplicates(inplace=True)
    return df
```

Why? *Because broken inputs = broken pipeline. Clean, consistent ingestion is the foundation of every ML project.*

3/ DATA VALIDATION

You want to validate incoming data before engineering features or training.

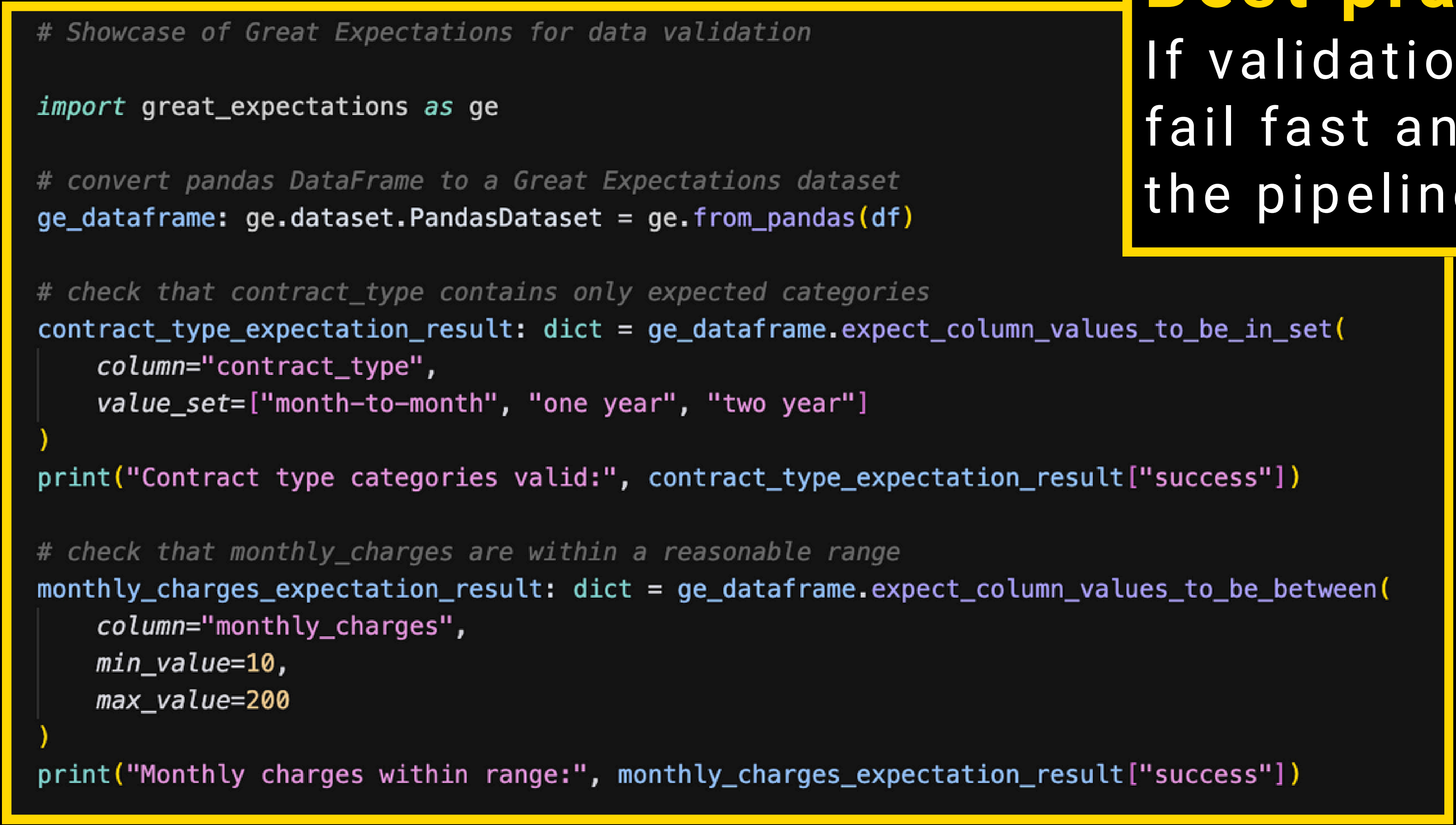
WHAT TO CHECK

- **Are column names/types correct?**
- **Are there missing or out-of-range values?**
- **Do values match expected formats?**

HOW TO VALIDATE

- **Use libraries like pydantic, Great Expectations, or pandas assertions.**
- **Run validation early so errors don't silently break your model later.**

Best practice
If validation fails, fail fast and stop the pipeline.



```python
# Showcase of Great Expectations for data validation

import great_expectations as ge

# convert pandas DataFrame to a Great Expectations dataset
ge_dataframe: ge.dataset.PandasDataset = ge.from_pandas(df)

# check that contract_type contains only expected categories
contract_type_expectation_result: dict = ge_dataframe.expect_column_values_to_be_in_set(
    column="contract_type",
    value_set=["month-to-month", "one year", "two year"]
)
print("Contract type categories valid:", contract_type_expectation_result["success"])

# check that monthly_charges are within a reasonable range
monthly_charges_expectation_result: dict = ge_dataframe.expect_column_values_to_be_between(
    column="monthly_charges",
    min_value=10,
    max_value=200
)
print("Monthly charges within range:", monthly_charges_expectation_result["success"])
```

Why? *Bad data silently breaks your pipeline. Validation catches issues early, before they corrupt training or hit production.*

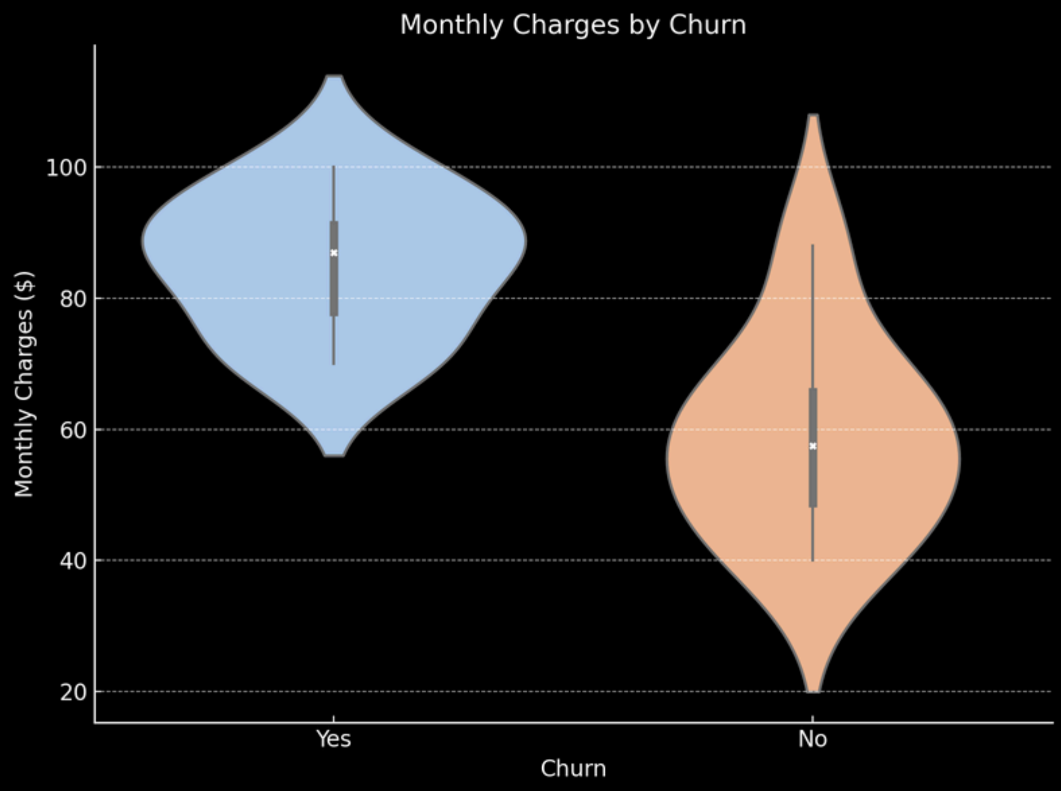
4/ EXPLORATORY DATA ANALYTICS

A WHY EDA?

UNDERSTAND • SPOT PROBLEMS • FORM HYPOTHESES

EDA isn't just "looking at data."
It's how you:

- Understand distributions and relationships
- Spot missing values and outliers
- Catch broken logic in your dataset
- Shape the path for modeling



TYPES OF EDA

UNIVARIATE

→ Understand distributions and relationships

BIVARIATE

→ Scatter plots to spot relationships

MULTIVARIATE

→ Correlation heatmaps for variable interactions

TIME SERIES

→ Line plots to detect trends and seasonality

Start simple, then go deeper as patterns emerge.

Why? *EDA helps you understand your data, spot issues early, and decide what to do next.*

4/ EXPLORATORY DATA ANALYTICS

B STRUCTURED EDA IN PRACTICE

3 steps to make your EDA faster, cleaner, and actually useful

1/ Dataset Overview

Understand your data before diving in (before your first plot).

- **Rows & columns:** Do you have 10K rows or 10M? It changes everything.
- **Data types:** Explicitly verify int, float, object, bool, datetime.
- **Missingness & Duplicates:** What % is missing? Are duplicates valid?
- **Target variable distribution:** Look for imbalance early.

2/ Feature Deep-Dive

Now zoom in. Focus on distributions and transformations.

- **Numerical:** Outliers? Skewed? Is normalization needed?
- **Categorical:** Too many levels? Are rare classes meaningful or noise?
- **Zero-heavy fields:** Are they real zeros, or just unused? (e.g. income = 0)
- **Leakage risk:** Any feature too good to be true?
- **Correlations:** Find redundancies before modeling.

3/ Quality Traps & Broken Logic

EDA is debugging for your data, not just describing it.

- Inconsistent records: Conflicting info across rows?
- Invalid categories: Typos, inconsistent casing, mixed encodings.
- Linked nulls: Are two features always missing together?
- Suspicious values: Negative ages? Future timestamps?

In churn prediction, EDA might reveal patterns like how “last login” or “support tickets opened” vary between retained vs churned users

---

Why? *A structured flow makes your EDA faster, repeatable, and easier to debug.*

5/ FEATURE ENGINEERING

Transform raw data into signals your model can learn from

TYPICAL STEPS

- Handle missing values
- Encode categoricals
- Normalize numeric features
- *Create new features from domain logic*
- *Aggregate or transform time-based features*
- *Drop redundant or leaking features*

FEATURE ENGINEERING

Numerical Features
→ Normalize values
→ Bin into ranges (e.g., low/med/high usage)
→ Create ratios or differences between columns

Categorical Features
→ Encode with one-hot or label encoding
→ Group rare categories
→ Map to risk levels if applicable

Date Features
→ Extract durations (e.g., account age)
→ Calculate time since events
→ Flag recent activity

Text Features
→ Count keyword mentions
→ Detect sentiment or intent
→ Encode using embeddings or TF-IDF

Interaction Features
→ Combine fields to reveal patterns
→ Multiply related columns (e.g., usage × cost)
→ Use domain logic to create new variables

Avoid Data Leakage
Only use info available at prediction time.
No future payments, no labels, no outcomes.

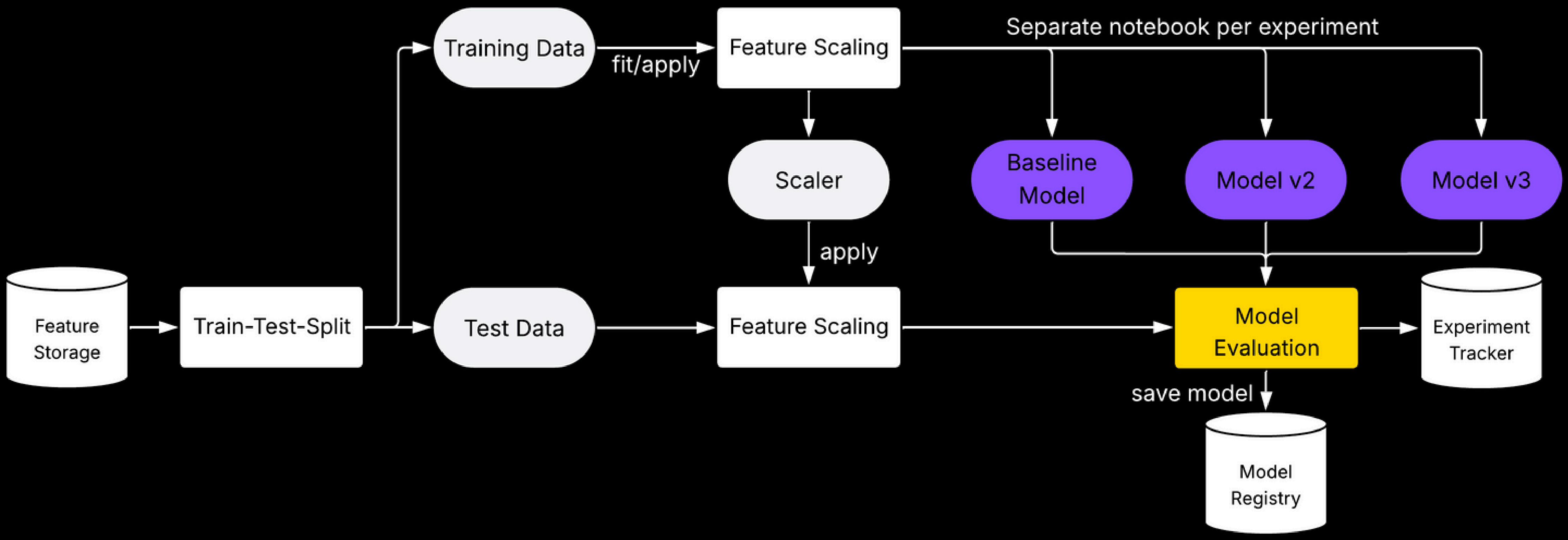
Why? *Smart features make patterns easier to learn and often have the biggest impact on model performance.*

6/ MODEL TRAINING

Model training means teaching your algorithm to generalize from past data to unseen data.

How it works:

- Split data into training + test sets
- Fit scalers on training data only
- Train a baseline model (e.g., logistic regression)
- Try new architectures or hyperparams
- Log results to track what works
- Evaluate and version your best model



Example (Churn):
You might start with logistic regression, then try XGBoost or LightGBM as iterations. Track every result in your experiment tracker (e.g., MLflow)

Why? *Training helps your model learn patterns, tracking helps you learn which patterns worked best.*

7/ EVALUATION

MEASURE WHAT MATTERS

Evaluation tells you how well your model performs and whether it’s ready for production.

What to evaluate:

- Accuracy or F1 → for classification (like churn prediction)
- MAE / RMSE → for regression
- AUC → when class imbalance matters
- Confusion matrix → for error analysis
- Calibration → if probabilities are used in decisions

Tips

- Always compare on the same test set
- Track metrics across all experiments
- Use plots (ROC, PR curve, residuals) to dig deeper
- Go beyond averages: inspect where your model fails

Example (Churn):
Your model might have 85% accuracy, but most of that comes from predicting "no churn". F1, F2 and confusion matrix will give a clearer picture.

What to log:

- Metrics
- Test dataset version
- Model version
- Dataset Hash
- Hyperparameters used

Why? *Evaluation reveals how your model performs, and where it breaks.*

7/ MODEL REGISTRY & EXPERIMENT TRACKING

TRACKING EXPERIMENTS

helps you avoid "notebook chaos" and compare what's actually working. A model registry makes your deployment traceable and versioned, like **Git for models.**

WHAT TO TRACK:

- Parameters (e.g., learning rate, max_depth)
- Metrics (e.g., F1, ROC-AUC)
- Artifacts (e.g., trained model files, plots)
- Code version & dataset snapshot

HOW TO USE IT:

- Use **MLflow** to log training runs with mlflow.log_params() and mlflow.log_metrics()
- Register the best-performing model in the Model Registry
- Deploy only models that passed evaluation + review

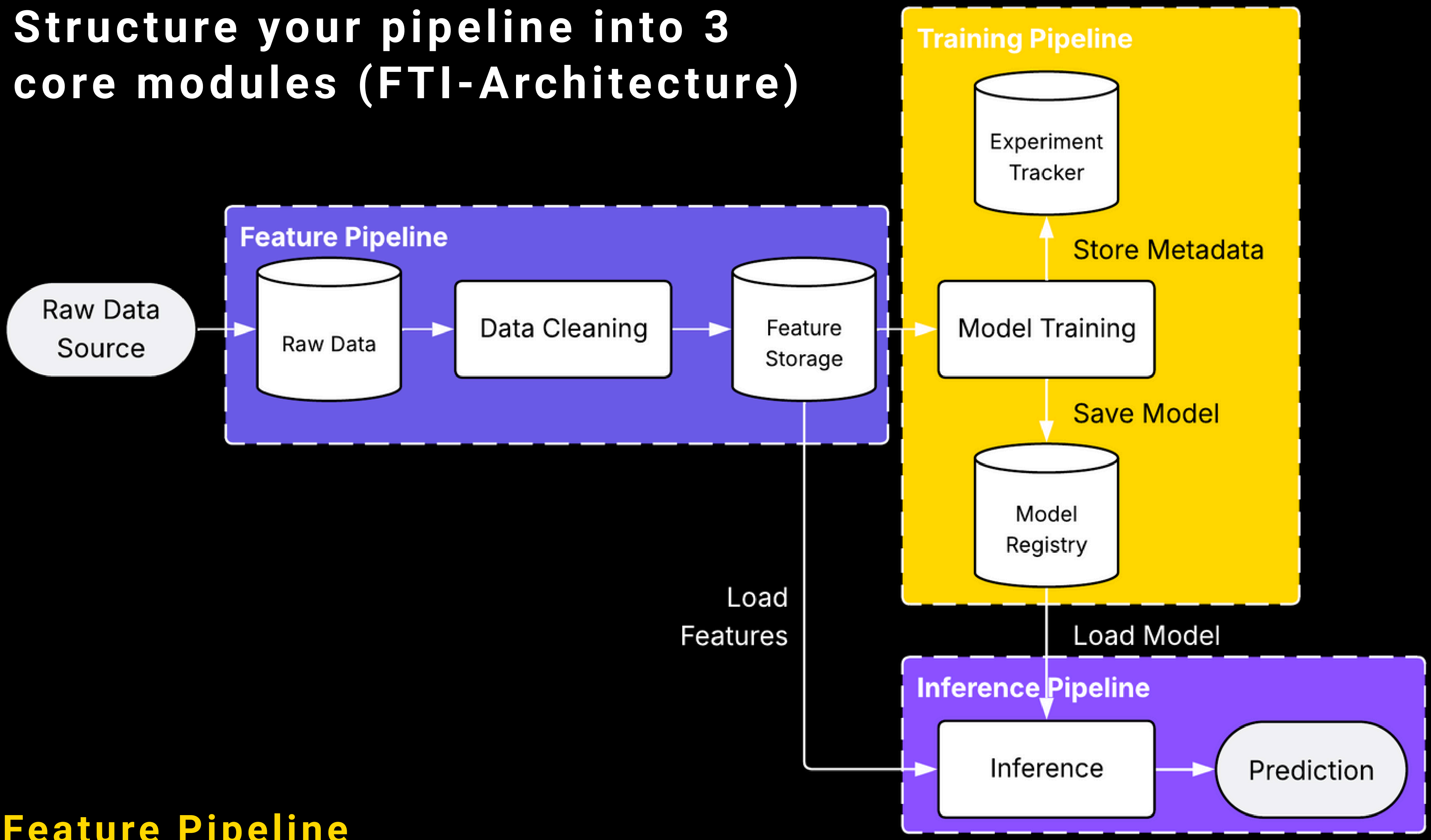
Tip: *Tag each model with metadata like production_ready=True, so it's clear what's deployable.*

Why? *Evaluation reveals how your model performs, and where it breaks.*

8/ MODULARIZE LOGIC

Tangled pipelines break fast.

Structure your pipeline into 3 core modules (FTI-Architecture)



Feature Pipeline

- Prepares reusable features for both training and inference.
- Ingestion: Handle raw source data & schemas
- Preprocessing: Clean, normalize, encode
- Feature Engineering: Transformations split by data type or domain

→ Output: Logged to a feature store (optional)

Training Pipeline

- Focuses on model development, reproducibility, and evaluation.
- Training: Model selection, hyperparams, config mgmt
- Evaluation: Metrics, diagnostics, drift checks

→ Output: Saved model & metadata

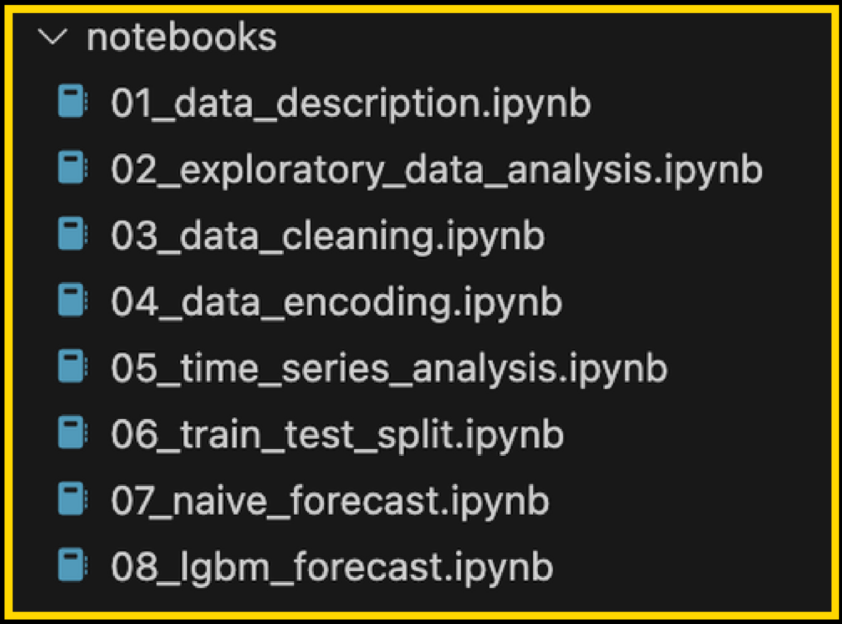
Inference Pipeline

- **S**erves predictions in production environments.
- Feature Retrieval: Pull latest features or recompute
- Serving: Wrap model with APIs, format outputs

→ Output: Scored results, downstream integrations

Why? *Modular pipelines make your ML system easier to scale, test, and maintain.*

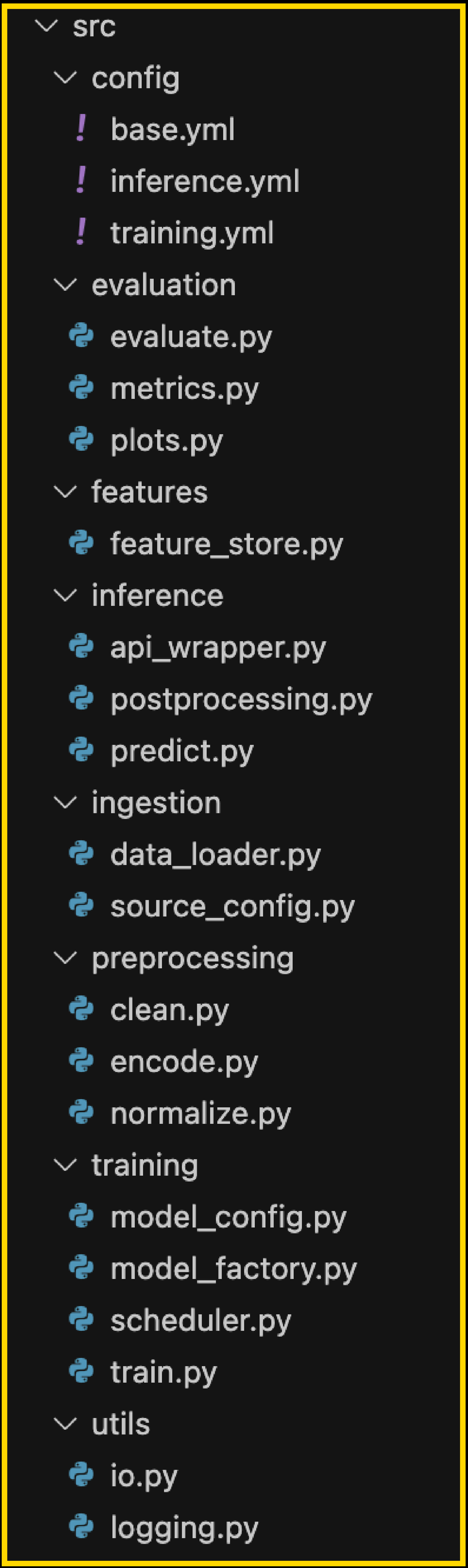
9/ MODULARIZE LOGIC



```
∨ notebooks
    01_data_description.ipynb
    02_exploratory_data_analysis.ipynb
    03_data_cleaning.ipynb
    04_data_encoding.ipynb
    05_time_series_analysis.ipynb
    06_train_test_split.ipynb
    07_naive_forecast.ipynb
    08_lgbm_forecast.ipynb
```

Core Logic

*separate notebooks for each processing step and experiment



```
∨ src
  ∨ config
     ! base.yml
     ! inference.yml
     ! training.yml
  ∨ evaluation
      evaluate.py
      metrics.py
      plots.py
  ∨ features
      feature_store.py
  ∨ inference
      api_wrapper.py
      postprocessing.py
      predict.py
  ∨ ingestion
      data_loader.py
      source_config.py
  ∨ preprocessing
      clean.py
      encode.py
      normalize.py
  ∨ training
      model_config.py
      model_factory.py
      scheduler.py
      train.py
  ∨ utils
      io.py
      logging.py
```

base.yml	Shared config defaults
inference.yml	Service-time behavior
training.yml	Model + training setup
evaluate.py	Run eval pipeline
metrics.py	Custom scoring logic
plots.py	Model diagnostics
feature_store.py	Store and load features
api_wrapper.py	Serve model via API
postprocessing.py	Tidy outputs for users
predict.py	Run model inference
data_loader.py	Pull and load source data
source_config.py	Raw data setup
clean.py	Fix raw data issues
encode.py	Encode categoricals
normalize.py	Scale numeric features
model_config.py	Set hyperparameters & options
model_factory.py	Build models from config
scheduler.py	Trigger + manage training
train.py	Run model training
io.py	Read/write data and models
logging.py	Central logging logic

Keep IO, logic, and config separate. Treat each stage of the architecture as a replaceable unit, not a one-off script.

Why? *Modular code lets you reuse parts, test in isolation, and swap logic without breaking the whole.*

10/ STARTER WORKFLOWS BUILD REAL SKILL

TRY THESE MINI WORKFOWS:

1.Notebook → Scripted Pipeline

Start with a notebook → extract code into preprocess.py, train.py, predict.py → run with main.py or CLI
Mental bridge from one-off notebook to repeatable projects

2. Train → Save → Predict

Train model → save with joblib.dump() → load later with joblib.load() in new script

Introduces serialization and persistance, a must-know before deployment

3. FastAPI Wrapper (Single File)

Load a model in FastAPI → define /predict endpoint → accept JSON → return predictions

Shows how a model becomes a real service

4. GitHub + CI Test

Push to GitHub → GitHub Actions runs test: e.g. "python predict.py works"

Easy intro to CI/CD; teaches that your code should always be runable

5. CLI For Feature Engineering

Wrap preprocessing into CLI script:

- *"python build_features.py --input raw.csv --output features.csv"*

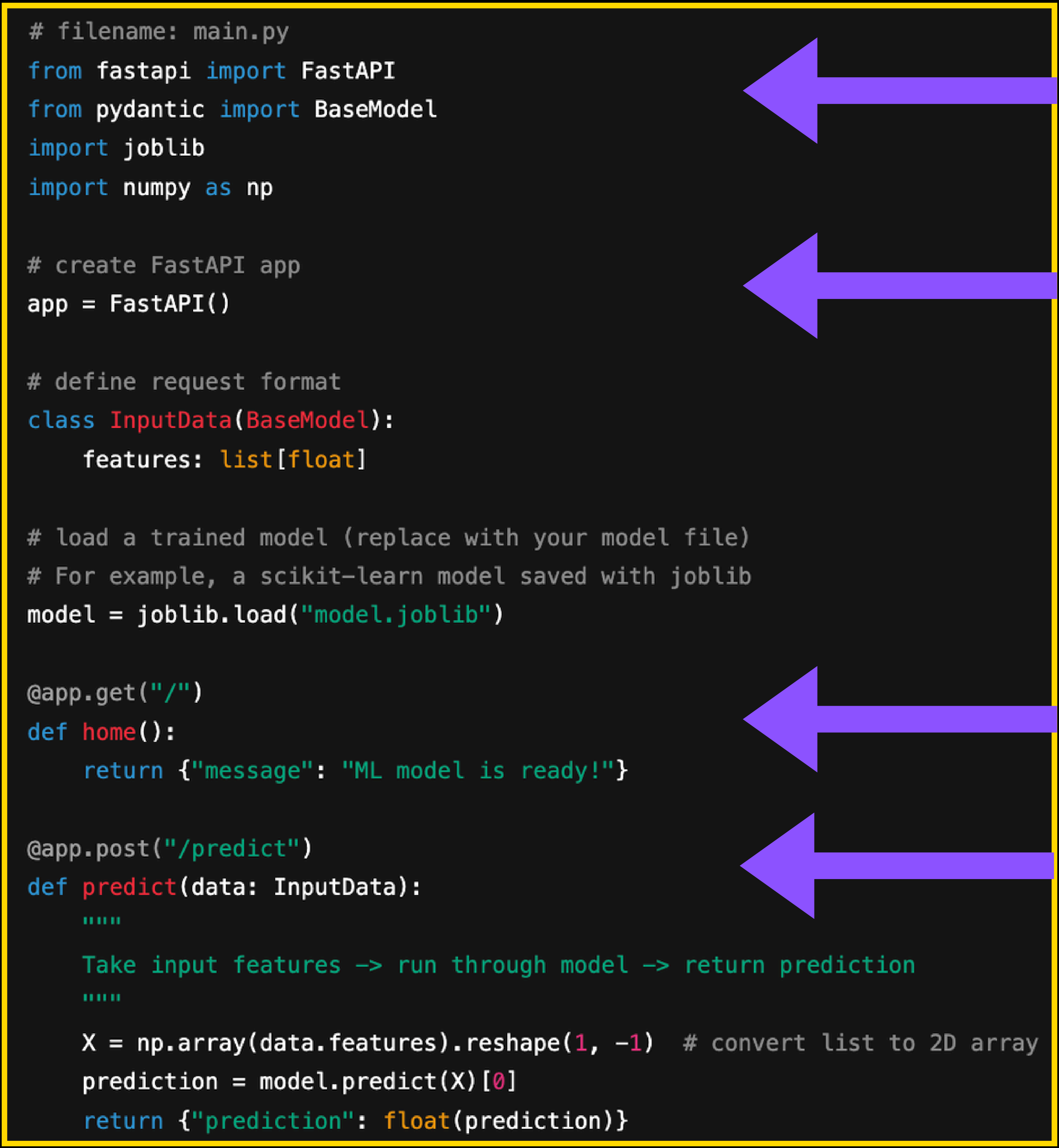
Promotes modularity and real-world thinking (users, parameters, file paths).

Why? These workflows train you to move beyond notebooks and think like a system builder

Why? *Pushes learners to structure and parameterize logic — a big skill jump.*

10/ FASTAPI ENDPOINT

FastAPI acts as the **serving layer**, it turns our ML model into an API that can **receive data** and **return predictions.**
It's necessary because without it, the model would just sit in Python files, and no external app, script, or user could interact with it.



```python
# filename: main.py
from fastapi import FastAPI
from pydantic import BaseModel
import joblib
import numpy as np

# create FastAPI app
app = FastAPI()

# define request format
class InputData(BaseModel):
    features: list[float]

# load a trained model (replace with your model file)
# For example, a scikit-learn model saved with joblib
model = joblib.load("model.joblib")

@app.get("/")
def home():
    return {"message": "ML model is ready!"}

@app.post("/predict")
def predict(data: InputData):
    """
    Take input features -> run through model -> return prediction
    """
    X = np.array(data.features).reshape(1, -1)  # convert list to 2D array
    prediction = model.predict(X)[0]
    return {"prediction": float(prediction)}
```

imports the FastAPI framework.

creates the web app object.

sets up a GET endpoint at the root URL (/).

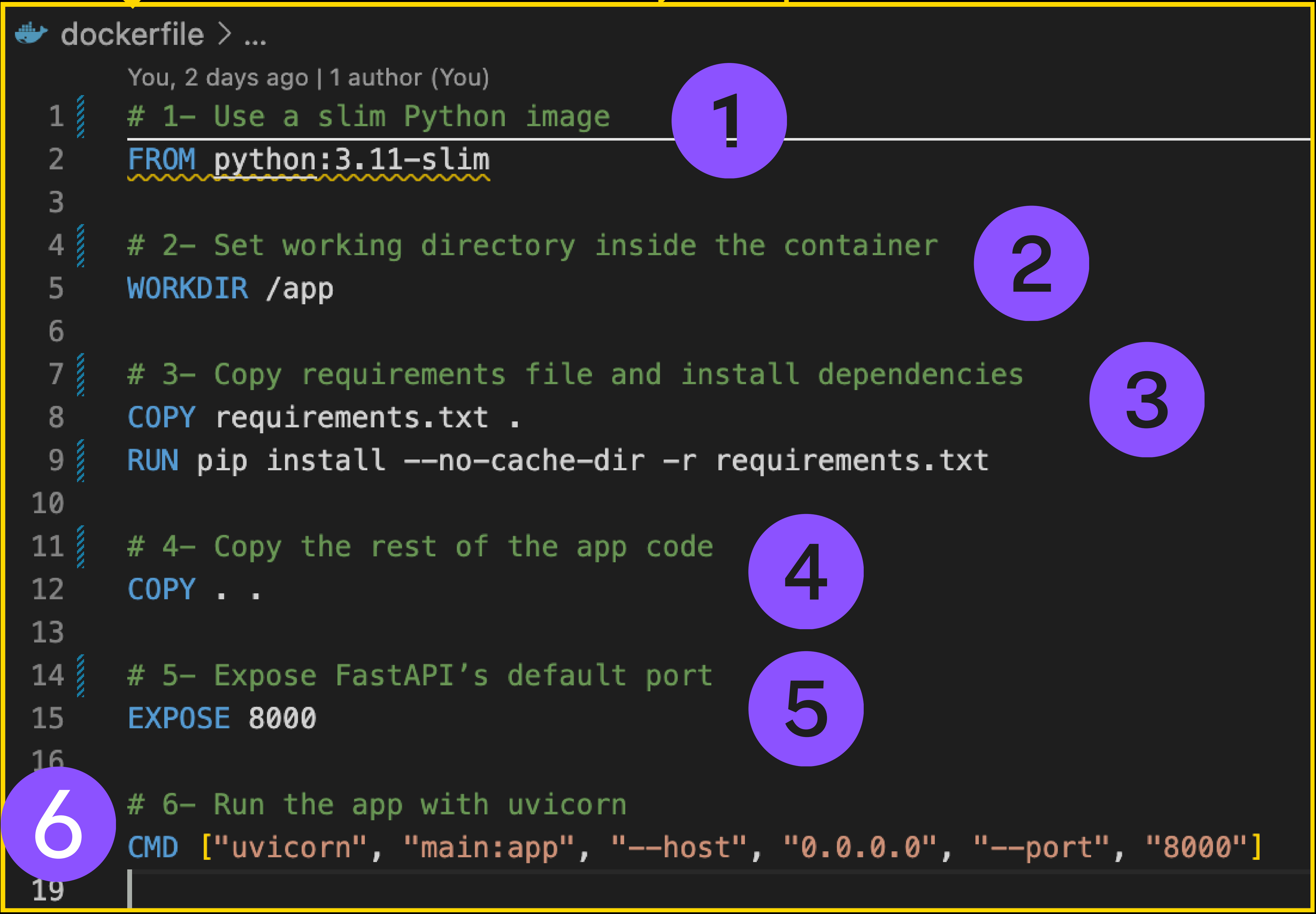
sets up a POST endpoint at /predict.

Why? *FastAPI makes it easy and fast to turn your ML model into a web API that others can call for predictions.*

11/ CONTAINERIZE (DOCKER)

Docker lets us package the whole project (code, dependencies, environment) into one container so it runs the same everywhere. The goal is for the project to run on any machine, not just yours.

create 'dockerfile' in your repo



dockerfile > ...

You, 2 days ago | 1 author (You)

```
# 1- Use a slim Python image
FROM python:3.11-slim

# 2- Set working directory inside the container
WORKDIR /app

# 3- Copy requirements file and install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 4- Copy the rest of the app code
COPY . .

# 5- Expose FastAPI's default port
EXPOSE 8000

# 6- Run the app with uvicorn
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

To start the container:
* churn-api is the name of the container to start

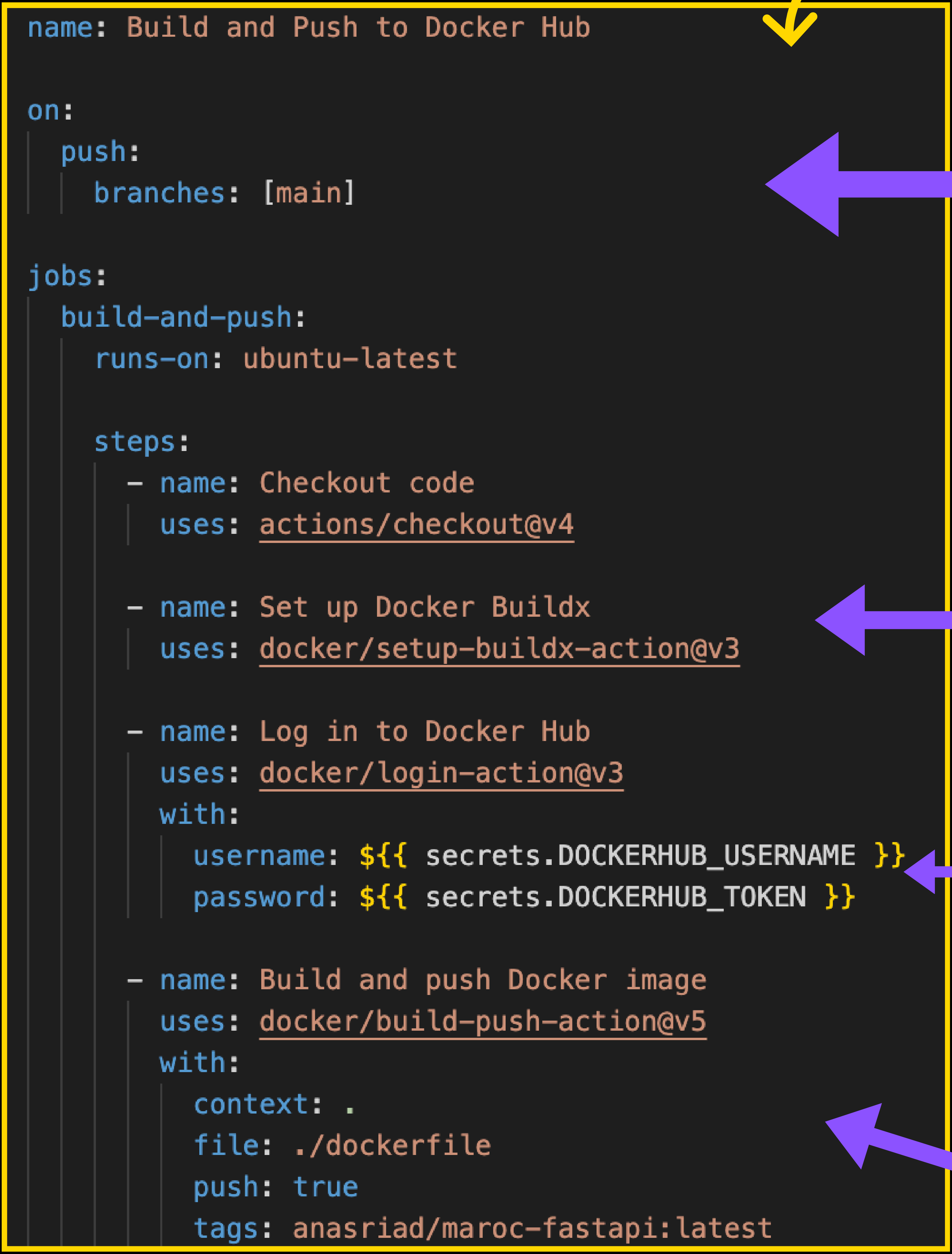
```
docker build -t churn-api .
docker run -p 8000:8000 churn-api
```

Why? *Docker makes ML projects portable and reproducible by packaging code, dependencies, and models into one container that runs anywhere.*

12/ CI/CD - GITHUB ACTIONS

A CI/CD pipeline **automates testing, building, and deployment** so code changes are checked and delivered quickly without manual steps.
GitHub Actions lets you set up this pipeline directly in your repo, running workflows (like tests or deployments) whenever you push code. Add this, when your model is stable and you are deploying regularly.

in your project create → .github/workflows/ci.yml



```yaml
name: Build and Push to Docker Hub

on:
  push:
    branches: [main]

jobs:
  build-and-push:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Log in to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Build and push Docker image
        uses: docker/build-push-action@v5
        with:
          context: .
          file: ./dockerfile
          push: true
          tags: anasriad/maroc-fastapi:latest
```

Only trigger on main branch pushes

modern Docker builder that supports multi-platform builds

Set API token username & password on Docker Hub.
Save those secrets in GitHub → Secrets & variables

Build & push docker image to docker hub

Why? *It lets you automatically build and publish your container to Docker Hub so anyone (or any server) can pull and run the latest version of your ML app easily.*

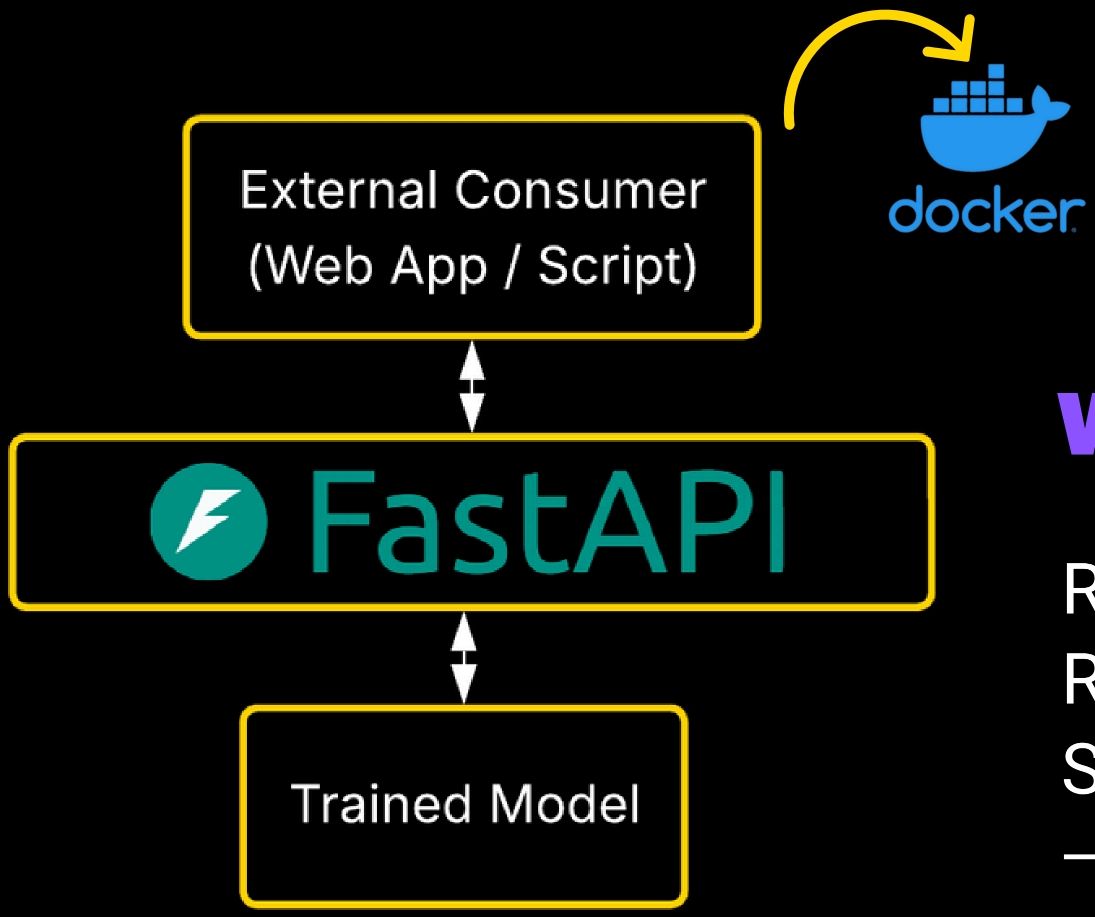
13/ DEPLOY

After training, your model is just a file. Deployment turns it into a real product that delivers value.

Monitoring ensures it keeps delivering value.

- Use FastAPI to serve predictions via a REST API
- Use .env files to separate dev/prod configs
- Docker Compose to run app + model together
- MLflow to version and track deployed models
- Run dummy requests post-deploy to catch silent failures

Return predictions as JSON, CSV, or database updates depending on consumer



wrapped in docker container, ready to be hosted

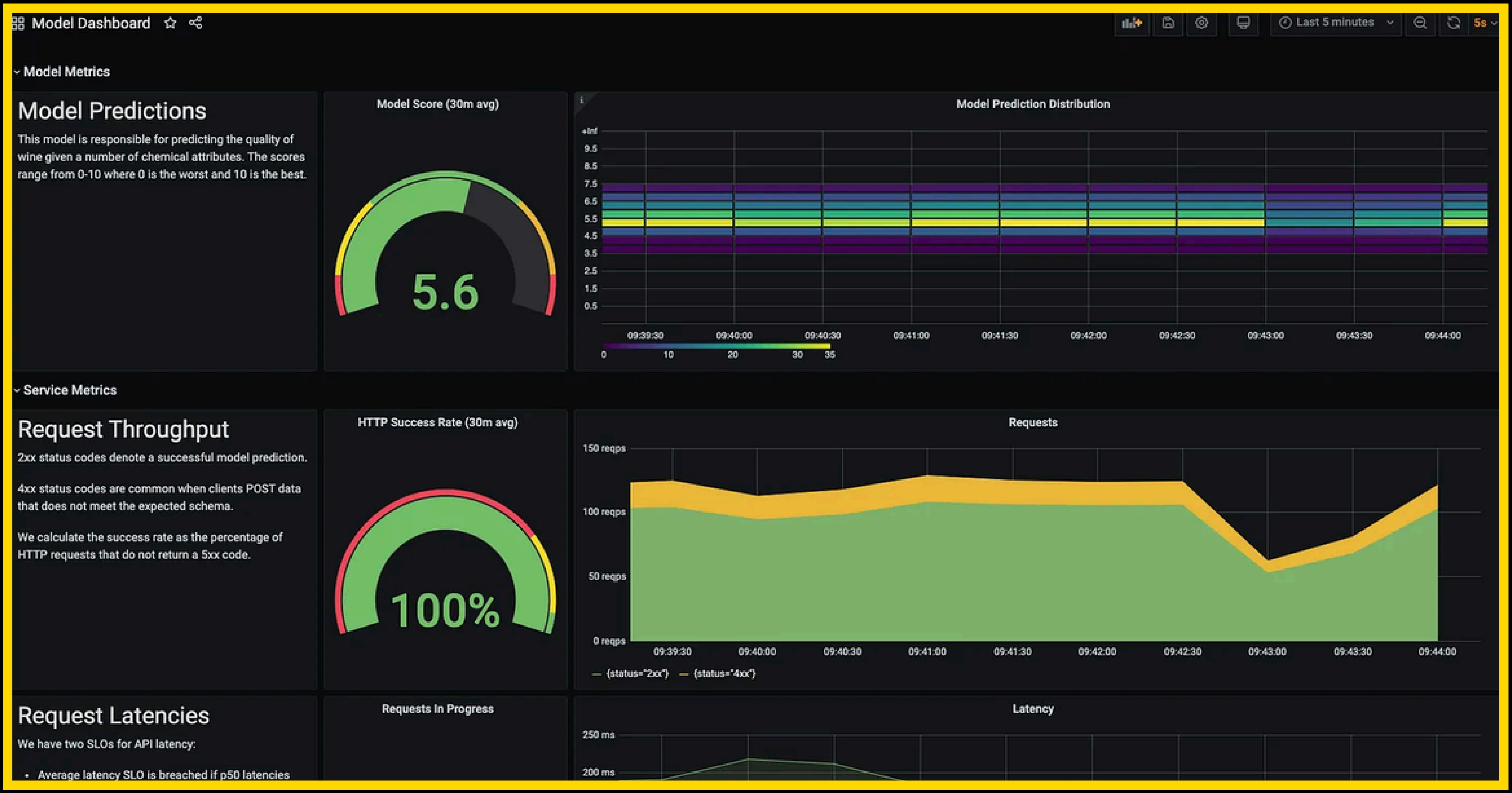
WHERE TO HOST?

Run on Vercel, Render, Railway, Hugging Face Spaces or a basic VM
→ keep it simple early on

Why? *Deployment isn't just “putting a model online”, it's making it stable, fast, and easy to use for others.*

14/ MONITOR

- **Data Drift:** Input distributions change over time. Log stats like mean/std for key features
- **Prediction quality:** Track metrics (e.g. accuracy, F1) and output distribution
- **Service health:** Uptime, latency, errors
- **Observability:** Use logging tools and set up alerts (e.g., with Prometheus + Grafana)
- **Traceability:** Store requests/responses for debugging and include inputs, model version and timestamps



Model Predictions

This model is responsible for predicting the quality of wine given a number of chemical attributes. The scores range from 0-10 where 0 is the worst and 10 is the best.

Request Throughput

2xx status codes denote a successful model prediction.

4xx status codes are common when clients POST data that does not meet the expected schema.

We calculate the success rate as the percentage of HTTP requests that do not return a 5xx code.

Request Latencies

We have two SLOs for API latency:

- Average latency SLO is breached if p50 latencies

Why? *Monitoring helps you fix silently failing systems before users notice.*

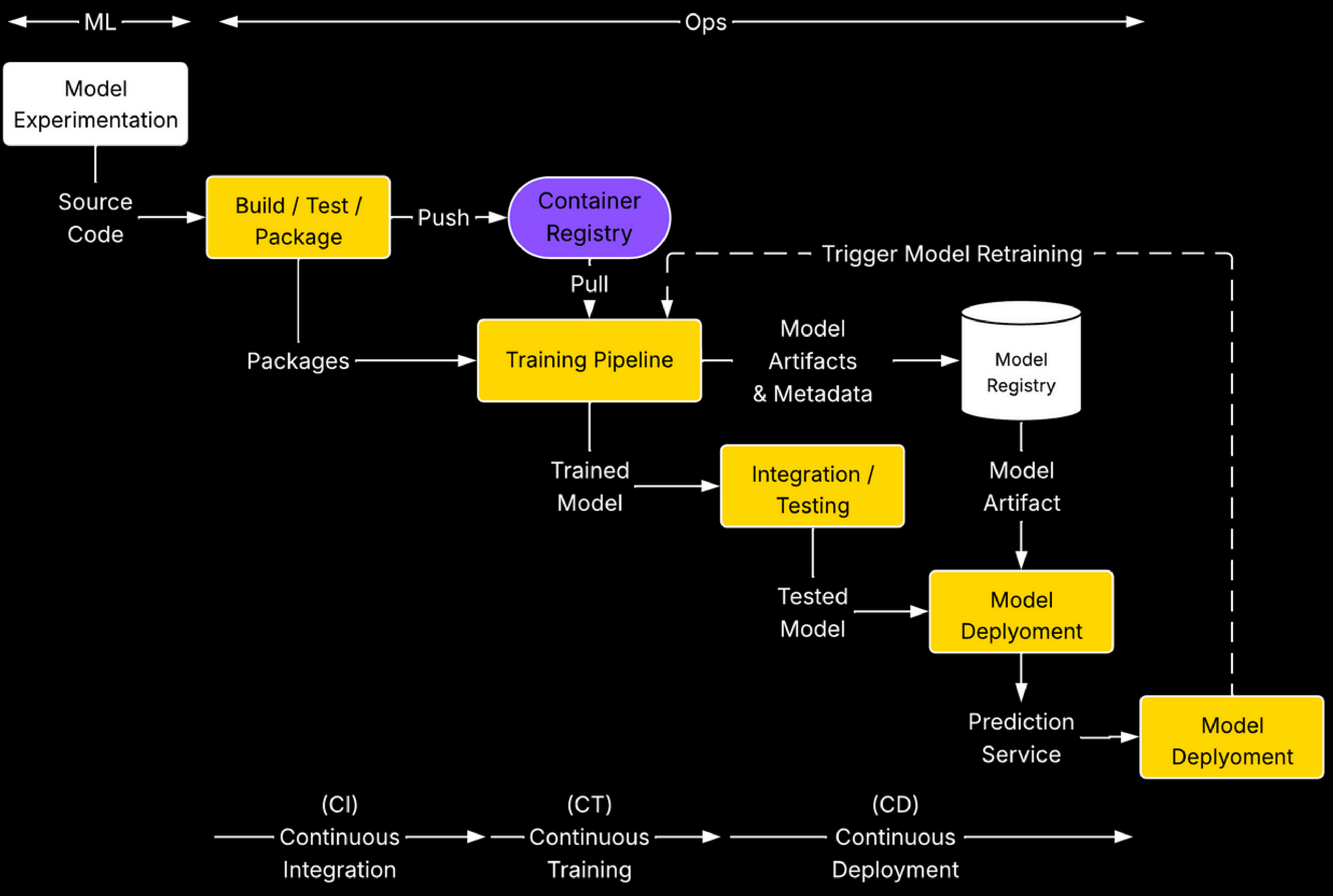
15/ RETRAINING LOOP

Why retrain?
→ Model quality drops over time (new users, new patterns, outdated logic)

What can trigger it?
→ Data drift, accuracy drop, new data, schema changes

How to automate it:
→ Cron job, GitHub Actions, or an Airflow DAG

What happens during retraining?
→ Pull fresh data → rerun pipeline → train → evaluate → register new model



Why? *ML systems degrade over time, a retraining loop keeps them relevant and accurate.*

16/ COMMON MISTAKES

Most ML mistakes are small, but costly.

They often hide in setup, not in the modeling itself.

Data Leakage:
You used future data in training without realizing it.

No Model Versioning
You deployed a model, but no one knows which one.

Only Trained on Training Data
No test set, no generalization.

Manual Retraining
You reran the notebook and silently broke something.

Silent Failures
The API returns results, but they're wrong, and no one knows.

Drift Ignored
Your input data changed months ago, but your model didn't.

No Monitoring
Something goes wrong, but there are no alerts.

Too Many Experiments, No Tracking
You tried 20 things, but forgot which one worked best.

Why? *Tiny mistakes in setup often break even great models later.*

WHICH STAGE YOU FOUND THE MOST INTERESTING?

COMMENT BELOW

FOLLOW ANAS AND MARKUS FOR MORE CONTENT ON DATA SCIENCE, AND ML ENGINEERING.