Building ML Infrastructure and Deploying Production-Ready Models

Alexander D'hoore

Welcome to the Workshop Day

- This is part of a 3-day hands-on ML workshop series.
- Today's focus: **Machine Learning Infrastructure & MLOps**.
- We'll bridge the gap between notebook experiments and real-world deployments.

Course Overview

What We'll Do Today

- Learn how to take **ML prototypes** and turn them into **production** systems.
- Explore **best practices** from modern software engineering.
- Build confidence working with **deployment**, **automation**, and **infrastructure** tools.

Topics We'll Cover

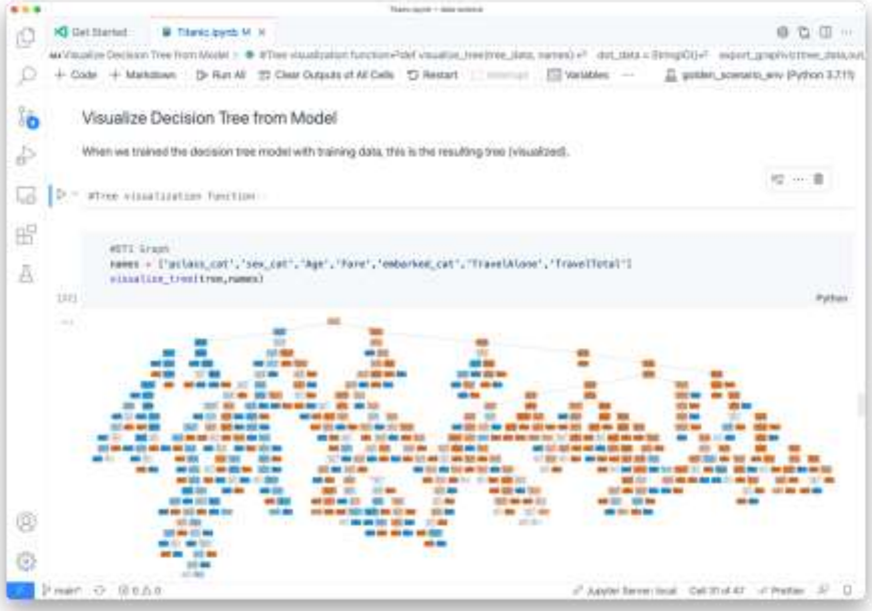
- **Refactoring notebooks** into testable Python code
- **Model deployment** via APIs (REST, gRPC, MQTT, WebSocket)
- **Docker and containerization** for reproducibility
- **Orchestration** with Docker Compose and Kubernetes
- **Cloud & On-Premise infrastructure** with Infrastructure as Code
- **Object storage** for data and models
- **CI/CD automation** with GitHub Actions
- **Data pipelines** using Airflow, Prefect, Dagster

From Notebooks to Production Code

Let's Get Started!

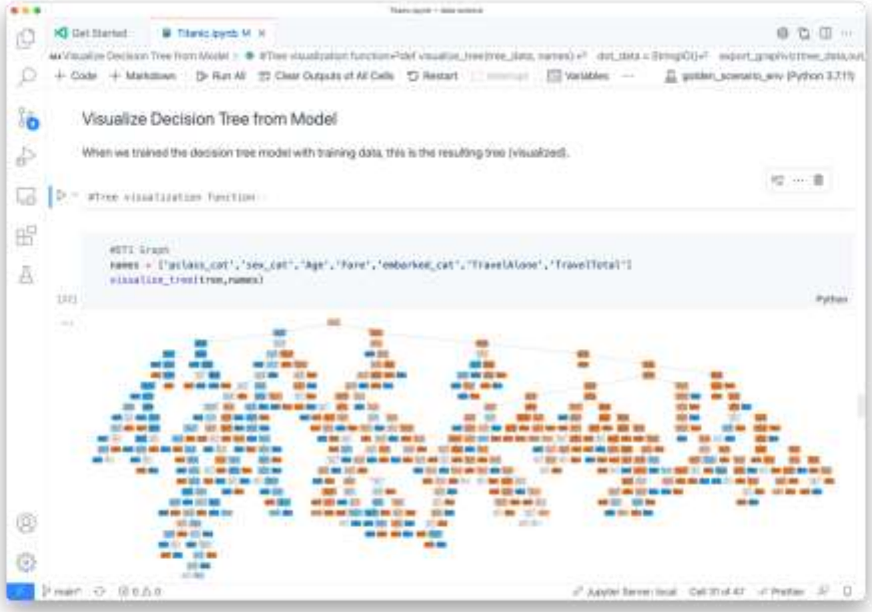
- First up: why we need to move **beyond notebooks**.
- What does "production-ready ML" actually look like?

- How do we get there?
- And what tools are involved?



From Notebooks to Production Code

- ML development often begins in Jupyter notebooks.
- Notebooks are great for:
  - Data exploration
  - Visualization
  - Rapid experimentation
- But they are **not suitable for production** environments.



Why Notebooks Don’t Scale

- Notebooks are **not plain text**
  - Stored as JSON: difficult to diff and merge
  - Hard to collaborate on via Git
- Difficult to reuse or test
  - No modular functions
  - No test coverage
- Poor integration with tools:
  - Formatters, linters, CI/CD, IDEs

```
"metadata": {
 "kernelspec": {
  "display_name": ".venv",
  "language": "python",
  "name": "python3"
 },
 "language_info": {
  "codemirror_mode": {
   "name": "ipython",
   "version": 3
  },
  "file_extension": ".py",
  "mimetype": "text/x-python",
  "name": "python",
  "nbconvert_exporter": "python",
  "pygments_lexer": "ipython3",
  "version": "3.11.2"
 }
```

What to Do Instead

- Refactor logic into modular .py files
- Use notebooks only as **thin wrappers**
  - Import reusable logic
  - Visualize results
- Combines strengths of both notebooks and Python modules

Refactoring Notebooks for Production

Example: A Messy Notebook (1/2)

Let’s take a common all-in-one notebook and refactor it.

```
# cell 1: install dependencies
!pip install pandas scikit-learn matplotlib seaborn

# cell 2: import libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
```

A Messy Notebook (2/2)

```
# cell 3: load data
df = pd.read_csv("data.csv")
X = df[["feature1", "feature2"]]
y = df["target"]

# cell 4: train and visualize
model = LinearRegression()
model.fit(X, y)
plt.scatter(X["feature1"], y)
plt.plot(X["feature1"], model.predict(X), color="red")
```

Why This is a Problem

- Everything is mixed together
- No separation of concerns
- Impossible to test or reuse
- Not suitable for CI/CD or automation

Project Refactor – Folder Structure

Example folder structure for cleanup:

```
├── data.py
├── model.py
├── train_model.py
├── run_inference.py
├── requirements.txt
├── tests/
│   └── test_model.py
└── notebooks/
    └── explore_and_plot.ipynb
```

data.py: Data Loading

```python
import pandas as pd

def load_data(path="data.csv"):
    df = pd.read_csv(path)
    X = df[["feature1", "feature2"]]
    y = df["target"]
    return X, y
```

model.py: Model Logic

```python
from sklearn.linear_model import LinearRegression
import joblib

def train_model(X, y):
    model = LinearRegression()
    model.fit(X, y)
    return model

def save_model(model, path="model.joblib"):
    joblib.dump(model, path)
```

Training Script: **train_model.py**

```python
from data import load_data
from model import train_model, save_model

X, y = load_data()
model = train_model(X, y)
save_model(model)
```

Inference Script: run_inference.py

```python
from data import load_data
from model import load_model, predict

X, _ = load_data()
model = load_model()
y_pred = predict(model, X)

print(y_pred[:5])
```

Unit Testing with **pytest**

```python
from data import load_data
from model import train_model, predict

def test_model_training_and_prediction():
    X, y = load_data()
    model = train_model(X, y)
    preds = predict(model, X)
    assert len(preds) == len(y)
```

Benefits of This Refactor

- Modular, reusable logic
- Ready for testing and automation
- Easier to debug, scale, and maintain
- Notebooks become **visualization tools**, not core logic

Python Dependencies: requirements.txt

- Avoid !pip install ... in notebooks
- Instead, define all dependencies in a **requirements.txt** file

```
pandas==2.2.3
scikit-learn==1.6.1
matplotlib==3.10.1
seaborn==0.13.2
```

- Ensures **reproducibility** and clean version control
- Use pinned versions to avoid unexpected changes

Generating requirements.txt

- You can automatically generate the file from your environment:

  pip freeze > requirements.txt

- Other users (or CI/CD systems) can then install everything with:

  pip install -r requirements.txt

- Essential for automation and containerization

Isolating Projects: Virtual Environments

- Avoid polluting system Python

- Keep each project self-contained and reproducible

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

- Add .venv/ to your .gitignore

Looking Ahead: Toward Containers

- requirements.txt is just the beginning
- Later in the course, we'll package the whole environment
- Container images will include:
  - Python version
  - System libraries
  - Your code + dependencies

Model Deployment and Serving

Model Deployment and Serving

- Up to now, we trained models using notebooks and Python scripts.

- But training is only the **first step**,
- the real goal is to **serve models**
- to **external users** or systems.

Deployment Strategies

- There are two main approaches to model deployment:

- **Server-side deployment**
- **Edge deployment**

Server-Side Deployment

- The model runs on **centralized server hardware**.
- Clients send requests (e.g. via API) to get predictions.
- Advantages:
  - **Centralized control** over model versions.
  - Access to **high-performance resources**.
  - Better **security** and logging.

Edge Deployment

- Model runs **close to the data source** (e.g. phone, sensor, microcontroller).
- Advantages:
  - **Low latency**, no need to send data to the cloud.
  - **Offline support**: works without internet.
  - Better **privacy**: data stays local.

Choosing a Deployment Strategy

- Server-side is common and flexible.
- Edge is emerging but **resource-constrained**.

- We focus on **server-side** in this workshop.
- Edge deployment will be covered **in a later course**.

Communication Protocols for Model Serving

Making Your Model Accessible

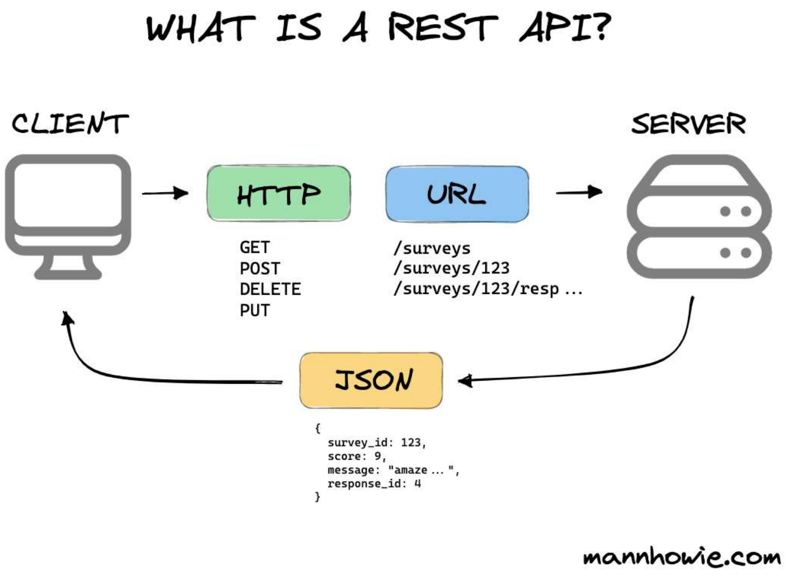
- Once your model is deployed on a server, it needs to talk to clients.

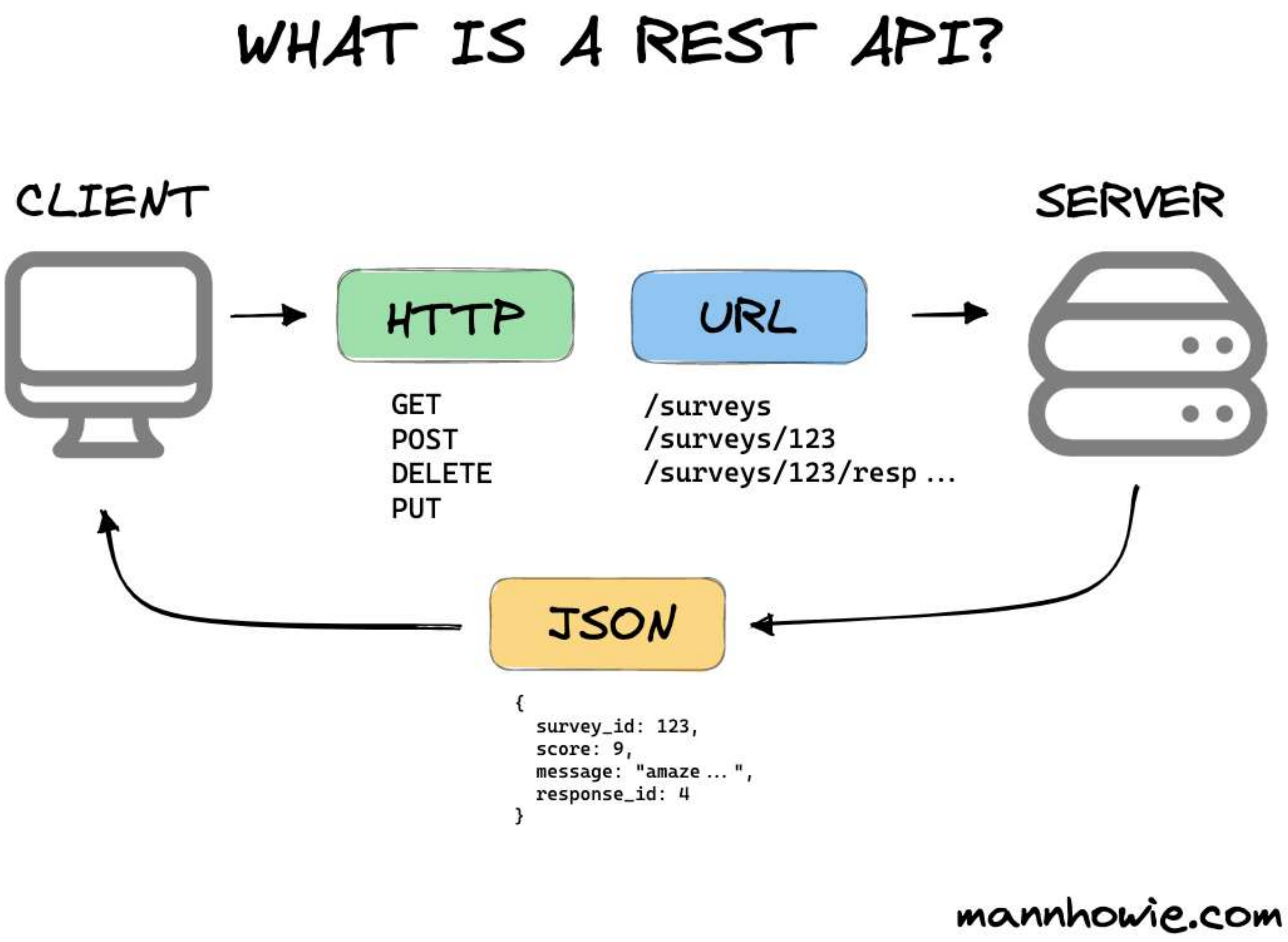
- This requires a **communication protocol** between the server and its users.

- We'll cover the **most common options** used in real-world ML systems.

REST APIs

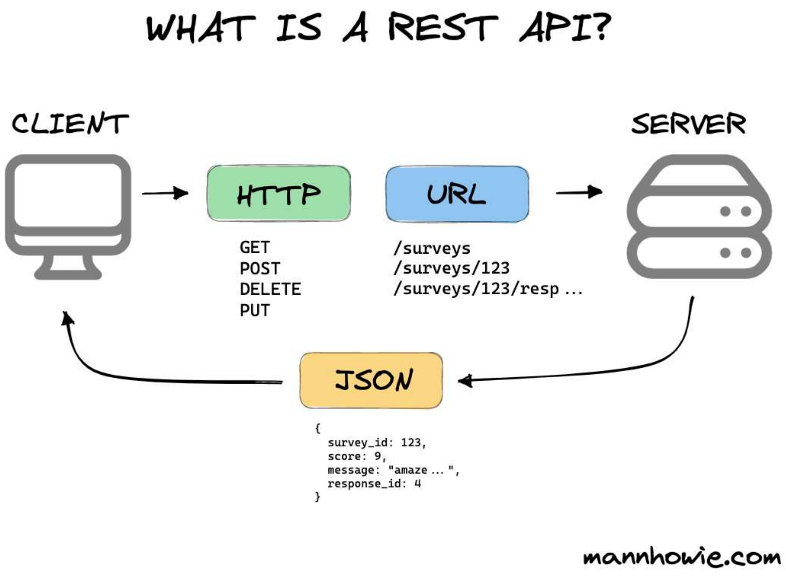
- REST = Representational State Transfer.
- Based on standard HTTP methods:
  - GET, POST, PUT, DELETE
- Works with all major platforms and tools.
- **Stateless**: each request is self-contained.




WHAT IS A REST API?
CLIENT
SERVER
HTTP
URL
GET
POST
DELETE
PUT
/surveys
/surveys/123
/surveys/123/resp ...
JSON
{
survey_id: 123,
score: 9,
message: "amaze ... ",
response_id: 4
}
mannhowie.com

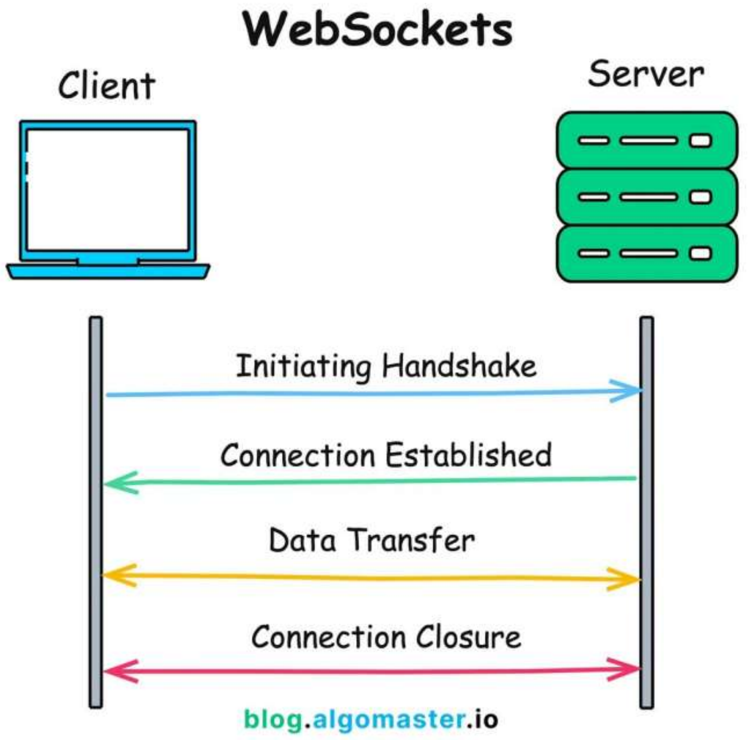
REST APIs – Example & Trade-offs

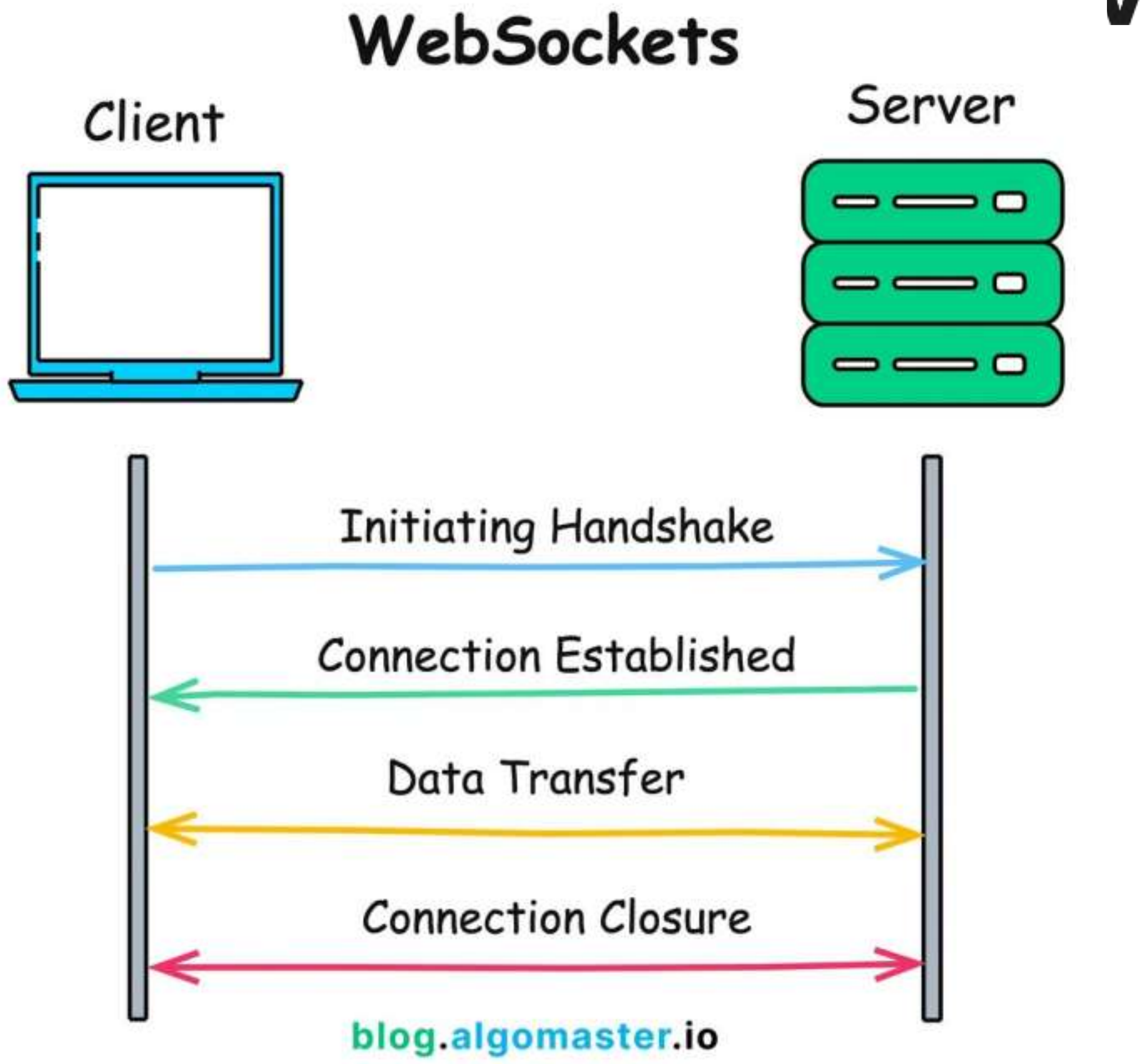
- **Example**:
  A client sends a POST /predict request with features in JSON.
  The server returns a JSON with the prediction.
- **Advantages**:
  - Easy to use and understand.
  - Works well across languages and platforms.
  - Scales well due to statelessness.
- **Considerations**:
  - Verbose payloads (JSON).
  - Higher latency per request.



WebSockets

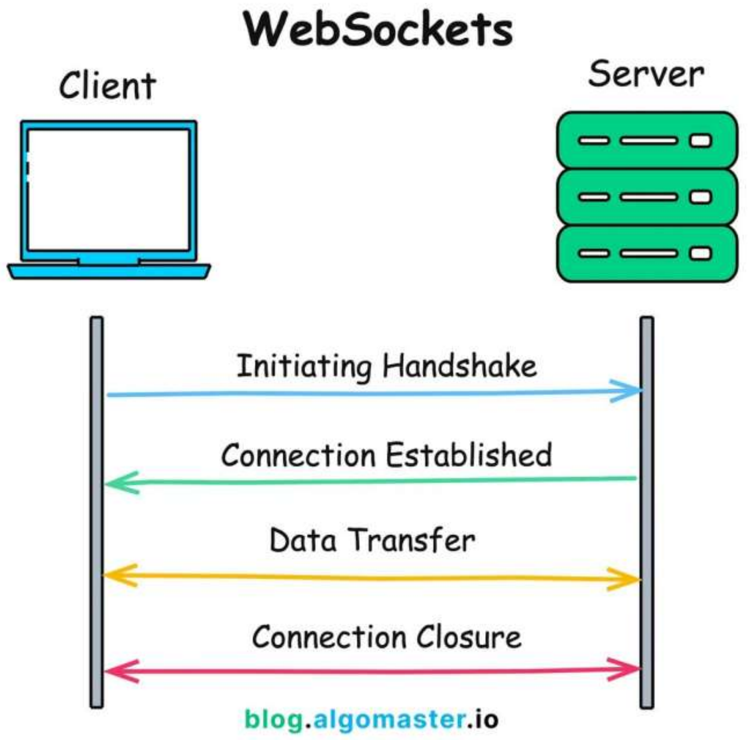
- Provides a **persistent, full-duplex** connection.
- Server and client can push messages to each other anytime.
- Ideal for **real-time** applications (dashboards, chat, etc.).




WebSockets
Client
Server
Initiating Handshake
Connection Established
Data Transfer
Connection Closure
blog.algomaster.io

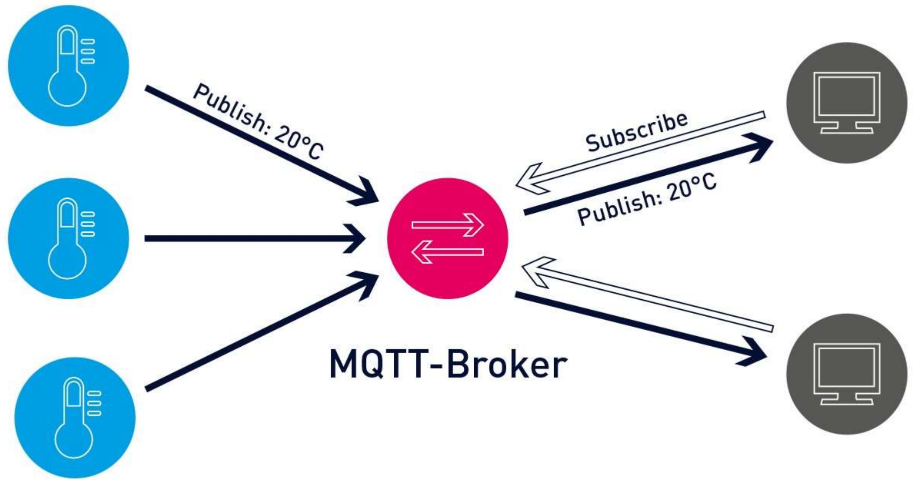
WebSockets – Use Cases

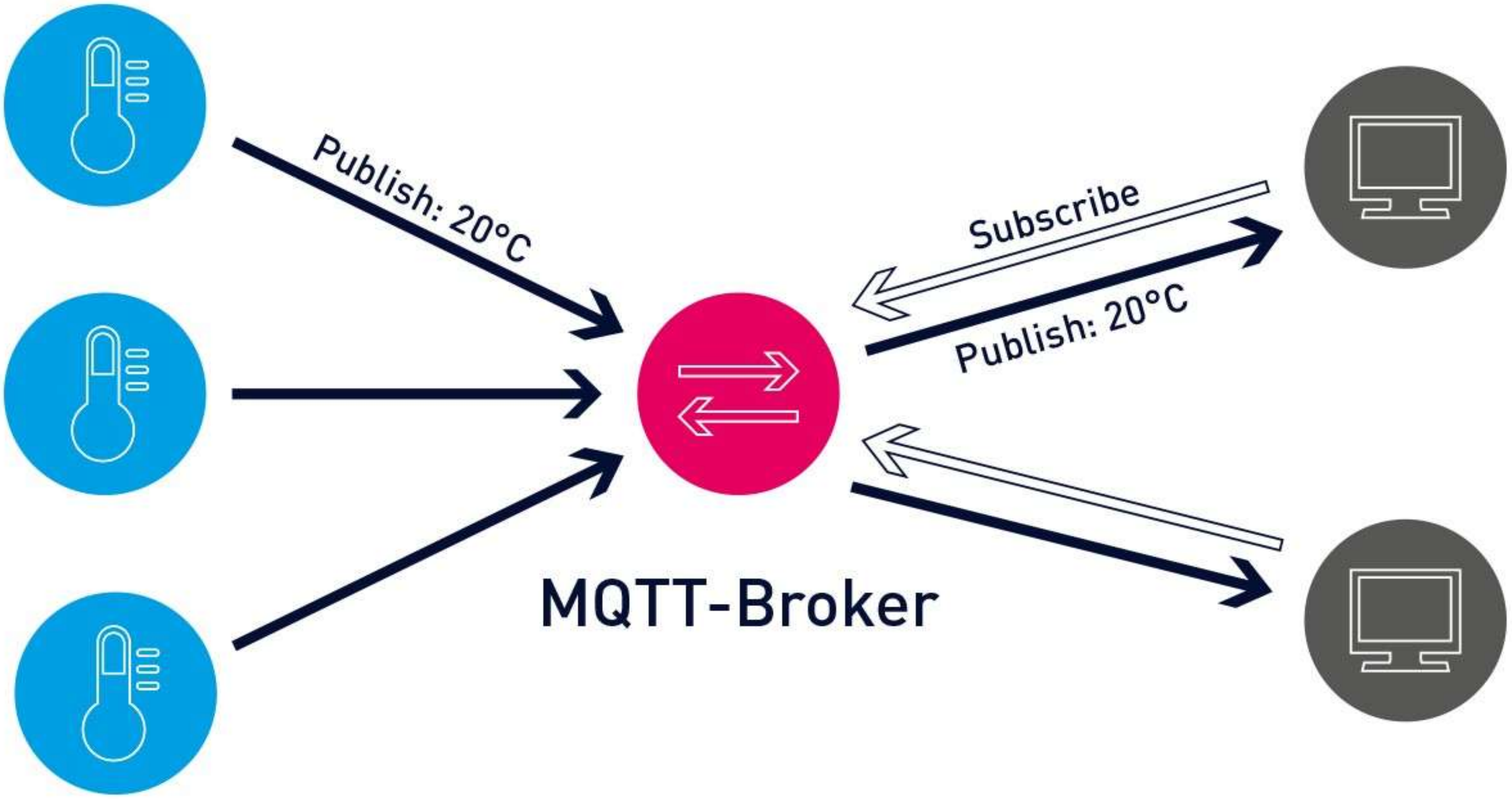
- **Advantages**:
  - Real-time, bidirectional communication.
  - Low overhead after connection is established.
  - Works across platforms.
- **Considerations**:
  - Requires connection management.
  - More complex to scale.
  - Open connections consume server resources.



MQTT

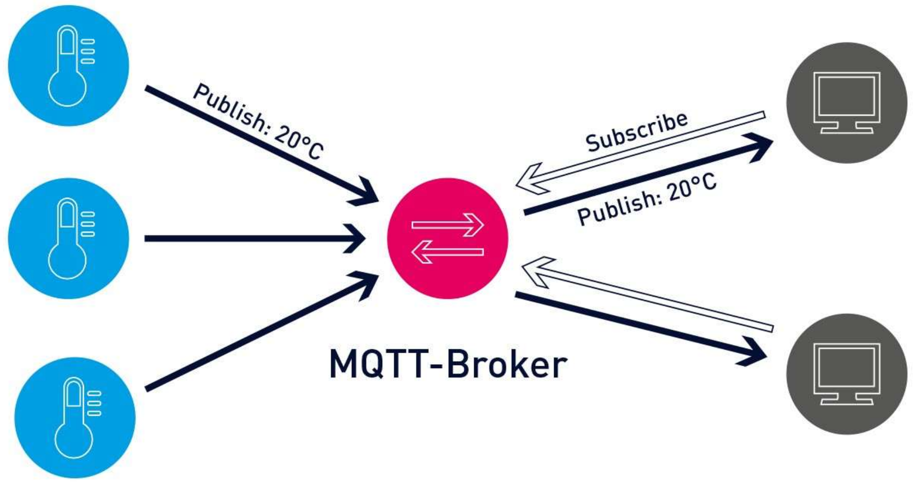
- Lightweight messaging protocol for **IoT and constrained devices**.
- Works on a **publish/subscribe** model.
- Designed for unreliable or low-bandwidth networks.




Publish: 20°C
Subscribe
Publish: 20°C
MQTT-Broker

MQTT – Strengths and Limitations

- **Advantages**:
  - Very lightweight and efficient.
  - Ideal for thousands of devices.
  - Supports reliable delivery (QoS).
- **Considerations**:
  - Not browser-friendly.
  - Needs custom handling for security.



FastAPI for Model Serving

What is FastAPI?

- Modern Python web framework for high-performance APIs.
- Built on Python type hints, making it easy and fast.
- Great for ML engineers and data scientists.

Why FastAPI?

- **Simplicity**: Define APIs with a few Python functions.
- **Performance**: Async support and fast execution.
- **Auto Docs**: Swagger UI is generated automatically.
- Lets you **stay in Python** while building production APIs.
- Perfect for wrapping ML models in REST endpoints.

FastAPI

Creating Routes: Example App (1/2)

```python
from fastapi import FastAPI
from pydantic import BaseModel
from typing import List

app = FastAPI()

class BlogPost(BaseModel):
    title: str
    content: str

posts = []
```

Creating Routes: Example App (2/2)

```python
@app.get("/posts", response_model=List[BlogPost])
def list_posts():
    return posts

@app.post("/posts")
def create_post(post: BlogPost):
    posts.append(post)
    return {"message": "Post added"}
```

Interacting with This API

- **GET** /posts: list all blog posts.
- **POST** /posts: add a new blog post.

- Each route uses decorators like @app.get(...).
- Pydantic models like BlogPost handle validation.

Serving ML Models with FastAPI (1/2)

```python
from fastapi import FastAPI
from pydantic import BaseModel
import torch

app = FastAPI()

model = torch.load("model.pt")
model.eval()
```

Serving ML Models with FastAPI (2/2)

```python
class InputData(BaseModel):
    feature1: float
    feature2: float

@app.post("/predict")
def predict(data: InputData):
    with torch.no_grad():
        inputs = torch.tensor([[data.feature1, data.feature2]])
        prediction = model(inputs)
    return {"prediction": prediction.item()}
```

Calling the Prediction Endpoint

- Send POST /predict with JSON like:


```json
{
  "feature1": 3.5,
  "feature2": 1.2
}
```

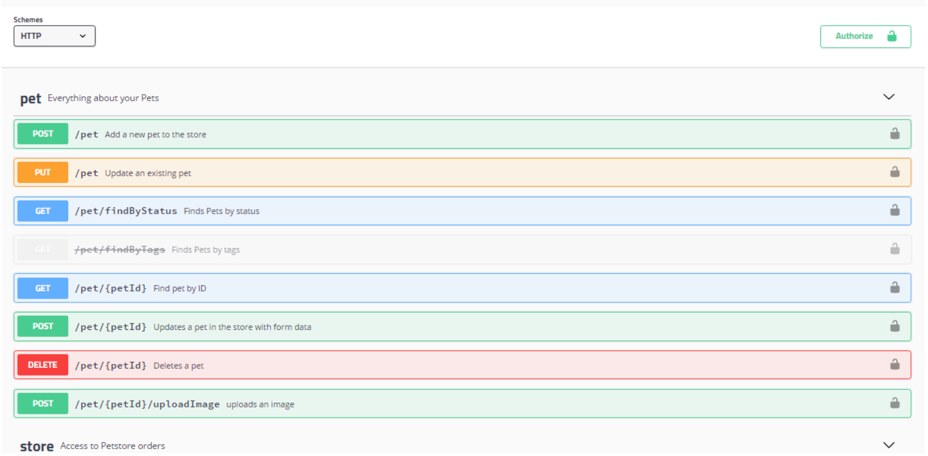

- Server returns:


```json
{
  "prediction": 842000.0
}
```


- Clean, Pythonic, and production-ready.

Swagger UI: Auto Docs for Free

- Visit http://localhost:8000/docs in your browser.
- You get:
    - List of endpoints
    - Input/output schemas
    - Live request testing
    - Developer-friendly interface



Schemes

HTTP

Authorize

pet Everything about your Pets

POST `/pet` Add a new pet to the store

PUT `/pet` Update an existing pet

GET `/pet/findByStatus` Finds Pets by status

GET ~~`/pet/findByTags`~~ Finds Pets by tags

GET `/pet/{petId}` Find pet by ID

POST `/pet/{petId}` Updates a pet in the store with form data

DELETE `/pet/{petId}` Deletes a pet

POST `/pet/{petId}/uploadImage` uploads an image

store Access to Petstore orders

Frontends for Model APIs

Frontends

- A model API needs clients to interact with it — this is the **frontend**.

- Could be:
  - Web browsers
  - Mobile apps
  - Desktop tools
  - Embedded systems

- We'll focus mainly on **browser-based** frontends today.

JavaScript Frameworks

- Popular for building interactive web apps.
- Written in JavaScript, running fully in the browser.
- Sends requests (via Fetch/AJAX) directly to REST APIs.
- Examples:
    - **React**
    - **Vue.js**
    - **Svelte**
- Very scalable and flexible.
- Backend and frontend are decoupled.

Python-Based Frontends with Flask

- Handy for **internal tools** or prototyping in Python.
- Flask serves:
  - The HTML frontend
  - Requests to the model-serving backend
- Simple: no JavaScript required.
- Acceptable overhead for internal apps.

```python
from flask import Flask, request, render_template_string
import requests
```

Dashboarding: Dash, Streamlit, Gradio

- Build UIs in Python, no HTML/JS required.
- Tools:
  - **Dash** (from Plotly)
  - **Streamlit**
  - **Gradio**
- Run on server, call model API from backend Python.
- Ideal for:
  - Rapid prototyping
  - Interactive demos
  - Internal dashboards


Settings
Categorical Variable
cat0
Continuous Variable
cont0
Continuous Variables for Correlation Matrix
cont0
cont1
cont2
cont3
cont4
cont5
cont6
cont7
cont8
cont9
cont10
target
apply
Target Variables
26.3%
73.7%
0
1
Distribution of Categorical Variable: cat0
target=1
target=0
600
400
200
0
A
B
Distribution of Continuous Variable: cont0
target=1
target=0
2.5
2
1.5
1
0.5
0
0
0.2
0.4
0.6
0.8
1
Correlation Matrix Heatmap
cont0 cont1 cont2 cont3 cont4 cont5 cont6 cont7 cont8 cont9 cont10 target
target -0.0 0.22 0.18 -0.16 -0.08 0.23 0.22 -0.01 0.28 0.09 -0.03 1.0
cont10 0.79 0.47 0.5 0.61 0.2 -0.14 -0.43 0.78 0.37 0.45 1.0 -0.03
cont9 0.38 0.4 0.42 0.35 0.11 0.16 -0.08 0.44 0.33 1.0 0.45 0.09
cont8 0.38 0.71 0.66 0.11 0.08 0.18 0.14 0.48 1.0 0.33 0.37 0.28
cont7 0.74 0.58 0.59 0.61 0.24 -0.05 -0.35 1.0 0.48 0.44 0.78 -0.01
cont6 -0.43 0.14 0.09 -0.42 -0.1 0.45 1.0 -0.35 0.14 -0.08 -0.43 0.22
cont5 -0.16 0.21 0.15 -0.09 -0.02 1.0 0.45 -0.05 0.18 0.16 -0.14 0.23
cont4 0.18 0.14 0.16 0.21 1.0 -0.02 -0.1 0.24 0.08 0.11 0.2 -0.08
cont3 0.53 0.2 0.23 1.0 0.21 -0.09 -0.42 0.61 0.11 0.35 0.61 -0.16
cont2 0.51 0.87 1.0 0.23 0.16 0.15 0.09 0.59 0.66 0.42 0.5 0.18
cont1 0.49 1.0 0.87 0.2 0.14 0.21 0.14 0.58 0.71 0.4 0.47
cont0 1.0 0.49 0.51 0.53 0.18 -0.16 -0.43 0.74 0.38 0.38 0.79

When to Use Which Frontend?

- **React/Vue**: Public or production web apps.
- **Flask**: Quick prototypes, Python-first teams.
- **Dash/Streamlit/Gradio**: Demos, dashboards, ML exploration.

- Tradeoffs:
  - Flexibility vs. simplicity
  - JS skills vs. Python comfort
  - Customization vs. speed of development

Beyond the Browser

- Not all clients are browsers.
- Other frontend types include:
  - **Smartphone apps** (Kotlin, Swift, Flutter)
  - **Desktop apps** (Electron, Qt, C#)
  - **IoT & embedded** (C++, Go, Rust)

- As long as they can send HTTP/MQTT/etc they can talk to your model server.

Containers and Virtual Machines

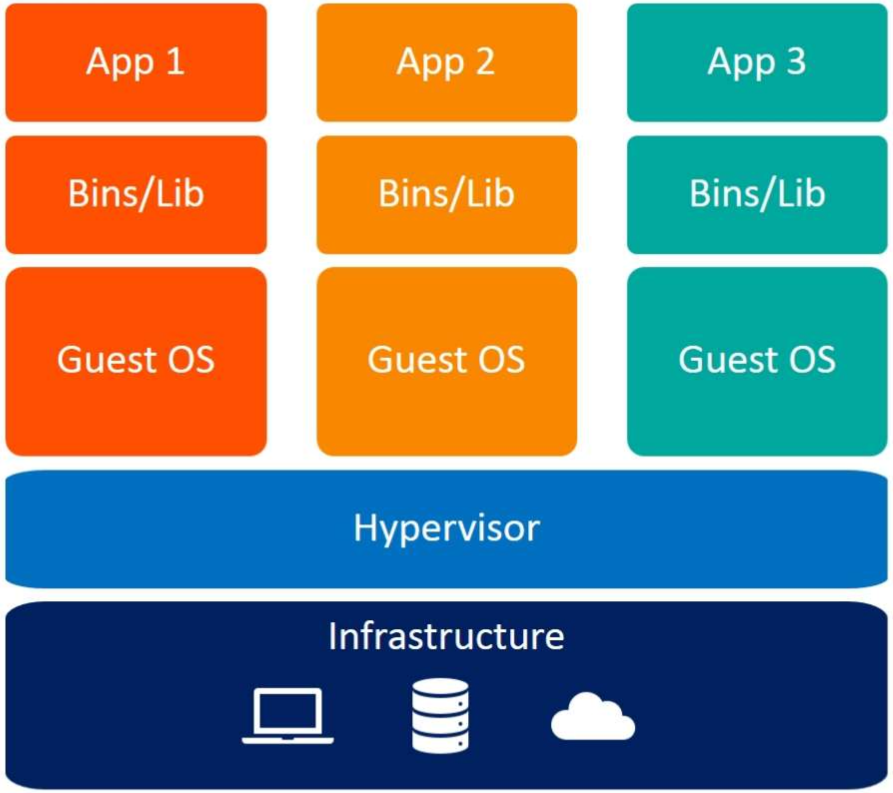
Why We Need Containers and VMs

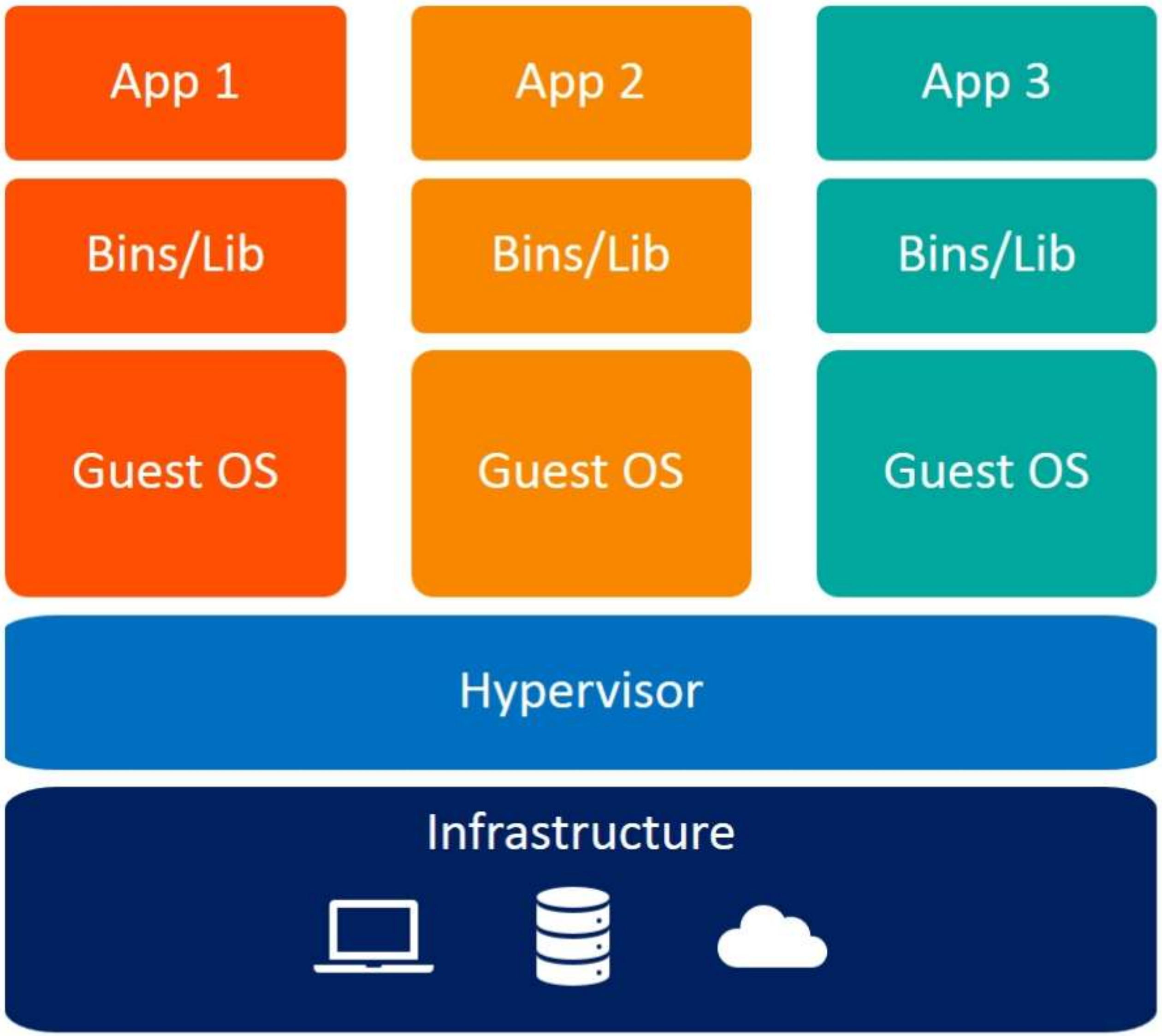
- We've learned how to train models and expose them via APIs.
- But deploying these services **reliably and consistently** is critical.
- We need packaging tools that work across:
  - Development
  - Staging
  - Production
- This is where **virtual machines** and **containers** come in.

Virtual Machines (VMs)

- A VM is a full emulation of a computer system.
- Runs a complete **guest OS** on top of a **host OS**.

- VMs include:
  - Their own OS kernel
  - File system and system libraries
  - Network stack and system tools

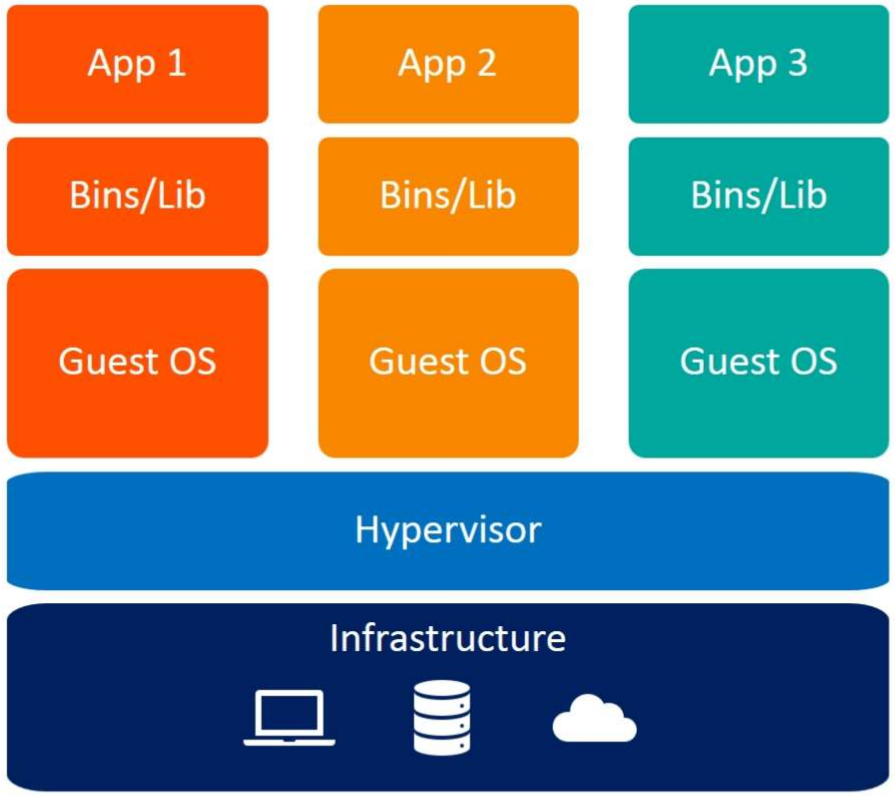



App 1
App 2
App 3
Bins/Lib
Bins/Lib
Bins/Lib
Guest OS
Guest OS
Guest OS
Hypervisor
Infrastructure

VM Overhead

- Downsides of VMs:
  - High memory and disk usage (gigabytes).
  - Slow startup (can take minutes).
  - Complex to manage at scale.

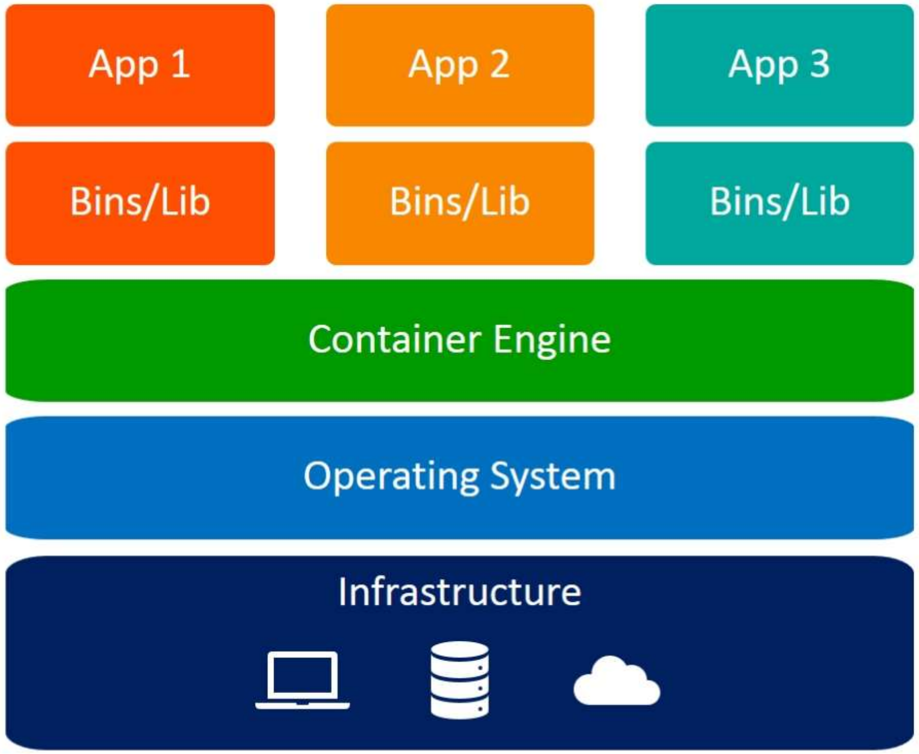
- Still valuable for infrastructure and OS-level testing.

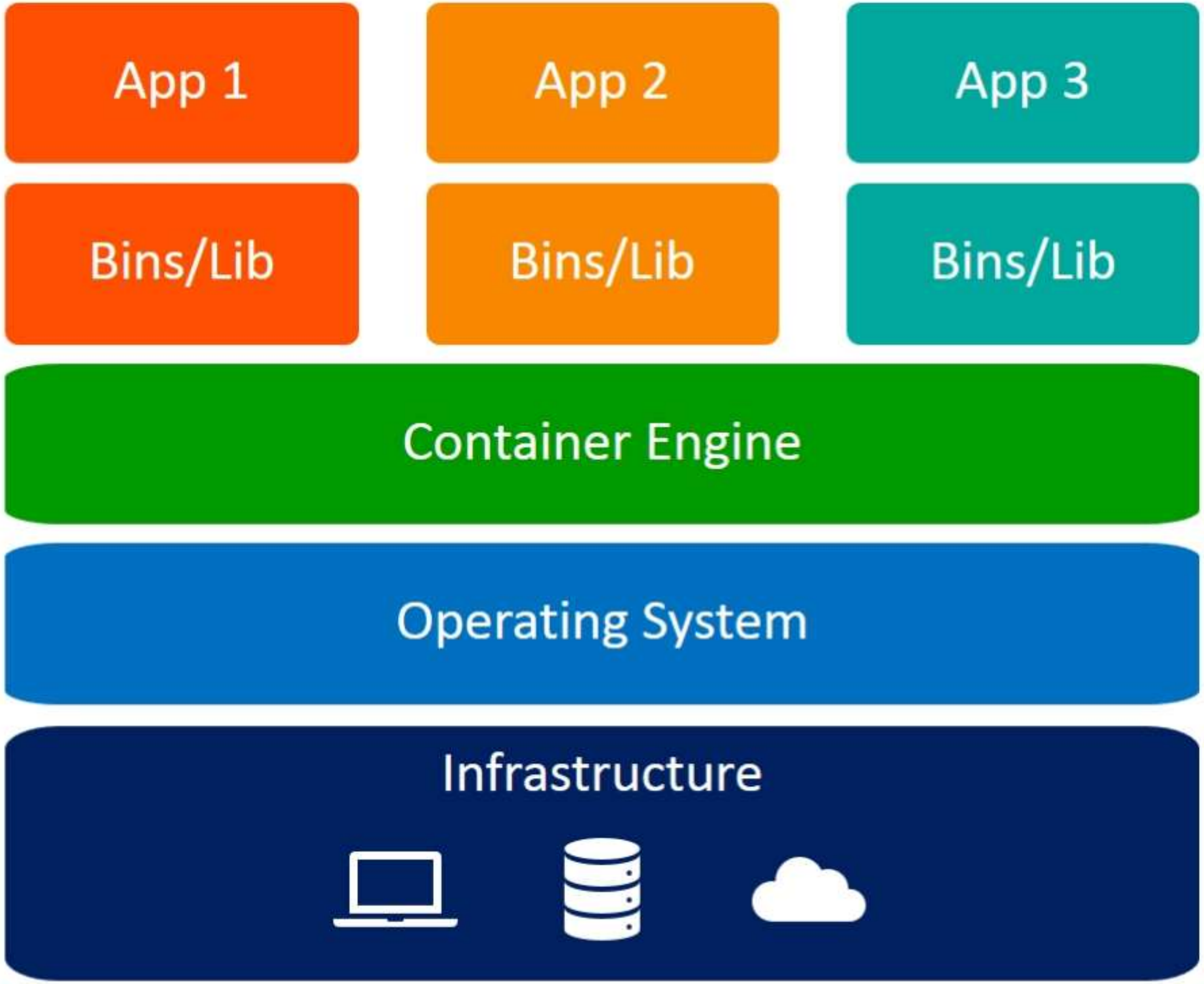


Containers

- Containers don’t run their own OS.
- Instead, they **share the host OS kernel**.

- A container includes:
  - Application code
  - Dependencies (Python libs, binaries)
  - Minimal system utilities

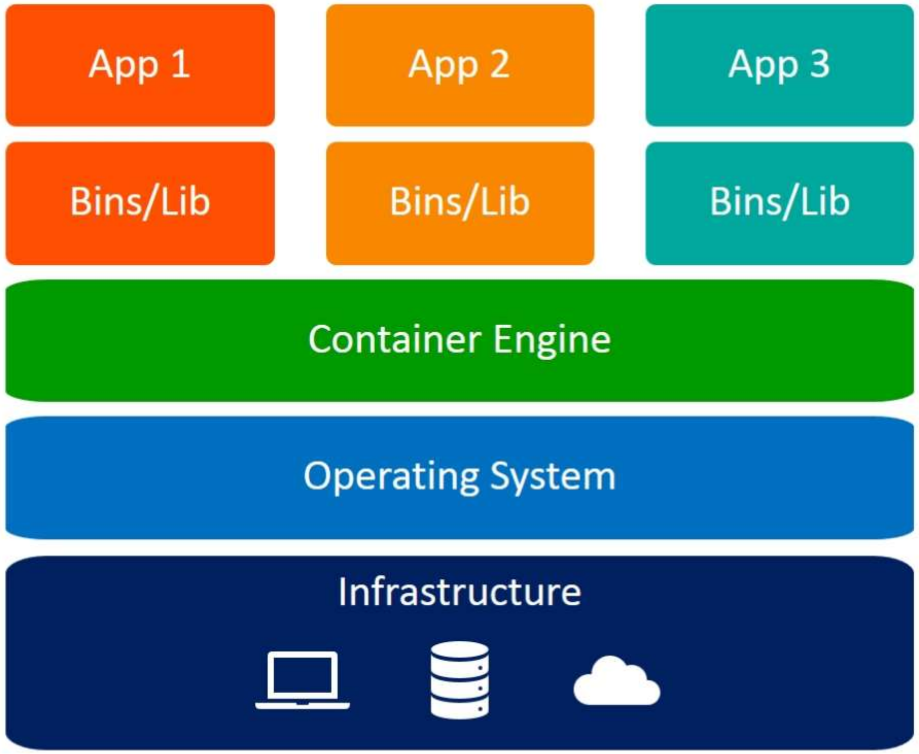



App 1
App 2
App 3
Bins/Lib
Bins/Lib
Bins/Lib
Container Engine
Operating System
Infrastructure

Why Containers Are Better for ML

- Containers start in **milliseconds**, not minutes.
- Use **megabytes**, not gigabytes.
- Easy to deploy and scale.

- Ideal for microservices and ML model serving.



Containers for Machine Learning

- ML models depend on exact library versions
  (e.g. NumPy, PyTorch, CUDA).
- Containers **capture the full environment**, not just your code.
- Makes ML projects:
  - **Reproducible**
  - **Portable**
  - **Deployable**
  - **Traceable**

Versioning and Reproducibility

- Containers are **versionable artifacts**:
  - Tag them: ml-service:1.0.0
  - Rebuild them deterministically
- Enables:
  - Safe rollbacks
  - Exact replays of past training runs
  - Consistent behavior across machines

- Key MLOps principle: **infrastructure = code**

Why This Matters in Practice

- Avoid:
  - “It works on my machine”
  - Library conflicts
  - Inconsistent runtime environments

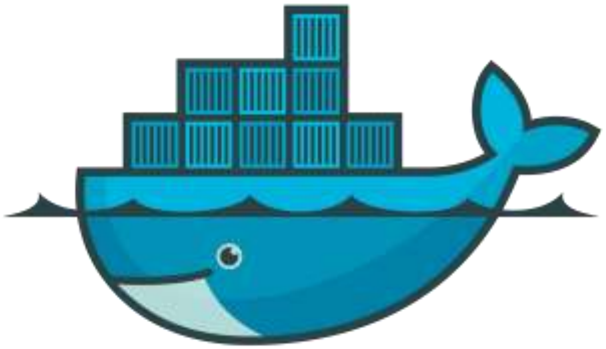
- Gain:
  - Confidence in experiments
  - Smooth collaboration across teams
  - Safer deployments in production

Introduction to Docker

What is Docker?

- Docker is the most widely used tool for working with containers.
- Packages your app and its dependencies into a **portable image**.
- Run it anywhere — dev machine, server, or cloud — **with the same behavior**.

- Solves the “it works on my machine” problem.



Running Your First Docker Container

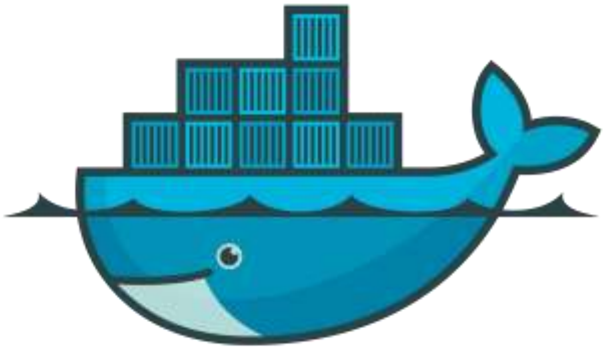
- Use pre-built images from **Docker Hub**.

  docker run hello-world

- Downloads and runs a test image to verify setup.

  docker run -it python:3.12

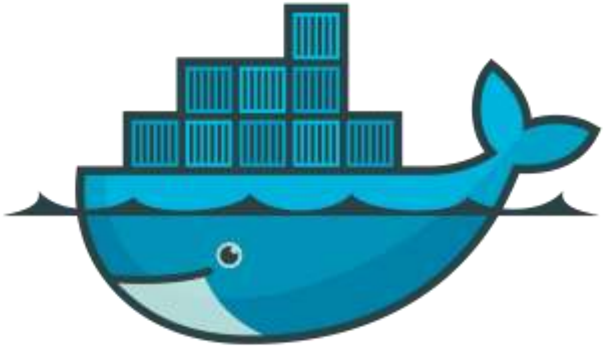
- Starts Python 3.12 in interactive mode (-it).
- Gives you a Python shell inside a container.



Useful Docker CLI Commands

- docker ps – show running containers
- docker ps -a – include stopped containers
- docker stop <name> – stop a container
- docker rm <name> – remove a stopped container
- docker images – list downloaded images

These help you **inspect and manage** your containers and images.



Example: PostgreSQL with Docker

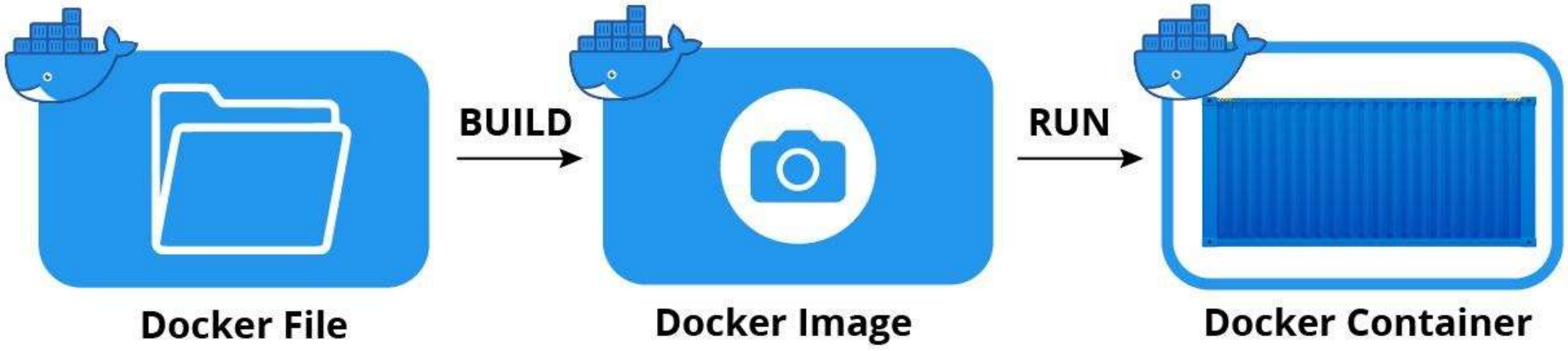
```
docker run -d --rm \
  -e POSTGRES_PASSWORD=secret \
  -p 5432:5432 \
  -v ./pgdata:/var/lib/postgresql/data \
  --name my-postgres postgres:17
```

- Runs PostgreSQL 17 in the background (-d)
- Cleans up automatically after shutdown (-rm)
- Exposes port 5432 for connections (-p)
- Persists data in ./pgdata (-v)
- Assigns an easy name for management (--name)

Building Custom Docker Images

Why Build Your Own Image?

- Prebuilt images are useful — but often not enough.
- Custom images let you:
  - Install your own dependencies
  - Package your code
  - Create consistent environments for others



Minimal Example: FastAPI Service

A simple Dockerfile that builds a container for a FastAPI app:

```
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .

CMD ["fastapi", "run", "main.py"]
```

Building the Image

To build the image:

docker build -t mlservice:1.0 .

- -t assigns a name and version tag.
- The . at the end means “build from current directory”.

You can now run this container like any other image.

Publishing to Docker Hub

```
docker login
docker tag mlservice:1.0 your-name/mlservice:1.0
docker push your-name/mlservice:1.0
```

- docker tag gives your image a full registry name.
- docker push uploads it to Docker Hub.
- Anyone can then pull and run it.

Alternative: Using Quay.io

```
docker login quay.io
docker tag mlservice:1.0 quay.io/your-org/mlservice:1.0
docker push quay.io/your-org/mlservice:1.0
```

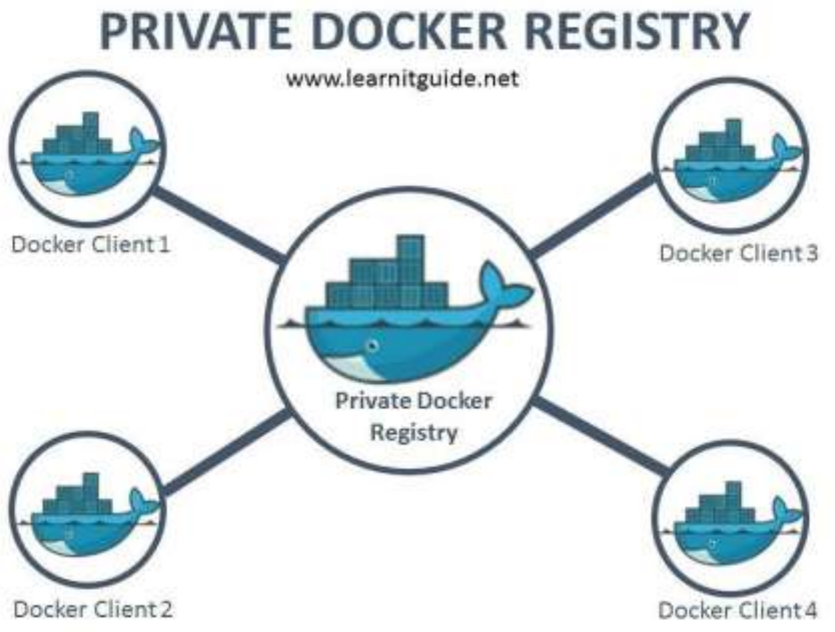
- Similar to Docker Hub.
- Has better usage limits (downloads)

Red Hat
Quay

Alternative: Custom Registry

```
docker login registry.example.com
docker tag mlservice:1.0 registry.example.com/team/mlservice:1.0
docker push registry.example.com/team/mlservice:1.0
```

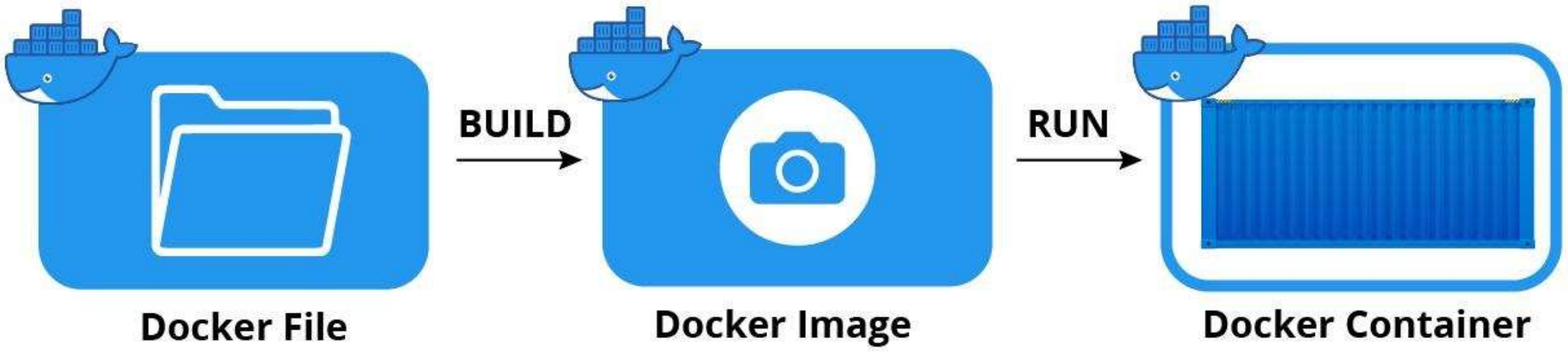
- Useful for:
  - Private images
  - Corporate environments
  - Air-gapped systems



Understanding Images vs. Containers

- **Image** = static blueprint (read-only)
- **Container** = running process with that image

Running an image creates a **new container instance**.



Avoiding the **latest** Tag

- Never use floating tags like **latest**.

```
FROM python:latest  # ❌ Don't do this
```

- Use **pinned** versions:

```
FROM python:3.12
```

- Same for requirements.txt:

```
pandas==2.2.3
scikit-learn==1.6.1
```

- Ensures **reproducibility** and safe deployments.

Containers and Docker for ML

- Containers let us **package code, dependencies, and environments** into a single unit.

- They solve real-world ML problems:
  - ✅ **Reproducibility**: Same environment every time
  - ✅ **Portability**: Runs on laptop, server, or cloud
  - ✅ **Isolation**: No conflicts with system or other projects
  - ✅ **Scalability**: Containers are lightweight and easy to deploy

💡 Containers are the **foundation** for deploying reliable, production-grade ML systems.

Introduction to Container Orchestration

Why Orchestration Matters

- Single containers are great for isolated tasks.
- Real-world systems involve **many containers** working together.

- Examples in ML systems:
  - FastAPI model server
  - PostgreSQL database
  - Object storage (S3)
  - Frontend (React, Vue)
  - Caching (Redis)
  - Monitoring (Grafana)

Tools for Orchestration

- Two common tools:
  - **Docker Compose**: Simple, local development
  - **Kubernetes**: Production-grade, distributed

- We'll start with **Docker Compose**.

What Is Docker Compose?

- Tool to define and run **multi-container apps** using YAML.

- Created for small to mid-sized projects.
- Runs everything with a single command.

Example: ML Deployment with Compose (1/2)

```yaml
services:
  frontend:
    image: my-frontend:1.0
    ports:
      - "3000:3000"
  backend:
    build:
      context: ./backend
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=postgresql://user:${PW}@db:5432/appdb"
```

Example: ML Deployment with Compose (2/2)

```yaml
  db:
    image: postgres:15
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=${PW}
      - POSTGRES_DB=appdb
    volumes:
      - pgdata:/var/lib/postgresql/data

volumes:
  pgdata:
```

Compose Project Breakdown

- **frontend**: Exposes port 3000 to host.
- **backend**:
  - Built from local Dockerfile
  - Uses env var to connect to database
- **db**:
  - Official Postgres image
  - Uses named volume for persistence

Starting the System

docker compose up

- Builds and runs all services.

Just a **single command** to bring up the **entire system**!

Core Docker Compose Commands

- docker compose up — start all services
- docker compose up -d — run in background
- docker compose up --build — force rebuild

- docker compose down — stop and remove everything

- docker compose build — build images from source

- docker compose ps — list running containers

- docker compose logs — view logs
- docker compose logs -f backend — follow backend logs

Using Prebuilt Images

```yaml
services:
  frontend:
    image: registry.example.com/my-frontend:1.0
```

- Compose pulls the image if missing.
- Fast and easy if the image already exists.

Building Local Images

```yaml
services:
  backend:
    build:
      context: ./backend
```

- Builds image from local Dockerfile.
- Use this during development.
- Rebuild with:

```
docker compose up --build
```

Exposing Services to Host

```yaml
ports:
  - "3000:3000"
  - "8000:8000"
```

- Frontend → localhost:3000
- Backend → localhost:8000

- Useful for browser access, Postman, etc.

Keeping Services Internal

```yaml
services:
  db:
    image: postgres:15
    environment:
      ...
```

- No ports: means not exposed externally.
- Improves **security** by reducing attack surface.

Volumes: Bind Mounts

```yaml
volumes:
  - ./notebooks:/home/jovyan/work
```

- Syncs folder from host to container.
- Good for development and notebooks.

- But:
  - Host-dependent
  - Can have permission issues

Volumes: Named Volumes

```yaml
volumes:
  - pgdata:/var/lib/postgresql/data

volumes:
  pgdata:
```

- Docker-managed storage.
- Survives compose down and container restarts.
- Ideal for **databases** and **production**.

Environment Variables (Inline)

```yaml
environment:
  - DEBUG=true
  - MAX_WIDTH=1000
  - NAME=example.com
```

- Simple, but hardcoded.
- Not suitable for secrets.

Environment Variables from .env

.env file:

```
DEBUG=true
BACKEND_PORT=8000
```

docker-compose.yaml:

```yaml
ports:
  - "${BACKEND_PORT}:${BACKEND_PORT}"
environment:
  - DEBUG=${DEBUG}
```

- Great for development **overrides** and **secrets**.

Kubernetes: Industrial-Scale Container Orchestration

Why Kubernetes?

- Docker Compose is great for small/local setups.
- Kubernetes solves orchestration at **production scale**.
- Designed to manage containers across **multiple machines**.
- Originally from Google; now maintained by the CNCF.

Kubernetes = a Platform, Not Just a Tool

- Requires a control plane:
  - Scheduler
  - Network controller
  - Volume manager
  - Service discovery

- Works using **declarative configuration**:
  - You define the desired state,
    Kubernetes enforces it.

Example: PostgreSQL with Kubernetes (1/4)

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pgdata
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Example: PostgreSQL with Kubernetes (2/4)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: db
  template:
    metadata:
      labels:
        app: db
```

Example: PostgreSQL with Kubernetes (3/4)

```yaml
spec:
  containers:
    - name: db
      image: postgres:15
      env:
        - name: POSTGRES_USER
          value: user
        - name: POSTGRES_PASSWORD
          value: password
        - name: POSTGRES_DB
          value: appdb
```

Example: PostgreSQL with Kubernetes (4/4)

```yaml
    volumeMounts:
      - name: pgdata
        mountPath: /var/lib/postgresql/data
volumes:
  - name: pgdata
    persistentVolumeClaim:
      claimName: pgdata
```

Why It’s More Complex

- More boilerplate than Compose

- Also includes:
  - Load balancing
  - Health checks
  - Auto-scaling
  - Monitoring

- **Huge power**, but higher **learning curve**

Kubernetes Is Not a Single Product

- Kubernetes defines **a standard**, not one tool

- Available from many sources:
  - **Cloud**: AWS, Azure, GCP
  - **On-prem**: K3s, MicroK8s
  - **Enterprise**: OpenShift, VMware Tanzu

Advanced Features: Scaling & Load Balancing

- Declare number of **replicas** per service
- Kubernetes distributes them and balances traffic

```yaml
spec:
  replicas: 3
```

- Containers are auto-restarted on failure

Advanced Features: Rolling Updates

- Kubernetes updates containers **without downtime**
- If rollout fails:
  - Kubernetes rolls back automatically
- Health checks detect broken containers

Advanced Features: Persistent Storage

- Supports:
  - NFS
  - Cloud volumes (EBS, Azure Disk)
  - Distributed file systems (Ceph)

- Works across **nodes**, not just local machine

Advanced Features: Resource Scheduling

- Set resource limits per container:

```yaml
resources:
  requests:
    cpu: "500m"
    memory: "256Mi"
  limits:
    cpu: "1"
    memory: "512Mi"
```

- Kubernetes uses this for
  fair **scheduling** and capacity planning

Orchestration: When to Use Each

- Use **Compose** for:
  - Dev environments
  - Simple demos
  - Local tools and dashboards

- Use **Kubernetes** for:
  - High-availability services
  - Production workloads
  - Teams and CI/CD pipelines

Infrastructure Management: Cloud, On-Premise and Infrastructure as Code

Why Infrastructure Matters

- ML applications need **more than code**:
  containers, orchestration, and hosting.

- We've used Docker and Kubernetes,
  but where do we run them?

- Real-world ML systems need:
  - **Always-on** compute
  - **Scalable** deployment
  - **Secure** networking and data access

What Is a Cloud Provider?

- A **cloud provider** rents out computing resources over the internet:
  - Virtual machines, storage, networking, databases, etc.
- You **pay for what you use** — no need to maintain hardware yourself.

- **Ideal for ML workloads** that vary over time (e.g. training jobs).

- No up-front investment: rent GPUs or large VMs as needed.

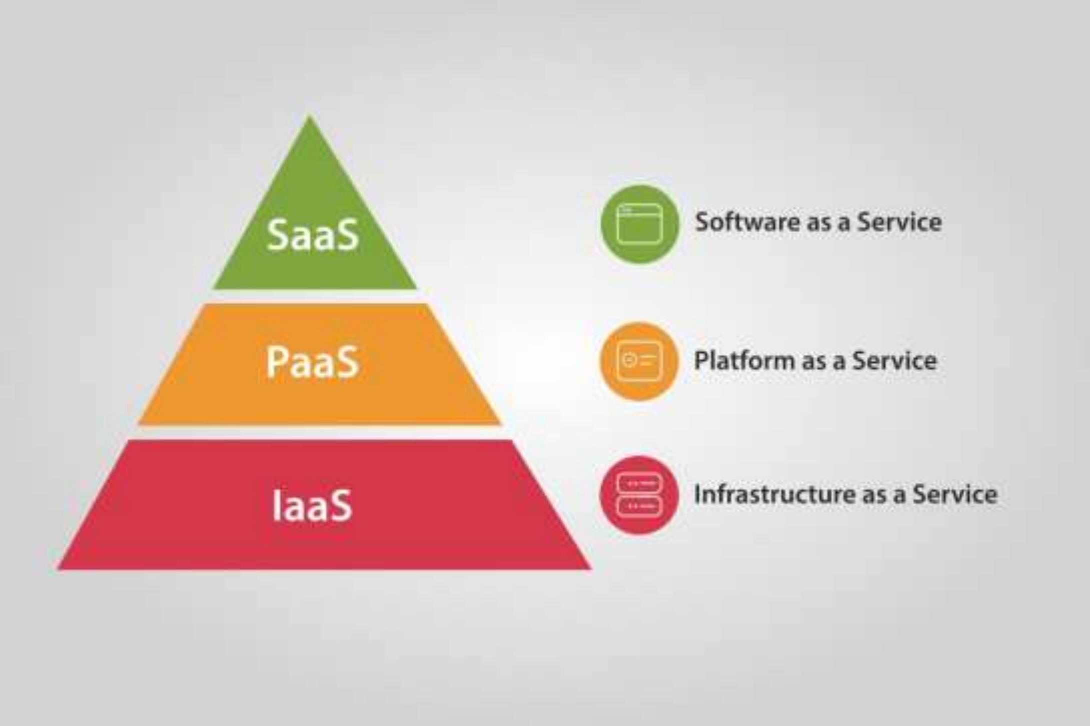
Major Cloud Providers

- **Amazon Web Services (AWS)**
  - Market leader, broadest service range
  - Compute (EC2), storage (S3), ML (SageMaker)
- **Microsoft Azure**
  - Strong in enterprise and Microsoft ecosystems
  - Good integration with Active Directory, Office
- **Google Cloud Platform (GCP)**
  - Known for AI/ML tools:
    GKE, Vertex AI, BigQuery

Cloud Abstraction Layers

- Cloud services are structured as layers:
  - **IaaS** – virtual machines and networks
  - **PaaS** – managed platforms for apps and databases
  - **SaaS** – fully finished services via UI or API

- The more abstraction,
  the less you manage,
  but also less control.



VLAIO TETRA MLOps4ECM

IaaS: Infrastructure as a Service

- You get virtual machines and networking,
  but manage what runs on them.

- Full flexibility, ideal for custom stacks or legacy support.
- You install software, configure services, and apply updates.

Examples:

- AWS EC2, Azure VMs, Google Compute Engine

PaaS: Platform as a Service

- Platform handles OS, runtime, scaling, networking.
- You deploy code or containers — less operational overhead.
- Great for APIs, web apps, and database-backed services.

Examples:

- AWS Elastic Beanstalk, Azure App Service, Google App Engine
- Managed SQL: AWS RDS, Azure SQL DB, Google Cloud SQL

SaaS: Software as a Service

- You just use the service — no code or deployment needed.
- Ideal for integrating common AI capabilities or tools.

Examples:

- OpenAI ChatGPT API
- AWS Rekognition
- Google Vision API

Managed Kubernetes Services

- Kubernetes = industry standard for orchestration
- But self-hosting is complex: certificates, control plane, networking
- Managed services handle the Kubernetes internals for you

Examples:

- AWS: **EKS**
- Azure: **AKS**
- Google Cloud: **GKE**

- You focus on YAML manifests, scaling rules, and deployments
- No need to provision VMs or worry about control plane

Serverless Container Platforms

- Serverless = no servers to manage
- You deploy code or containers, platform handles the rest

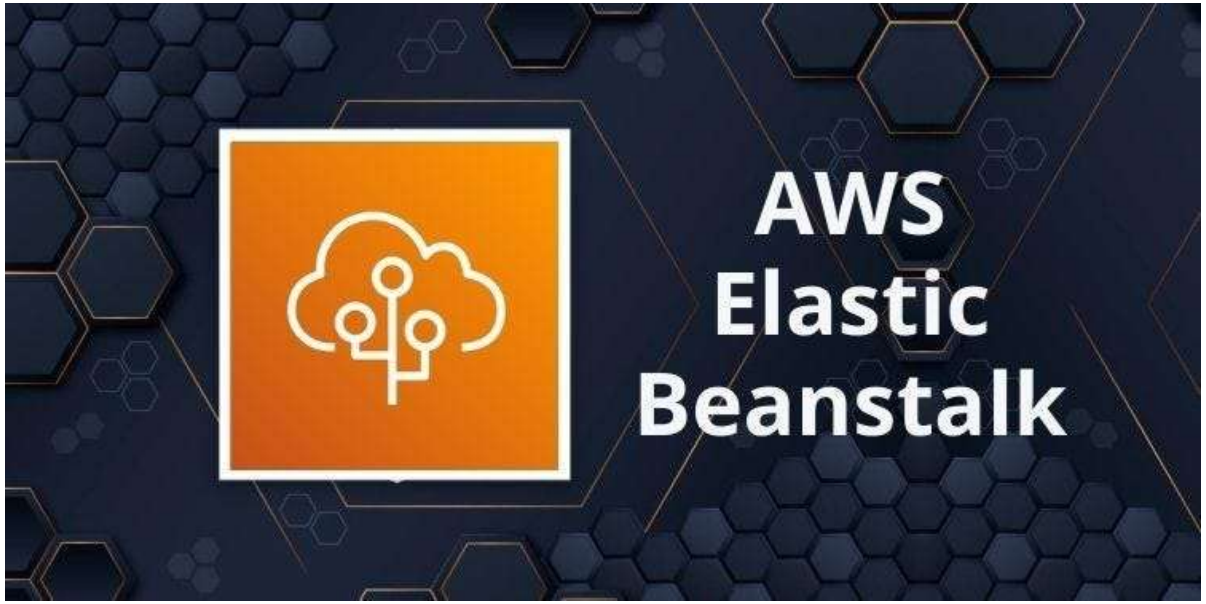
- Key benefit: **scale to zero**
  - No requests → zero cost
  - Auto-scale up on demand

PaaS: Simplifying Development

- Focus on **writing code**, not managing infrastructure
- Great for web apps, APIs, backends

- **Examples:**
  - AWS Elastic Beanstalk
  - Azure App Service
  - Google App Engine

- Auto-setup:
  - Runtime environment
  - Scaling and load balancing
  - Deployment from Git



PaaS for ML: SageMaker Example

- High-level ML platforms on cloud providers:
  - AWS SageMaker
  - Azure Machine Learning
  - Google Vertex AI

- Full ML lifecycle:
  - Data prep
  - Training (GPU support)
  - Model deployment
  - Monitoring and CI/CD pipelines

Amazon SageMaker

SageMaker Features

- **Studio**: Web IDE for ML
- **Data Wrangler**: Prepare datasets
- **Training**: Scalable, GPU-enabled
- **Autopilot**: AutoML for tabular data
- **Hyperparameter tuning**
- **Inference**: Real-time, batch, async
- **Model Monitor**: Drift detection
- **Pipelines, Feature Store, Experiments**

Amazon SageMaker

SaaS for ML and AI

- Highest abstraction: finished apps or APIs
- No model building — just call the API

Examples:

- AWS Rekognition, Polly, Comprehend
- Azure Cognitive Services
- Google Vision AI, Translation API
- OpenAI ChatGPT / GPT-4 API

SaaS: Benefits and Use Cases

- **Pretrained models** via HTTP API
  - No infra, no training, no deployment
  - You just send input and get predictions
- Great for:
  - Text summarization
  - Speech-to-text
  - Image tagging
  - Chatbots
- Usage-based billing
  (tokens, seconds, requests)

On-Premise Infrastructure

- Some companies prefer **full control**
- Run your own servers in data centers

- Use platforms like:
  - **Proxmox VE**
  - **VMware ESXi**
  - **Microsoft Hyper-V**

- Run VMs, containers (LXC),
  and clusters on owned hardware

Why Go On-Premise?

- **Data control**
  - Medical, legal, or regulated data
- **Security**
  - Air-gapped or private network systems
- **Performance**
  - Low-latency or large local datasets
- **Cost**
  - Long-running, predictable workloads
- **Existing infra or staff**
  - Skilled sysadmins already in place

Infrastructure as Code (IaC)

- Manual setup doesn’t scale — automation is key

- **IaC = Define infrastructure using config files**
- Code = truth; reusable, versioned, testable

- Tools:
  - **Terraform**: Define VMs, storage, networks (multi-cloud)
  - **Kubernetes YAML**: Declarative service definitions
  - **Argo CD / Flux**: GitOps for K8s (auto-sync from Git)

HashiCorp Terraform

Why IaC Matters

- Enables:
  - Automation across environments (dev/stage/prod)
  - Repeatable infrastructure
  - Safer changes (PRs, version control)

- Treat infrastructure like software:
  - Git history, CI/CD, testing
  - No snowflake servers

HashiCorp Terraform

Scalable Storage for Machine Learning: Object Storage

Why We Need Scalable Storage

- ML projects generate large, binary files:
  - Datasets, model checkpoints, logs, telemetry

- Git is not designed to handle:
  - Binary blobs, large files
  - High-frequency changes

- ML pipelines need storage that is:
  - Scalable, API-driven, binary-safe

What Doesn’t Belong in Git

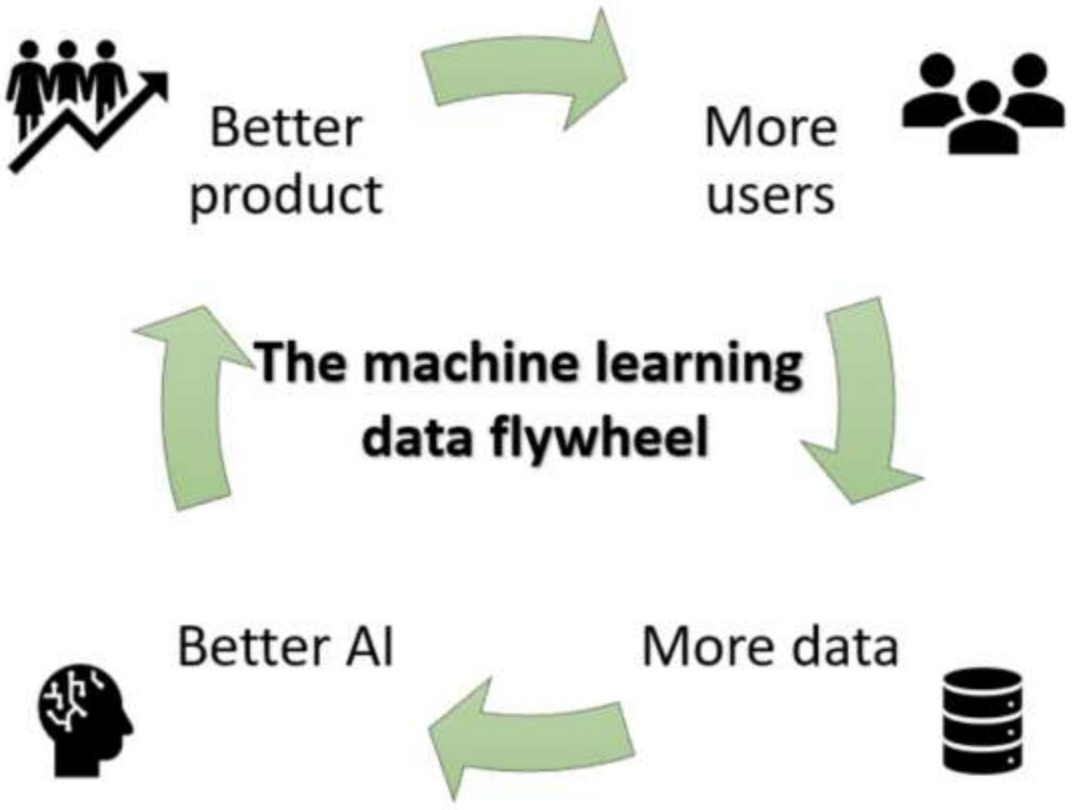
- Git is perfect for **source code**

- But not for:
  - Evolving datasets
  - Model binaries (.pt, .onnx, etc.)
  - Large log files and experiment outputs

- Binary files in Git:
  - Cannot be diffed or merged
  - Bloat the repo over time

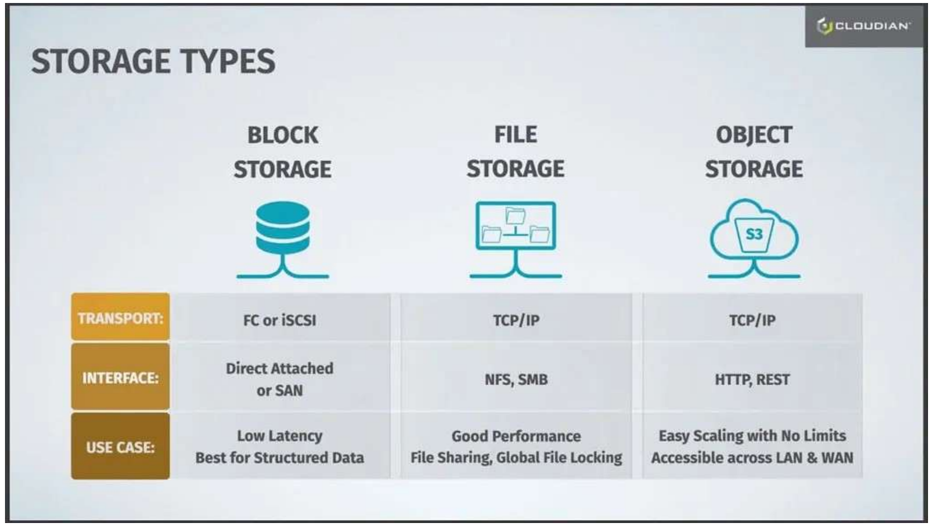
The Data Flywheel: Continuous Learning

- Production ML often supports **automated retraining**
- New data constantly collected via:
  - User interaction, sensors, logs…
- System needs to:
  - Ingest → Clean → Version → Retrain → Deploy
- All steps involve **binary files** too large for Git



What Is Object Storage?

- A scalable system for storing **unstructured binary data**
- Upload binary blobs (called **objects**)
- Each object has:
  - A unique key (like a filename)
  - Optional metadata (tags, content type)
- Read/write via simple HTTP API



CLOUDIAN

STORAGE TYPES

	BLOCK STORAGE	FILE STORAGE	OBJECT STORAGE
TRANSPORT:	FC or iSCSI	TCP/IP	TCP/IP
INTERFACE:	Direct Attached or SAN	NFS, SMB	HTTP, REST
USE CASE:	Low Latency Best for Structured Data	Good Performance File Sharing, Global File Locking	Easy Scaling with No Limits Accessible across LAN & WAN

Key Properties of Object Storage

- **Write-once, read-many** model
- Immutable uploads – **no in-place mutation**
- Designed for:
    - Durability (e.g. 11 nines)
    - High throughput
    - Distributed architectures
- **Ideal for ML**: logs, data, model weights

S3: The Industry Standard API

- AWS S3 launched in 2006

- Uses simple **HTTP methods**:
  - GET, PUT, DELETE, HEAD

- Other providers adopted the **S3 API**

→ S3 is the **de facto interface** for object storage today

Amazon S3

S3-Compatible Platforms

- Cloud:
  - **AWS S3, Azure Blob, Google Cloud Storage**
  - **Backblaze B2, Cloudflare R2, Wasabi**

- On-prem / self-hosted:
  - **MinIO**
  - **Ceph**

- Same code, different backends, just change the endpoint

Buckets and Objects

- Object storage is made of:
  - **Buckets** = top-level namespaces
  - **Objects** = binary files inside buckets

- Objects have unique **keys** (names)

- Metadata is stored alongside the object
  - Size, type, date, custom tags

HTTP Operations in S3

- PUT → Upload object or create bucket
- GET → Download object or list contents
- DELETE → Remove object or bucket
- HEAD → Read metadata only

→ **RESTful API**: standard HTTP verbs

Using S3 from Python: Boto3

```python
import boto3

s3 = boto3.client(
    's3',
    endpoint_url='https://s3.example.com',
    aws_access_key_id='ACCESS',
    aws_secret_access_key='SECRET',
)
s3.upload_file('model.pt', 'my-bucket', 'models/model.pt')
```

- Works with AWS or **any S3-compatible endpoint**
- Full CRUD (create-read-update-delete) API

Recommended File Formats for ML Artifacts

- File format matters for:
  - Storage cost
  - I/O performance
  - Portability & safety

- **Good formats are**:
  - ✅ Compressed
  - ✅ Portable
  - ✅ Safe to load

Avoid: raw CSVs, pickle, raw BMP/WAV

File Format Cheat Sheet

Artifact Type	Recommended Format(s)
Configs, logs	YAML/JSON + gzip, zstd
Tabular/time series	Parquet
Images	JPEG (lossy), PNG (lossless)
Audio	FLAC (lossless), MP3 (lossy)
Models	.pt, .onnx, .tar.gz

Versioning Strategies (S3)

Reproducibility depends on knowing
exactly what data + model was used.

1. Manual Naming

models/model_v1.2.3.pt

datasets/2024-05-01/images.parquet

- ✅ Flexible
- ❌ Error-prone at scale

Versioning Strategies (S3)

2. Native S3 Versioning

- Enable on bucket
- S3 keeps old versions when key is overwritten
- Basic protection against overwrites

3. Versioning Tools

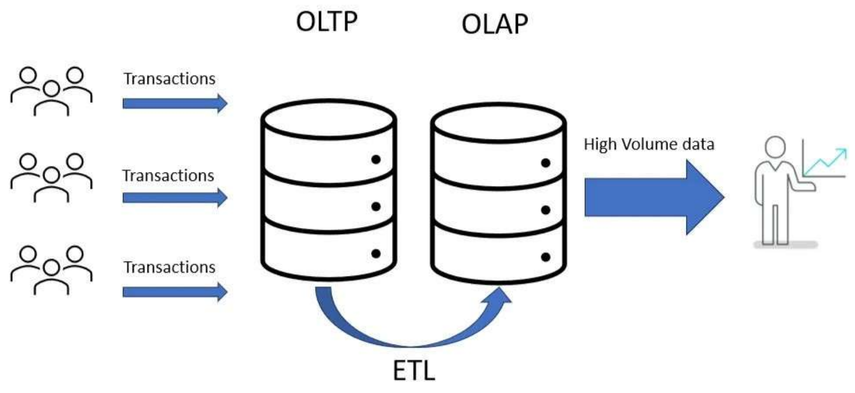
- **DVC**: Git-style versioning for data
- **LakeFS**: Branching + commits over S3 buckets

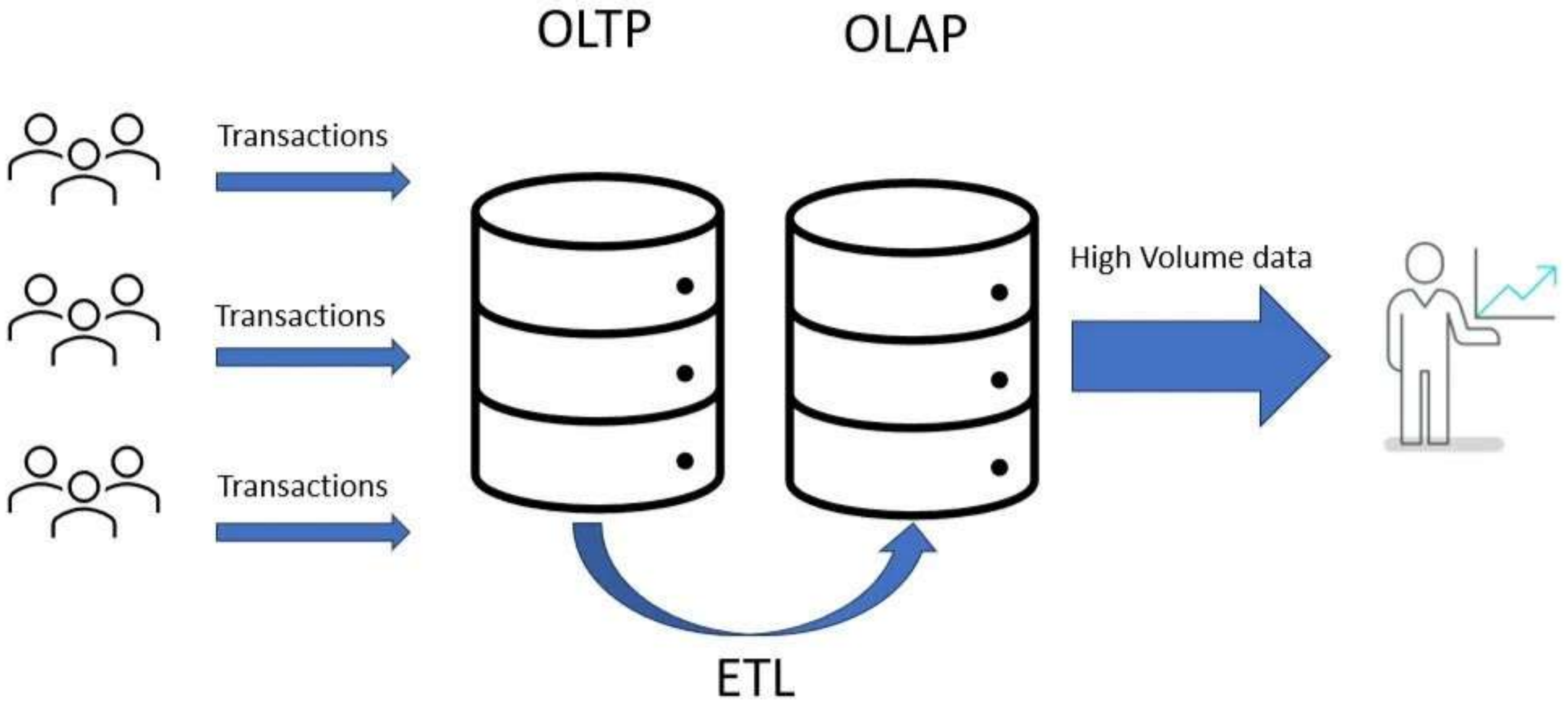
→ Best option for production systems

From OLTP to OLAP: Freezing Live Data

- Live systems → OLTP (e.g. PostgreSQL, InfluxDB)
- ML training → Needs **frozen snapshots**
- Store training data in **OLAP format** (e.g. Parquet on S3)

→ Ensures reproducibility




OLTP
OLAP
Transactions
Transactions
Transactions
High Volume data
ETL

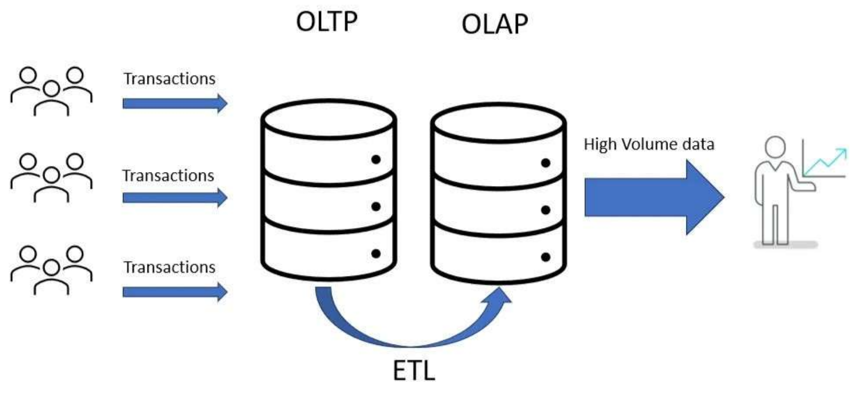
ETL to S3: The Batch Path

1. **Extract** from OLTP (e.g. PostgreSQL)

2. **Transform** → clean, enrich, normalize

3. **Load** into S3 as partitioned **Parquet** files
→ Enables analytics, training, debugging

Tools: Airflow, dbt, Spark, Polars, DuckDB, …



S3 as ML Data Backbone

Object storage like S3 is the **core layer** for ML data:

- ✅ Stores all binary artifacts
- ✅ Integrates with Python, CI, cloud
- ✅ Versioned, scalable, portable

→ A simple **foundation**
that scales with your ML system

Amazon S3

Continuous Integration and Continuous Deployment

Why Automate ML Workflows?

- ML projects often start with notebooks and ad-hoc scripts.
- **Manual steps**: training, saving models, uploading to servers…
- This works early on — but **doesn't scale**.

- Real ML systems must:
  - **Ingest** new data
  - **Train** and evaluate models
  - **Deploy**, **monitor**, and **iterate**

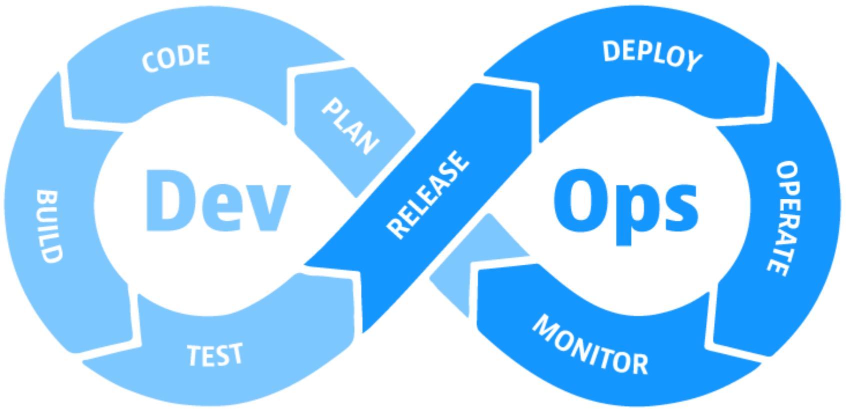
Manual Workflows Don’t Scale

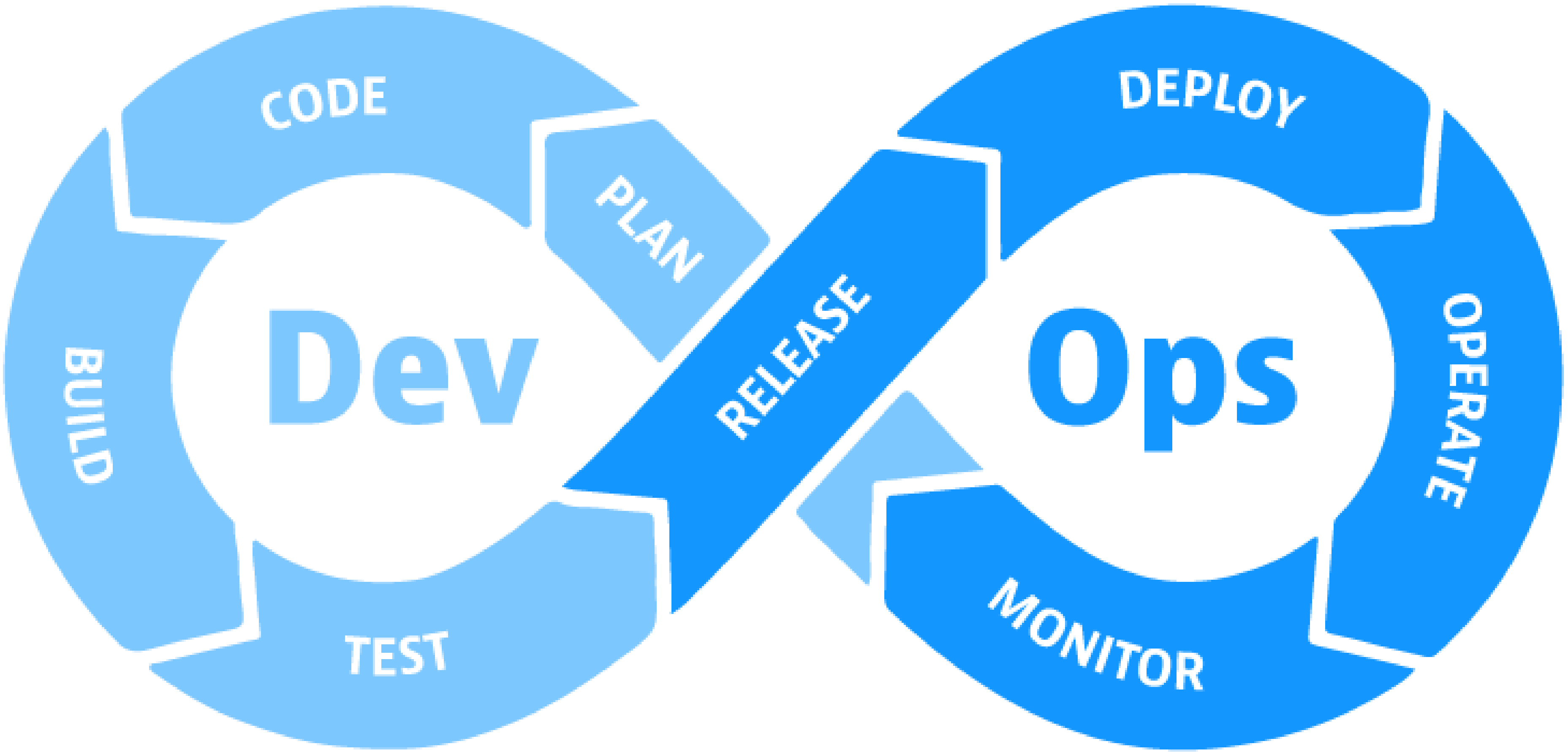
- Too many manual steps = **risk of errors**
- Common symptoms:
  - “It worked on my machine”
  - “Which model version is live?”
  - “Can we retrain from last month’s data?”
- We need **automation** to make ML:
  - Reliable
  - Reproducible
  - Maintainable

From Dev + Ops to DevOps

- Traditional teams:
  - Devs write code
  - Ops manage infrastructure

- Handoff problems:
  - Different goals (speed vs. stability)
  - No shared **responsibility**



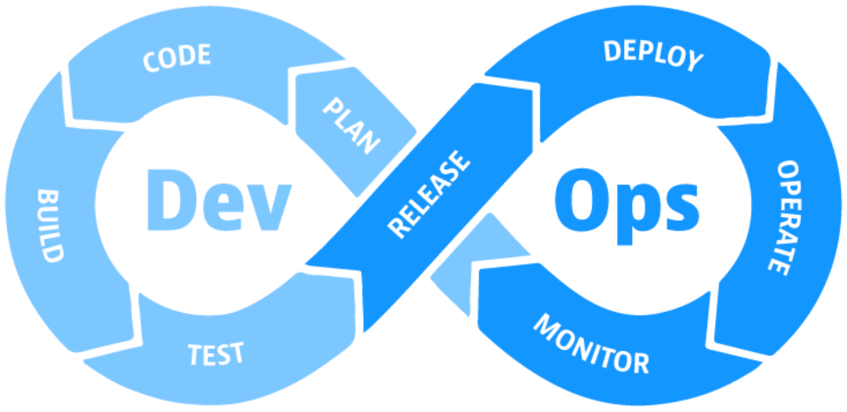

CODE
PLAN
BUILD
Dev
TEST
RELEASE
DEPLOY
Ops
OPERATE
MONITOR

The DevOps Mindset

- DevOps = break the wall between dev and ops

- Use **automation pipelines** to move from code → production
- Developers own what they deploy

- Core practices:
  - **CI** – test every change automatically
  - **CD** – deliver those changes to production

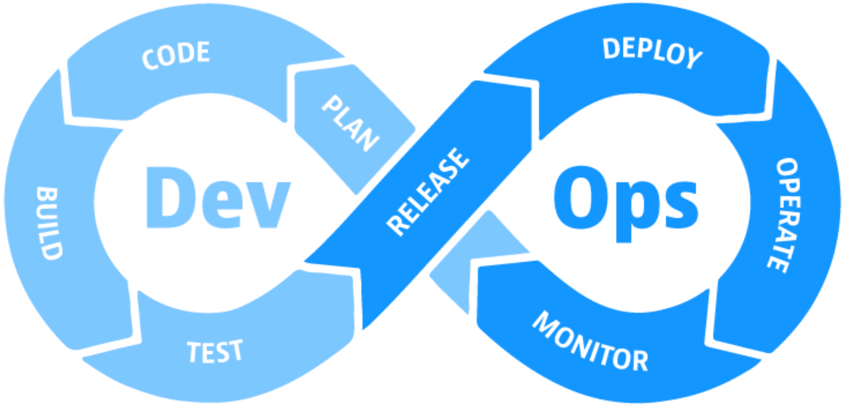


Continuous Integration (CI)

- CI = integrate code changes **frequently**

- Encourages:
  - Short-lived branches
  - Small, reviewable pull requests

- CI systems automatically:
  - Install dependencies
  - Run tests, linters, builds
  - Block broken code from merging



CI = Trust in your shared codebase

Continuous Delivery & Deployment (CD)

- CD = deliver new code **safely and often**
- Small, **incremental** changes → fewer surprises

Two main modes:

- **Continuous Delivery**: automated pipeline, but human deploys
- **Continuous Deployment**: fully automatic deploy after tests pass

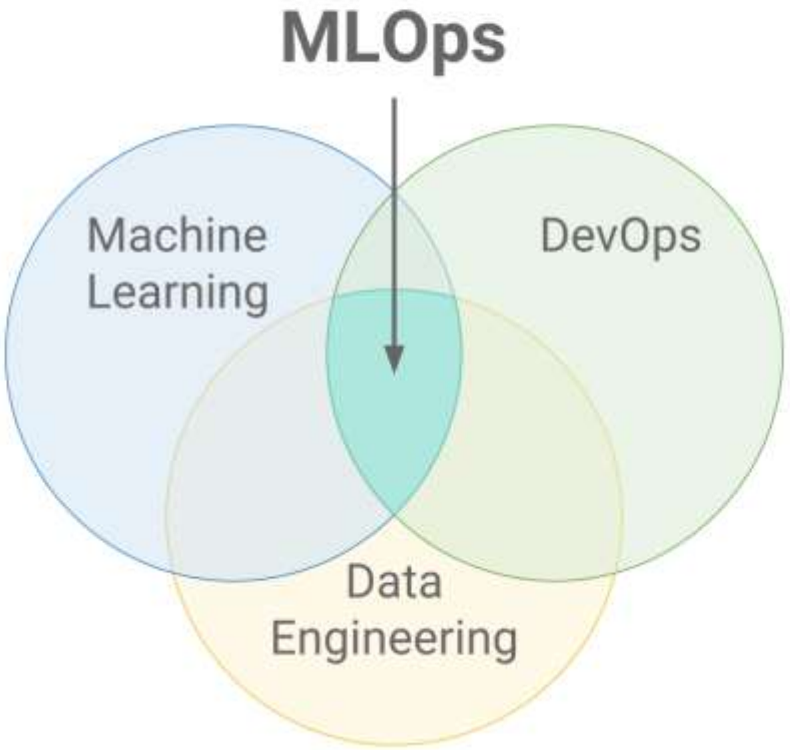
Deployments become **routine**, not scary

Why CI/CD Matters for ML

- **ML = more complex** than normal software
  - Code + data + models + infrastructure

- CI/CD helps:
  - Test training pipelines
  - Validate model performance
  - Automate retraining and deployment

→ CI/CD is **essential for reliable ML systems**



GitHub Actions for CI/CD

What Is GitHub Actions?

- GitHub’s built-in automation system
  - Runs pipelines **directly from your repo**

- Triggered by:
  - Code pushes
  - Pull requests
  - Manual or scheduled events

- Great for ML workflows:
  - No setup needed
  - Fully version-controlled
  - Integrated with GitHub

Workflows: Structure Overview

A GitHub Actions workflow has:

- **Triggers**: when to run the workflow
- **Jobs**: logical units of work (can run in parallel)
- **Steps**: actual commands or reusable actions

Files live in:

.github/workflows/*.yml

Workflows evolve with your codebase.

Example: Minimal Test Workflow

.github/workflows/test.yml

```yaml
name: Run tests on main branch
on:
  push:
    branches: [main]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.13'
      - run: pip install -r requirements.txt
      - run: pytest .
```

Triggers: When Workflows Run

GitHub Actions workflows can be **triggered** by:

- **push**: On code pushes
- **pull_request**: On PR creation or updates
- **schedule**: Cron jobs (e.g. nightly retraining)
- **workflow_dispatch**: Manual runs via UI/API

Each trigger is useful for different stages of the ML lifecycle.

Trigger: Code Push

```yaml
on:
  push:
    branches: [main]
```

- Run when code is pushed to main
- Ideal for: **deployments, building software**

⚠️ Use with care — not all pushes should auto-deploy!

Trigger: Pull Request

```yaml
on:
  pull_request:
```

- Best for **tests, linting, and validation**
- Enforce with **branch protection rules**
  - Blocks merging until all checks pass

✅ Prevents broken code from reaching main

Trigger: Cron Schedule

```yaml
on:
  schedule:
    - cron: '0 0 * * *'
```

- Run workflows on a **recurring basis**
- Examples:
  - Nightly model retraining
  - Weekly data cleanup
  - Periodic evaluations

Runs at midnight UTC in this example.

Trigger: Manual (workflow_dispatch)

```yaml
on:
  workflow_dispatch:
    inputs:
      environment:
        description: 'Target env'
        required: true
        default: 'staging'
```

- Adds a **Run Workflow** button in GitHub UI
- Use for:
  - Manual deployment
  - One-off scripts
  - Controlled experiments

Jobs and Runners

- A **job** = a group of steps, run on a virtual machine (runner)

- Runners can be:
  - **GitHub-hosted** (default, auto-provisioned)
  - **Self-hosted** (GPU, on-premise, cloud)

```
runs-on: ubuntu-latest
```

Parallel and Dependent Jobs

Jobs can run in parallel:

```
jobs:
  test: ...
  lint: ...
```

Or one after another:

```
jobs:
  build: ...
  deploy:
    needs: build
```

Use **needs:** to control job order.

Self-Hosted Runners

```
runs-on: [self-hosted, gpu]
```

Use your own machines for:

- GPU access
- Private data or networks
- Custom dependencies
- CI on air-gapped systems

Register runners in repo or organization settings.

Steps: The Core Building Blocks

- Steps run **inside jobs**, in order
- Share the same environment (files, variables)
- Can be:
  - Shell commands
  - GitHub Actions

```yaml
steps:
  - run: pip install -r requirements.txt
  - run: python train.py
```

Using Reusable Actions

Use community-built actions from the GitHub Marketplace:

```yaml
- uses: actions/checkout@v4
- uses: actions/setup-python@v5
  with:
    python-version: '3.10'
```

Other useful actions:

- actions/cache
- actions/upload-artifact
- docker/build-push-action
- aws-actions/configure-aws-credentials

→ Browse: github.com/marketplace?type=actions hogeschool vives KU LEUVEN

Actions

Automate your workflow from idea to production

Filter: All | By: All creators | Sort: Popularity

TruffleHog OSS — Action
Find and verify leaked credentials in your source code

Metrics embed — Action
An infographics generator with 40+ plugins and 300+ options to display stats about your GitHub account

yq - portable yaml processor — Action
create, read, update, delete, merge, validate and do more with yaml

Super-Linter — Action
Super-linter is a ready-to-run collection of linters and code analyzers, to help validate your source code

Gosec Security Checker — Action
Runs the gosec security checker

Rebuild Armbian and Kernel — Action
Support Amlogic, Rockchip and Allwinner boxes

OpenCommit — improve commits with AI... — Action
Replaces lame commit messages with meaningful AI-generated messages when you push to remote

Checkout — Action
Checkout a Git repository at a particular version

SSH Remote Commands — Action
Executing remote ssh commands

GitHub Pages action — Action
GitHub Actions for GitHub Pages 🚀 Deploy static files and publish your site easily. Static-Site-Generators-friendly

Environment Variables

Use env: to define variables:

```yaml
jobs:
  deploy:
    env:
      ENVIRONMENT: production
```

Available to all steps in the job:

```yaml
- run: echo "Deploying to $ENVIRONMENT"
```

⚠️ Environment variables are **not secure** by default

GitHub Secrets

For credentials and tokens, use **GitHub Secrets**:

1. Go to **Settings → Secrets → Actions**
2. Add AWS_ACCESS_KEY_ID, etc.

In workflow:

```yaml
env:
  AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
```

→ Never commit credentials in code!

Example: Upload to S3 with Secrets

```
- name: Upload to S3
  run: aws s3 cp model.pkl s3://my-ml-models/
  env:
    AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
    AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

- Keeps credentials **safe**
- Works on any runner
- Essential for ML workflows that deploy or store artifacts

Treat ML Like Engineering

- CI/CD is your **first step** toward ML engineering
- It brings structure, speed, and reliability to fast-moving ML workflows
- From experiment to automation: **ML needs DevOps too**

Data Pipelines and Orchestration Frameworks

Why We Need Orchestration

- ML systems aren't just about models,
  they involve **data, logic, dependencies, and time**.

- Scripts break when pipelines become:
  - Multi-step, data-triggered
  - Long-running, failure-prone

- Orchestration frameworks help you build **reliable, maintainable workflows**.

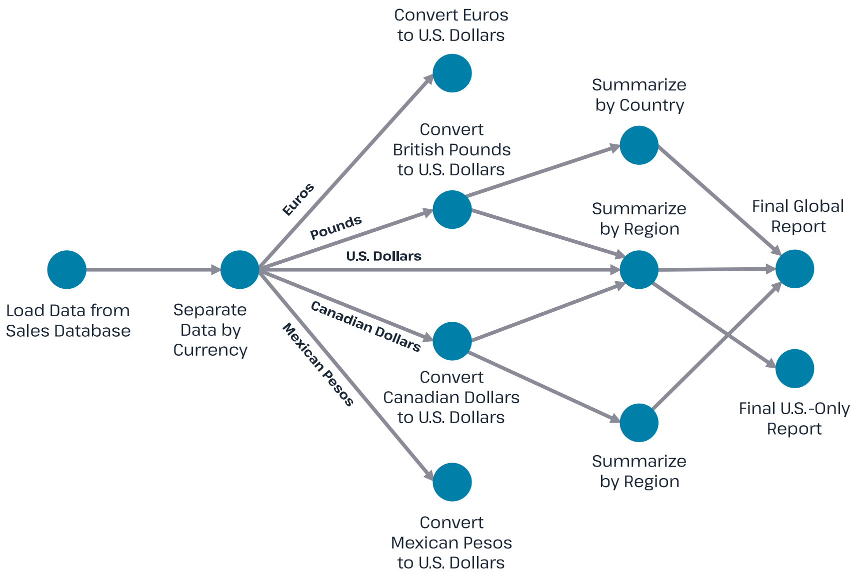
Beyond CI/CD: A Different Kind of Automation

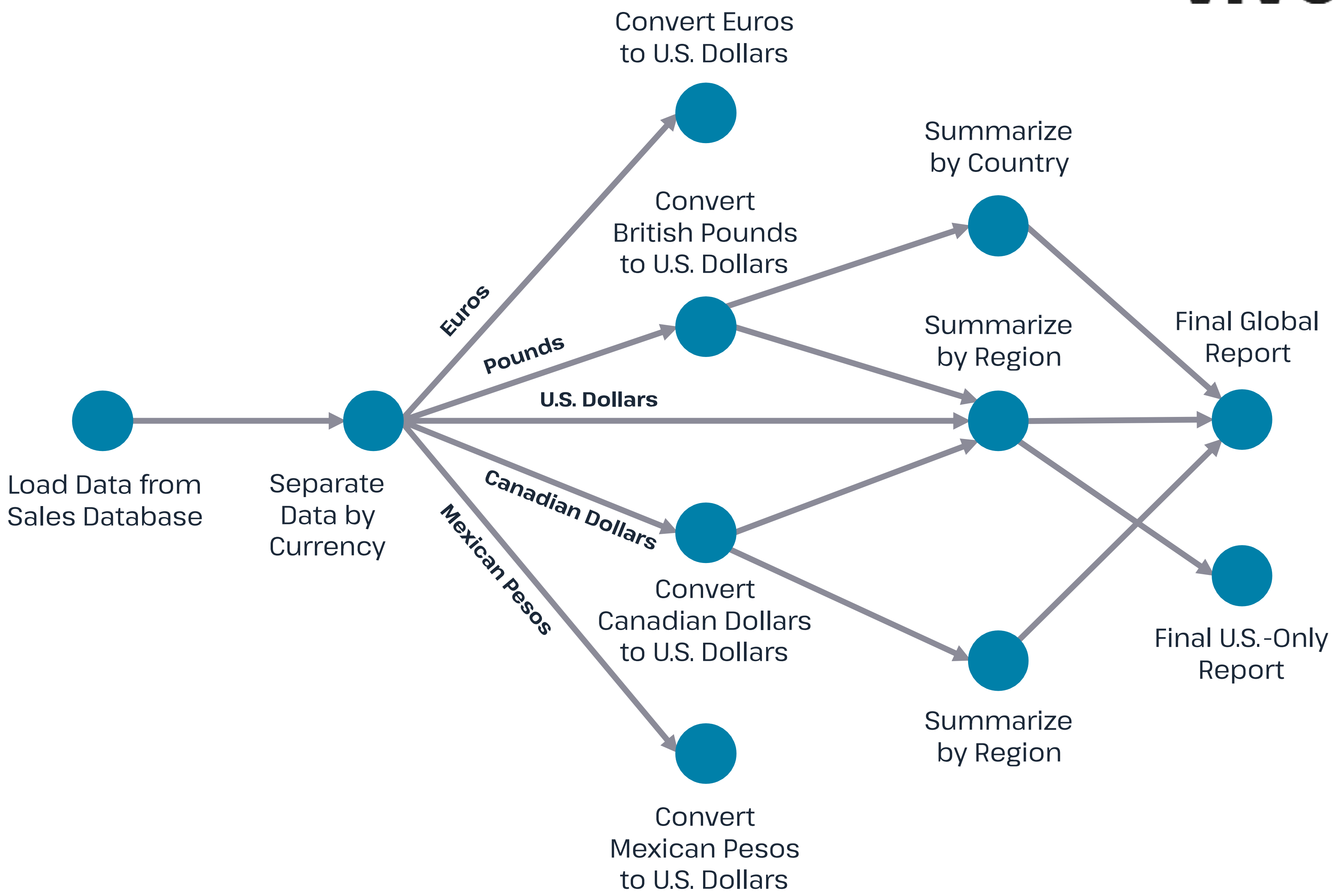
- CI/CD tools react to **code changes** (e.g. Git push, PR).
- Orchestration tools react to **data or time** (e.g. new files, daily run).

- CI/CD is great for:
  - Testing code
  - Building containers
  - Deploying services
- Orchestration is better for:
  - Scheduling workflows
  - Managing task dependencies
  - Handling retries, monitoring, and lineage

DAGs: The Backbone of Orchestration

- Orchestrators define workflows as **DAGs** — Directed Acyclic Graphs.
- Each task is a node, dependencies are edges.
- DAGs:
  - Ensure correct **execution order**
  - Support **parallelism**
  - Enable resumable, observable workflows




Convert Euros to U.S. Dollars
Summarize by Country
Convert British Pounds to U.S. Dollars
Euros
Pounds
Summarize by Region
Final Global Report
U.S. Dollars
Load Data from Sales Database
Separate Data by Currency
Canadian Dollars
Mexican Pesos
Convert Canadian Dollars to U.S. Dollars
Final U.S.-Only Report
Summarize by Region
Convert Mexican Pesos to U.S. Dollars

Tool Spotlight: Apache Airflow

- ✅ Proven and widely adopted
- ✅ Strong scheduling and monitoring
- ⚠️ Rigid and boilerplate-heavy

```python
@dag(schedule_interval="@daily")
def pipeline():
    clean_data() >> train_model()
              >> evaluate_model()
```

Great for: **structured, schedule-driven pipelines in stable environments**

Tool Spotlight: Dagster

- ✅ Asset-centric approach
- ✅ Strong modularity, type safety, and lineage
- ⚠️ Some newer concepts and tooling curve

```python
@asset
def trained_model(cleaned_data): ...
```

Great for: **modern ML projects with evolving assets and structure**

Tool Spotlight: Prefect

- ✅ Easy to adopt, very Pythonic
- ✅ Supports dynamic, runtime-generated DAGs
- ⚠️ Less opinionated; cloud features gated

```
@flow
def my_pipeline():
    model = train_model(clean_data())
    evaluate(model)
```

Great for: **flexible research workflows, rapid iteration**

Who Does What: CI/CD vs. Orchestration

Use Case	CI/CD Tools	Orchestration Tools
Test code	✅	❌
Deploy services	✅	❌
Ingest new data	❌	✅
Run nightly training	❌	✅
Retry failed steps	⚠️ Limited	✅ Robust
Monitor pipelines	⚠️ Basic logs	✅ Full UI

You've Reached the End of the Pipeline

Final Thoughts: MLOps at Scale

- ML in production means **software, data and automation**.
  - You don’t need all tools at once — **start small** and scale up.

- Choose the tools that match your **team size**, **data complexity**, and **maturity**.

- Containers, object storage, CI/CD and data pipelines are the **foundation** of robust ML systems.

Thank You & Good Luck!

🎓 You’ve reached the end of the workshop — well done!

🚀 Good luck with your future **machine learning adventures**!