MLOps

Lecture Notes

Master M2 — 2025–2026

Yaë Ulrich Gaba

"Artificial intelligence is the new electricity."
— Andrew Ng

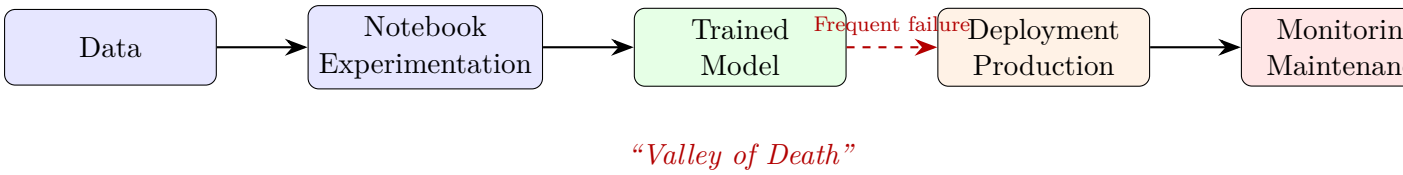
March 25, 2026

Contents

Preface

Why 90 % of ML Projects Fail in Deployment

A recurring finding across industry is alarming: according to Gartner (2022), only 54 % of machine learning (ML) models make it from prototype to production, and among those that do, many are retired within the first few months. VentureBeat reported in 2019 that 87 % of data science projects never reach production. This is not a modelling problem—it is an **engineering** problem.



The main causes of these failures are:

- **Technical debt**: ML code accumulates hidden complexity (data dependencies, feedback loops, unversioned configurations) [10].
- **Irreproducibility**: impossible to recreate a colleague's results, even with the same code, because the environment, data, or hyperparameters differ.
- **Organizational gap**: data scientists work in isolated notebooks, ML engineers do not understand modelling choices, and ops teams do not know how to monitor a model.
- **Lack of monitoring**: a deployed model without surveillance silently drifts (data drift, concept drift).
- **No reproducible pipeline**: every step (preprocessing, training, evaluation) is executed manually.

What MLOps Solves

MLOps (Machine Learning Operations) is a set of practices that unify ML development (Dev) and operations (Ops) to automate, standardize, and make reliable the lifecycle of models. It is the equivalent of DevOps for machine learning, with specific challenges: data versioning, experiment tracking, result reproducibility, drift monitoring.

Course Organization

This course is designed for Master-level students in data science, machine learning, or applied mathematics. It assumes a solid foundation in Python, familiarity with the Linux/Unix command line, and knowledge of machine learning.

Each chapter follows a consistent pedagogical structure:

1. **Motivation and concepts**: why this tool or practice is necessary.
2. **Hands-on tutorial**: working code that the student can run on their own machine.
3. **Best practices and anti-patterns**: what to do and what to avoid.
4. **Tool comparison**: comparison tables when multiple solutions exist.
5. **Mini-project**: integrative exercise using tools from previous chapters.
6. **Exercises**: graded by difficulty (★ setup/config, ★★ pipeline building, ★★★ end-to-end project).
7. **Cheatsheet**: summary of essential commands.

Happy reading and happy deploying!

Chapter 1

Introduction to MLOps — From Notebook to Production

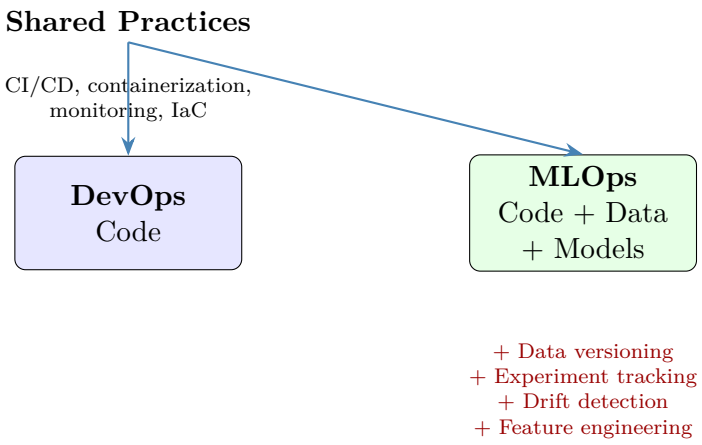
1.1 What is MLOps?

MLOps refers to the set of practices, tools, and cultural principles aimed at deploying and maintaining machine learning models in production reliably and efficiently. The term is a portmanteau of *Machine Learning* and *Operations*, by analogy with *DevOps* from software engineering.

Definition 1.1 (MLOps)**.** MLOps is the discipline that applies DevOps principles (continuous integration, continuous deployment, monitoring, automation) to the lifecycle of machine learning models, adding specificities related to data, experiments, and model drift.

1.1.1 DevOps vs MLOps

Traditional DevOps manages code. MLOps manages code **and** data **and** models—three artifacts whose interaction creates additional complexity.

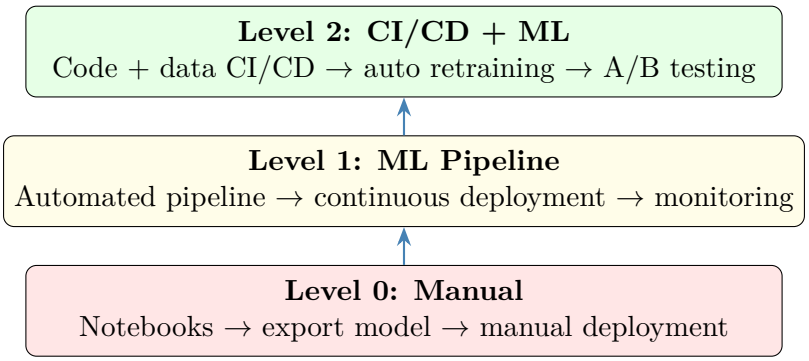


Remark 1.2. The fundamental difference is that the behaviour of an ML system depends not only on the code but also on the data. The same code trained on different data produces a different model. This requires versioning data just like code.

1.2 MLOps Maturity Levels

Google proposed three maturity levels for ML systems in production:

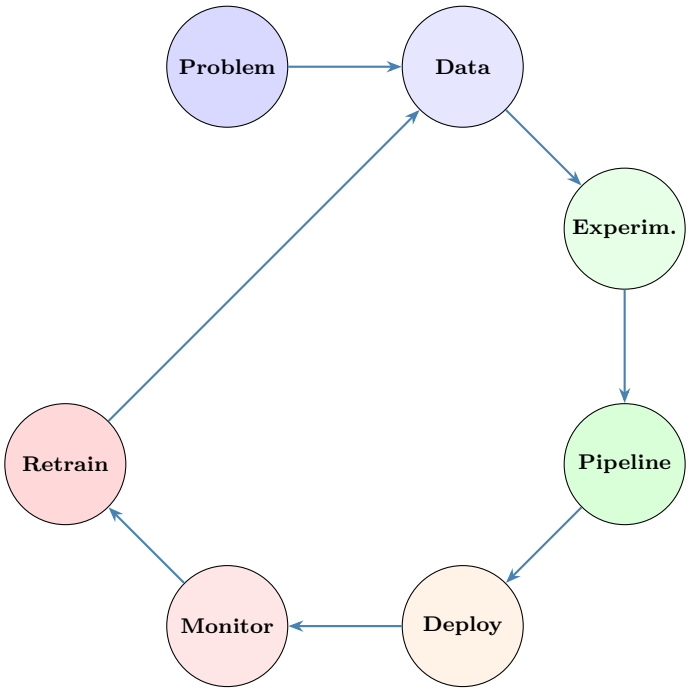
Level	Name	Description
0	Manual	Entirely manual process. Notebooks, no pipeline. Complete separation between data scientists and ops.
1	ML Pipeline	Automated training pipeline. Continuous model deployment. Basic monitoring.
2	CI/CD Pipeline	CI/CD pipeline for ML code. Automated training triggered by data. Automatic retraining on drift.



1.3 The ML Lifecycle

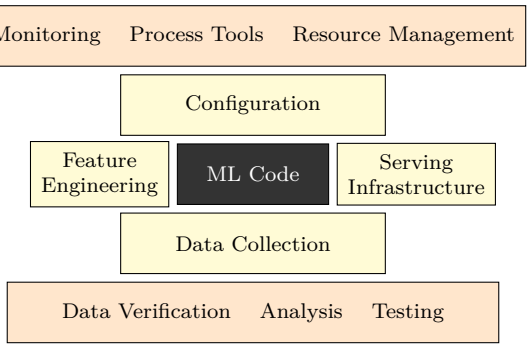
An ML project in production goes through several phases that form a cycle (not a linear sequence):

1. **Problem definition**: what is the business objective? What is the success criterion?
2. **Data collection and preparation**: acquisition, cleaning, labelling, feature engineering.
3. **Experimentation**: model selection, hyperparameter optimization, evaluation.
4. **Pipeline development**: transforming the notebook into modular, tested, versioned code.
5. **Deployment**: putting the model into production (API, batch, edge).
6. **Monitoring**: tracking performance, detecting drift, alerts.
7. **Retraining**: updating the model when data or performance evolve.



1.4 Technical Debt in ML

Sculley et al. [10] identified that ML code represents only a small fraction of a production ML system. The rest is infrastructure:



The notebook trap

Jupyter notebooks are excellent for exploration but poor for production. Common problems:

- Non-linear execution (cells run out of order)
- Hidden state (global variables that persist)
- No unit tests
- Difficult to version (large JSON files)
- No modularity (functions and classes not reusable)

1.5 MLOps System Architecture

A mature MLOps system includes the following components:

MLOps system components

Code management Git, GitHub/GitLab

Data management DVC, Delta Lake, LakeFS

Environment management Conda, pip, venv, Poetry

Experiment tracking MLflow, Weights & Biases, Neptune

Training pipeline Airflow, Prefect, Kubeflow Pipelines

Feature store Feast, Tecton, Hopsworks

Containerization Docker, Kubernetes

Serving FastAPI, TorchServe, TensorFlow Serving, Triton

CI/CD GitHub Actions, GitLab CI, Jenkins

Monitoring Prometheus, Grafana, Evidently, WhyLabs

1.6 Tutorial: From Notebook to Modular Script

1.6.1 The starting notebook

Consider a typical classification notebook:

Typical notebook (anti-pattern)

```
# Cell 1
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Cell 2
df = pd.read_csv("data.csv")
X = df.drop("target", axis=1)
y = df["target"]

# Cell 3
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Cell 4
model = RandomForestClassifier(n_estimators=100, random_state=42)
```

```
model.fit(X_train, y_train)
print(f"Accuracy: {accuracy_score(y_test, model.predict(X_test)):.4f}")
```

1.6.2 The modular script

Recommended project structure

```
my_ml_project/
  src/
    __init__.py
    data.py          # Loading and preprocessing
    model.py         # Model definition
    train.py         # Training loop
    evaluate.py      # Evaluation metrics
  tests/
    test_data.py
    test_model.py
  configs/
    config.yaml      # Hyperparameters
  requirements.txt
  Makefile
  README.md
```

src/data.py — Data module

```python
"""Data loading and preprocessing module."""
import pandas as pd
from sklearn.model_selection import train_test_split


def load_data(path: str) -> pd.DataFrame:
    """Load dataset from CSV file."""
    df = pd.read_csv(path)
    return df


def split_data(
    df: pd.DataFrame,
    target_col: str = "target",
    test_size: float = 0.2,
    random_state: int = 42,
) -> tuple:
    """Split data into train and test sets."""
    X = df.drop(target_col, axis=1)
    y = df[target_col]
    return train_test_split(
        X, y, test_size=test_size, random_state=random_state
```

```
    )
```

src/train.py — Training script

```python
"""Training script with configuration."""
import argparse
import yaml
import joblib
from data import load_data, split_data
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score


def train(config: dict) -> None:
    """Train model with given configuration."""
    df = load_data(config["data"]["path"])
    X_train, X_test, y_train, y_test = split_data(
        df,
        target_col=config["data"]["target_col"],
        test_size=config["data"]["test_size"],
    )

    model = RandomForestClassifier(**config["model"])
    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    print(f"Accuracy: {acc:.4f}")

    joblib.dump(model, config["output"]["model_path"])
    print(f"Model saved to {config['output']['model_path']}")


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--config", default="configs/config.yaml")
    args = parser.parse_args()

    with open(args.config) as f:
        config = yaml.safe_load(f)

    train(config)
```

configs/config.yaml

```yaml
data:
  path: "data/dataset.csv"
  target_col: "target"
```

```yaml
  test_size: 0.2

model:
  n_estimators: 100
  max_depth: 10
  random_state: 42

output:
  model_path: "models/rf_model.joblib"
```

Makefile

```
.PHONY: install train test clean

install:
	pip install -r requirements.txt

train:
	python src/train.py --config configs/config.yaml

test:
	pytest tests/ -v

clean:
	rm -rf models/*.joblib __pycache__ .pytest_cache
```

1.7 Best Practices and Anti-patterns

Fundamental principles

1. **Separation of concerns**: data, model, training, evaluation in separate modules.
2. **Externalized configuration**: hyperparameters in YAML files, not hard-coded.
3. **Reproducibility by default**: fix random seeds, version data and code.
4. **Automated tests**: test data loading, tensor shapes, convergence on a small sample.
5. **Automation**: Makefile or scripts for all repetitive operations.

Absolutely avoid

- Hardcoding absolute paths: `/home/alice/data/train.csv`
- Committing large datasets to Git

- Committing secrets (API keys, passwords)
- Working exclusively in notebooks without modular code
- Ignoring tests because “it’s ML”
- Deploying a model without monitoring

1.8 Mini-project: Structuring an ML Project

Mini-project 1: From notebook to structured project

1. Create the recommended directory structure.
2. Transform the classification notebook above into separate Python modules.
3. Write a `config.yaml` file for hyperparameters.
4. Create a `Makefile` with targets `install`, `train`, `test`, `clean`.
5. Write at least two unit tests (one for data loading, one for model output shape).
6. Verify that `make train` works end-to-end.

1.9 Exercises

Exercise 1.1 (★ — Project structure)**.** Create an ML project skeleton for a regression problem with the recommended structure. Include a `README.md` describing the project and a `.gitignore` adapted for ML (ignore `data/`, `models/`, `*.pyc`, `.ipynb_checkpoints/`).

Exercise 1.2 (★★ — Notebook refactoring)**.** Take an existing Jupyter notebook (your own or a Kaggle notebook) and transform it into a structured project:

1. Identify reusable functions
2. Create the corresponding Python modules
3. Externalize the configuration
4. Write tests for critical functions

Exercise 1.3 (★★★ — Complete project)**.** Build a complete ML project for text classification (e.g., sentiment analysis on the IMDb dataset):

1. Complete modular structure
2. YAML configuration
3. Unit tests with `pytest`
4. Training script with command-line arguments

5. Automatic metric reporting (precision, recall, F1)
6. Complete `Makefile`

1.10 Cheatsheet

Chapter 1 Summary

Concept	Key Point
MLOps	DevOps + data + models
Maturity levels	0 (manual) → 1 (pipeline) → 2 (CI/CD)
Technical debt	ML code is a small fraction of the system
Notebook anti-pattern	Hidden state, no tests, no modularity
Project structure	`src/`, `tests/`, `configs/`, Makefile
External config	YAML for hyperparameters

Chapter 2

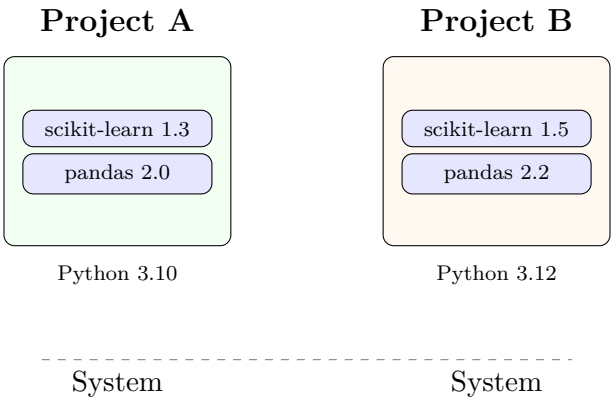
Environments and Dependency Management

2.1 Why Isolate Environments?

A classic problem in ML: "It works on my machine!" The causes:

- Different Python versions
- Incompatible library versions
- Missing system dependencies (CUDA, cuDNN)
- Unconfigured environment variables

Definition 2.1 (Virtual environment)**.** A **virtual environment** is an isolated directory containing a specific Python installation and a set of packages, independent of the system and other projects.



2.2 venv — Python's Built-in Tool

`venv` is Python's standard module for creating virtual environments. It is simple, lightweight, and requires no additional installation.

Essential venv commands

```
# Create a virtual environment
python -m venv .venv

# Activate (Linux/macOS)
source .venv/bin/activate

# Activate (Windows)
.venv\Scripts\activate

# Install packages
pip install numpy pandas scikit-learn

# Save dependencies
pip freeze > requirements.txt

# Recreate the environment
pip install -r requirements.txt

# Deactivate
deactivate
```

pip freeze pitfalls

`pip freeze` includes **all** transitive dependencies, making the file fragile. Prefer:

- Manually maintaining a `requirements.txt` with direct dependencies and their versions.
- Using `pip-tools` to generate a locked `requirements.txt` from a `requirements.in`.

2.2.1 pip-tools: dependency locking

pip-tools workflow

```
# Install pip-tools
pip install pip-tools

# requirements.in (direct dependencies only)
# numpy>=1.24
# pandas>=2.0
# scikit-learn>=1.3

# Compile (resolve and lock)
pip-compile requirements.in -o requirements.txt

# Install exactly the locked versions
```

```
pip-sync requirements.txt
```

2.3 Conda — Environments with System Dependencies

Conda manages not only Python packages but also system libraries (CUDA, MKL, OpenSSL). It is the preferred tool for ML projects requiring GPUs.

Essential Conda commands

```
# Create an environment
conda create -n ml-project python=3.11

# Activate
conda activate ml-project

# Install packages
conda install numpy pandas scikit-learn
conda install -c pytorch pytorch torchvision  # PyTorch channel

# Export the environment
conda env export > environment.yml

# Export without OS-specific dependencies
conda env export --from-history > environment.yml

# Recreate the environment
conda env create -f environment.yml

# List environments
conda env list

# Remove an environment
conda env remove -n ml-project
```

environment.yml

```
name: ml-project
channels:
  - pytorch
  - conda-forge
  - defaults
dependencies:
  - python=3.11
  - numpy=1.26
  - pandas=2.1
  - scikit-learn=1.3
```

```
  - pytorch=2.1
  - pip:
    - mlflow>=2.8
    - wandb>=0.16
```

Conda vs pip

- Use Conda for system dependencies (CUDA, compilers).
- Use pip inside a Conda environment for pure Python packages.
- **Never** mix `conda install` and `pip install` for the same package—Conda will not see packages installed by pip and vice versa.

2.4 Poetry — Modern Dependency Management

Poetry is a modern dependency manager that combines package management, version locking, and package publishing in a single tool.

Poetry workflow

```
# Install Poetry
curl -sSL https://install.python-poetry.org | python3 -

# Create a new project
poetry new ml-project
cd ml-project

# Add dependencies
poetry add numpy pandas scikit-learn
poetry add --group dev pytest black mypy

# Install dependencies
poetry install

# Run in the environment
poetry run python train.py
poetry run pytest tests/

# Export to requirements.txt (compatibility)
poetry export -f requirements.txt -o requirements.txt
```

pyproject.toml (excerpt)

```
[tool.poetry]
name = "ml-project"
version = "0.1.0"
```

```
description = "ML classification project"

[tool.poetry.dependencies]
python = "^3.10"
numpy = "^1.26"
pandas = "^2.1"
scikit-learn = "^1.3"

[tool.poetry.group.dev.dependencies]
pytest = "^7.4"
black = "^23.10"
mypy = "^1.6"
```

2.5 Tool Comparison

Criterion	venv+pip	Conda	Poetry	uv
Built into Python	Yes	No	No	No
System deps	No	Yes	No	No
Locking	pip-tools	Yes	Yes	Yes
GPU/CUDA	Manual	Yes	Manual	Manual
Speed	Fast	Slow	Medium	Very fast
Pkg publishing	No	No	Yes	No
Learning curve	Low	Medium	Medium	Low

2.6 uv — The Fast New Tool

uv is a Python package manager written in Rust, compatible with pip, extremely fast (10–100× faster than pip).

uv workflow

```
# Install uv
curl -LsSf https://astral.sh/uv/install.sh | sh

# Create environment and install
uv venv
source .venv/bin/activate
uv pip install numpy pandas scikit-learn

# Or all at once
uv pip install -r requirements.txt

# Locking
uv pip compile requirements.in -o requirements.txt
uv pip sync requirements.txt
```

2.7 Best Practices for Reproducibility

Golden rules for environments

1. **One project = one environment**: never share an environment between projects.
2. **Versioned dependency file**: `requirements.txt` or `environment.yml` in the Git repository.
3. **Pin versions**: `numpy==1.26.2`, not `numpy` without a version.
4. **Separate dev and prod**: development tools (pytest, black) should not be in production.
5. **Document the Python version**: in the README or `pyproject.toml`.
6. **Never modify the system environment**: never use `sudo pip install`.

Common anti-patterns

- Installing all packages in the system environment
- Using `pip install` without `-r requirements.txt`
- Forgetting to update `requirements.txt` after adding a package
- Mixing Conda and pip haphazardly
- Sharing a Conda environment via an absolute path

2.8 Mini-project: Reproducible Environment

Mini-project 2: Setting up an environment

Building on the structured project from Chapter 1:

1. Create a virtual environment with `venv` or Conda.
2. Write a `requirements.in` with direct dependencies.
3. Generate a locked `requirements.txt` with `pip-tools`.
4. Write an equivalent `environment.yml` for Conda.
5. Verify that a colleague can recreate the environment from the dependency file alone.
6. Add a `setup.sh` script that automates environment creation.

2.9 Exercises

Exercise 2.1 (★ — Environment creation)**.** Create a virtual environment with each of the three tools (venv, Conda, Poetry) for a project using `numpy`, `pandas`, and `matplotlib`. Compare the environment sizes and creation times.

Exercise 2.2 (★★ — Conflict resolution)**.** A project requires `tensorflow>=2.15` and `numpy<1.24` (a real conflict). Use `pip-tools` to identify the conflict, then find compatible versions. Document the resolution process.

Exercise 2.3 (★★★ — Cross-platform environment)**.** Create an ML project that works on Linux, macOS, and Windows:

1. Write a multi-OS compatible `environment.yml` for Conda (`conda env export --from-history`)
2. Create a `setup.sh` / `setup.bat` script that detects the OS and installs appropriate dependencies
3. Test on at least two systems (or use Docker containers to simulate)
4. Handle the GPU case (install PyTorch CPU vs CUDA depending on the machine)

2.10 Cheatsheet

Chapter 2 Summary

Tool	Key Commands
venv	`python -m venv .venv`, `source .venv/bin/activate`
pip-tools	`pip-compile requirements.in`, `pip-sync`
Conda	`conda create -n env python=3.11`, `conda env export`
Poetry	`poetry add pkg`, `poetry install`, `poetry run`
uv	`uv pip install`, `uv pip compile`, `uv pip sync`

Chapter 3

Version Control — Git and DVC

3.1 Why Version Everything?

In software engineering, versioning code with Git is standard practice. In ML, we must also version **data** and **models**—two artefacts that evolve independently of the code and that jointly determine the behaviour of the system.

Definition 3.1 (Reproducibility)**.** An ML experiment is **reproducible** if, given the same code, the same data, and the same configuration, it produces the same model and the same metrics (up to hardware non-determinism).

Remark 3.2. Git alone is insufficient for ML projects: datasets and model weights are often too large for Git (files $> 100\,\text{MB}$ are impractical), and binary files do not benefit from line-level diffing.

3.2 Git Essentials for ML

3.2.1 Core workflow

Essential Git commands for ML projects

```
# Initialise a repository
git init

# Stage and commit
git add src/ configs/
git commit -m "Add training pipeline"

# Create a feature branch
git checkout -b experiment/new-feature-engineering

# Merge back
git checkout main
git merge experiment/new-feature-engineering

# Tag a release (e.g., after a successful model)
git tag -a v1.0.0 -m "Baseline model, accuracy 0.87"
```

3.2.2 .gitignore for ML projects

.gitignore tailored for ML

```
# Data and models (managed by DVC)
data/
models/
*.pkl
*.joblib
*.h5
*.pt
*.onnx

# Environments
.venv/
__pycache__/
*.pyc

# Notebooks
.ipynb_checkpoints/

# Experiment artefacts
mlruns/
wandb/
outputs/

# Secrets
.env
*.secret
```

Git branching strategy for ML

- **main**: production-ready code and pipeline definitions.
- **develop**: integration branch for ongoing work.
- **experiment/***: one branch per experiment (short-lived).
- **data/***: branches for data processing changes.
- Use **tags** to mark successful model versions (e.g., `v1.2.0-acc0.91`).

3.3 DVC — Data Version Control

DVC (Data Version Control) extends Git to handle large files (datasets, model weights) without storing them in the Git repository. Instead, DVC stores metadata (`.dvc` files) in Git and the actual data in a configurable remote storage (S3, GCS, Azure Blob, local, SSH).

Definition 3.3 (DVC)**.** DVC is an open-source version control system for data and ML models. It works *alongside* Git: Git tracks code and `.dvc` metadata files, while DVC

tracks the actual large files via content-addressable storage.

3.3.1 Getting started with DVC

DVC initialisation and basic usage

```
# Install DVC
pip install dvc dvc-s3  # add dvc-gdrive, dvc-azure, etc.

# Initialise DVC in an existing Git repo
cd my-ml-project
dvc init

# Track a dataset
dvc add data/train.csv
# Creates: data/train.csv.dvc  (metadata, tracked by Git)
#          data/.gitignore      (so Git ignores train.csv)

git add data/train.csv.dvc data/.gitignore
git commit -m "Track training data with DVC"

# Configure remote storage
dvc remote add -d myremote s3://my-bucket/dvc-store

# Push data to remote
dvc push

# Pull data on another machine
git clone <repo-url>
dvc pull
```

3.3.2 .dvc file anatomy

data/train.csv.dvc

```
outs:
  - md5: a1b2c3d4e5f6a1b2c3d4e5f6a1b2c3d4
    size: 157286400
    hash: md5
    path: train.csv
```

The `.dvc` file records the MD5 hash and size of the tracked file. When you run `dvc push`, DVC uploads the file to the remote, indexed by its hash. When you run `dvc pull`, DVC downloads the file matching the hash in the `.dvc` file.

3.3.3 Switching between data versions

Time-travelling through data versions

```
# Check out a previous data version
git checkout v1.0 -- data/train.csv.dvc
dvc checkout

# Compare data versions
dvc diff HEAD~1

# List tracked files
dvc list . --dvc-only
```

3.3.4 .dvcignore

Like `.gitignore`, the `.dvcignore` file tells DVC which files to exclude when scanning directories.

.dvcignore

```
# Temporary files
*.tmp
*.log

# OS files
.DS_Store
Thumbs.db

# Intermediate artefacts
data/raw/scratch/
```

3.4 DVC Pipelines

DVC can define reproducible ML pipelines in a `dvc.yaml` file. Each stage declares its dependencies and outputs; DVC tracks which stages need to be re-run when inputs change.

dvc.yaml — Complete ML pipeline

```
stages:
  prepare:
    cmd: python src/prepare.py --config configs/config.yaml
    deps:
      - src/prepare.py
      - data/raw/
      - configs/config.yaml
    outs:
      - data/processed/train.csv
```

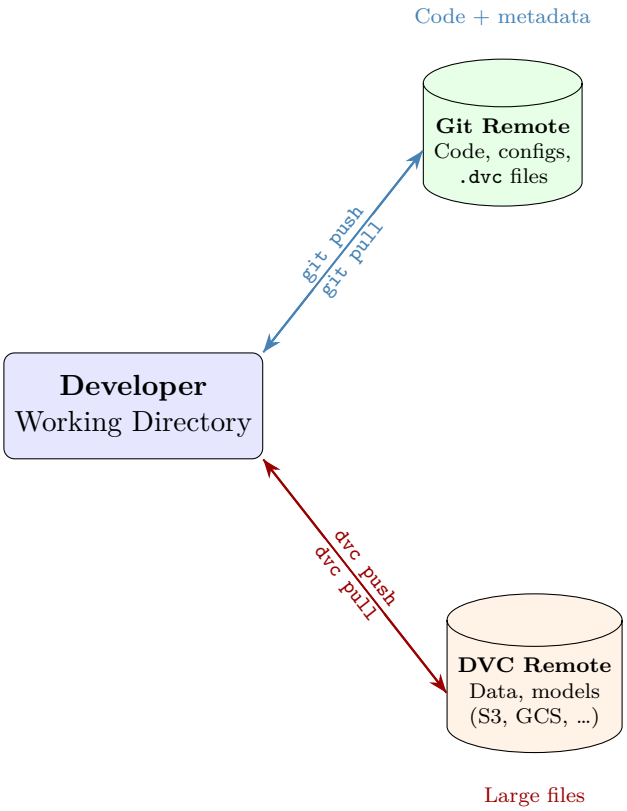
```
      - data/processed/test.csv

  train:
    cmd: python src/train.py --config configs/config.yaml
    deps:
      - src/train.py
      - data/processed/train.csv
      - configs/config.yaml
    outs:
      - models/model.pkl
    metrics:
      - metrics/train_metrics.json:
          cache: false

  evaluate:
    cmd: python src/evaluate.py --config configs/config.yaml
    deps:
      - src/evaluate.py
      - models/model.pkl
      - data/processed/test.csv
    metrics:
      - metrics/eval_metrics.json:
          cache: false
    plots:
      - metrics/confusion_matrix.csv:
          x: predicted
          y: actual
```

Running and inspecting DVC pipelines

```
# Run the full pipeline (only re-runs changed stages)
dvc repro

# Visualise the DAG
dvc dag

# Compare metrics across Git commits
dvc metrics diff

# Show metrics
dvc metrics show

# Show plots
dvc plots show
```

Remark 3.4. DVC uses file hashes to determine whether a stage needs re-execution. If neither the dependencies nor the code have changed, the stage is skipped—making iteration fast.

3.5 Git + DVC Workflow



3.6 Tool Comparison

Criterion	DVC	Git LFS	Delta Lake	LakeFS
Open source	Yes	Yes	Yes	Yes
Git integration	Excellent	Native	No	Git-like
Pipeline support	Yes	No	No	No
Storage backends	Many	Git server	Object store	S3-compatible
Data format	Any	Any	Parquet	Any
Branching for data	Via Git	Via Git	No	Yes
Scalability	Good	Limited	Excellent	Excellent
Learning curve	Low	Very low	Medium	Medium

Git LFS limitations

Git LFS stores large files on the Git server itself, which can become expensive and slow for multi-GB datasets. DVC decouples data storage from the Git server, allowing you to use cheap cloud storage (S3, GCS). Git LFS also lacks pipeline support and metric tracking.

3.7 Best Practices

Version control golden rules

1. **Commit often, commit small**: each commit should represent one logical change.
2. **Never commit data to Git**: use DVC or a similar tool.
3. **Never commit secrets**: use `.env` files excluded via `.gitignore`.
4. **Tag successful experiments**: `git tag v1.0-acc0.92`.
5. **Use DVC pipelines**: make your workflow reproducible with `dvc.yaml`.
6. **Automate with hooks**: pre-commit hooks for linting, formatting, secret scanning.

Common anti-patterns

- Storing datasets in Git (slow clone, bloated history)
- Using `model_v2_final_FINAL.pkl` naming instead of proper versioning
- Not versioning data at all ("I'll remember which file I used")
- Giant commits with thousands of changed files
- Forgetting to `dvc push` after `git push`

3.8 Mini-project: Versioned ML Pipeline

Mini-project 3: Git + DVC pipeline

Building on the project from Chapters 1 and 2:

1. Initialise a Git repository and DVC.
2. Track a dataset (≥ 50 MB) with DVC and configure a remote (local directory or S3).
3. Write a `dvc.yaml` with three stages: `prepare`, `train`, `evaluate`.
4. Run `dvc repro` and verify that only changed stages re-run.
5. Tag the first successful run as `v1.0`.
6. Modify a hyperparameter, re-run, and use `dvc metrics diff` to compare with the previous version.
7. Push code to Git and data to DVC remote; verify a colleague can reproduce the results with `git clone` + `dvc pull` + `dvc repro`.

3.9 Exercises

Exercise 3.1 (★ — DVC basics)**.** Initialise a Git+DVC project. Track a CSV file (≥ 10 MB) with DVC. Configure a local remote (`dvc remote add -d local /tmp/dvc-store`). Push and pull the file. Verify the `.dvc` file contents and explain each field.

Exercise 3.2 (★★ — Pipeline design)**.** Design and implement a DVC pipeline (`dvc.yaml`) for a text classification task with the following stages:

1. `download`: download the dataset from a URL
2. `preprocess`: tokenisation, vocabulary building
3. `train`: train a model, log metrics to JSON
4. `evaluate`: compute precision, recall, F1 on the test set

Run the pipeline, change a hyperparameter, re-run, and compare metrics with `dvc metrics diff`.

Exercise 3.3 (★★★ — Collaborative workflow)**.** Simulate a two-person collaboration using Git branches and DVC:

1. Person A creates an experiment branch, modifies the feature engineering, trains, and pushes code + data
2. Person B creates a different experiment branch with a different model architecture
3. Both merge into `develop`, resolving any conflicts in `dvc.yaml` and `dvc.lock`
4. Compare results using `dvc metrics diff` across branches
5. Write a short report documenting the collaboration workflow and any issues encountered

3.10 Cheatsheet

Chapter 3 Summary

Tool / Concept	Key Commands
Git basics	`git init`, `git add`, `git commit`, `git tag`
DVC init	`dvc init`, `dvc add <file>`, `dvc remote add`
DVC data	`dvc push`, `dvc pull`, `dvc checkout`, `dvc diff`
DVC pipelines	`dvc.yaml`, `dvc repro`, `dvc dag`, `dvc metrics show`
Best practice	Git for code + `.dvc` files; DVC remote for data + models

Chapter 4

Experiment Tracking — MLflow, Weights & Biases

4.1 Why Track Experiments?

During an ML project, a data scientist may run hundreds of experiments varying hyperparameters, features, data splits, and model architectures. Without systematic tracking, critical questions become unanswerable:

- Which hyperparameters produced the best F1 score?
- What data version was used for model v2.3?
- Why did accuracy drop between Tuesday and Wednesday?
- Can we reproduce the result from last month?

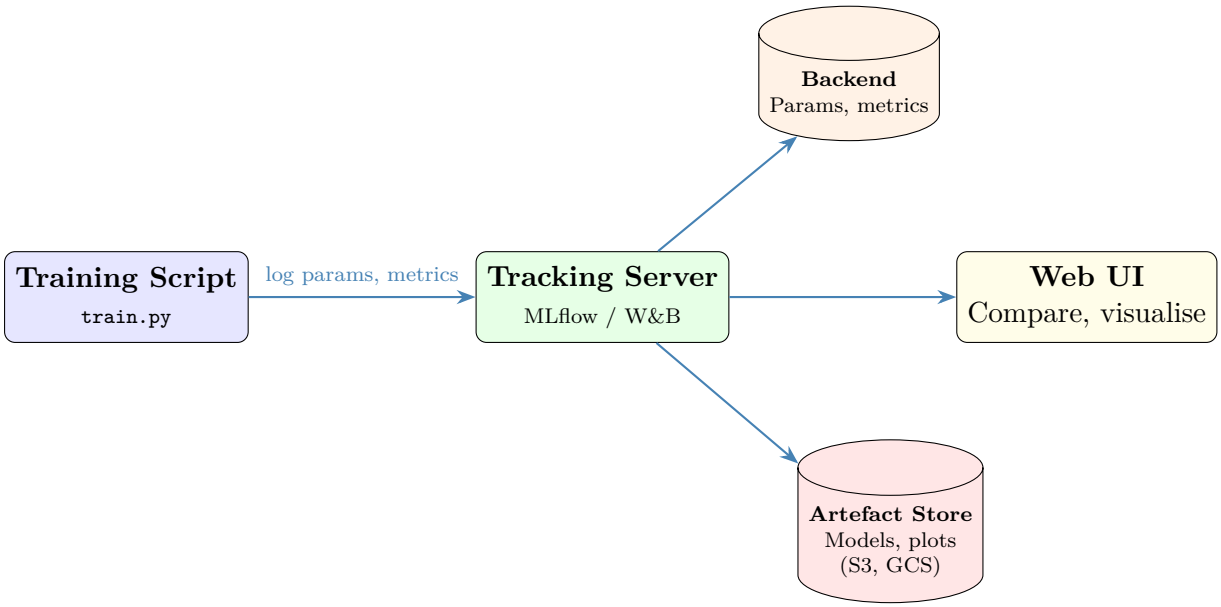
Definition 4.1 (Experiment tracking)**. Experiment tracking** is the practice of systematically recording the parameters, metrics, artefacts, code versions, and environment details of every ML experiment, enabling comparison, reproducibility, and auditability.

The spreadsheet trap

Tracking experiments in spreadsheets or notebooks is fragile:

- Manual entry is error-prone and incomplete.
- No link between logged values and actual code/data.
- No automatic artefact storage.
- Impossible to search, filter, or visualise at scale.

4.2 Experiment Tracking Architecture



4.3 MLflow

MLflow is an open-source platform for managing the ML lifecycle. Its *Tracking* component logs parameters, metrics, and artefacts. Its *Model Registry* manages model versions and deployment stages.

4.3.1 MLflow Tracking API

Complete MLflow tracking example

```python
import mlflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score

# Set the tracking URI (local or remote server)
mlflow.set_tracking_uri("http://localhost:5000")
mlflow.set_experiment("iris-classification")

# Load data
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Experiment parameters
params = {
    "n_estimators": 200,
    "max_depth": 10,
    "min_samples_split": 5,
```

```
    "random_state": 42,
}

with mlflow.start_run(run_name="rf-baseline"):
    # Log parameters
    mlflow.log_params(params)

    # Train
    model = RandomForestClassifier(**params)
    model.fit(X_train, y_train)

    # Evaluate
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred, average="weighted")

    # Log metrics
    mlflow.log_metric("accuracy", acc)
    mlflow.log_metric("f1_weighted", f1)

    # Log the model as an artefact
    mlflow.sklearn.log_model(model, "random-forest-model")

    # Log additional artefacts (plots, configs, etc.)
    mlflow.log_artifact("configs/config.yaml")

    print(f"Run ID: {mlflow.active_run().info.run_id}")
    print(f"Accuracy: {acc:.4f}, F1: {f1:.4f}")
```

Launching the MLflow UI

```
# Start the tracking server
mlflow server --host 0.0.0.0 --port 5000 \
    --backend-store-uri sqlite:///mlflow.db \
    --default-artifact-root ./mlartifacts

# Open in browser: http://localhost:5000
```

4.3.2 MLflow Model Registry

The Model Registry provides a centralised store for managing model versions and their lifecycle stages (*Staging*, *Production*, *Archived*).

MLflow Model Registry workflow

```
import mlflow
from mlflow.tracking import MlflowClient
```

```
client = MlflowClient()

# Register a model from a run
model_uri = f"runs:/{run_id}/random-forest-model"
result = mlflow.register_model(model_uri, "IrisClassifier")
print(f"Version: {result.version}")

# Transition to staging
client.transition_model_version_stage(
    name="IrisClassifier",
    version=result.version,
    stage="Staging",
)

# Promote to production
client.transition_model_version_stage(
    name="IrisClassifier",
    version=result.version,
    stage="Production",
)

# Load a production model for inference
model = mlflow.pyfunc.load_model(
    "models:/IrisClassifier/Production"
)
predictions = model.predict(X_test)
```

MLflow best practices

- Use a **remote tracking server** (not local `mlruns/`) for team collaboration.
- Use **experiment names** that reflect the project and task.
- Log **all** hyperparameters, not just the ones you are tuning.
- Use `mlflow.autolog()` for supported frameworks (scikit-learn, PyTorch, TensorFlow) to automatically log parameters and metrics.
- Store artefacts (models, plots) in a shared object store (S3, GCS).

4.4 Weights & Biases (W&B)

Weights & Biases (W&B) is a cloud-hosted (or self-hosted) experiment tracking platform with rich visualisation, hyperparameter sweeps, and team collaboration features.

Complete W&B tracking example

```
import wandb
import torch
```

```python
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset

# Initialise a W&B run
wandb.init(
    project="mnist-classification",
    name="cnn-baseline",
    config={
        "learning_rate": 1e-3,
        "batch_size": 64,
        "epochs": 20,
        "architecture": "SimpleCNN",
        "optimizer": "Adam",
    },
)
config = wandb.config

# Define model
class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2)
        self.fc = nn.Linear(64 * 7 * 7, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        return self.fc(x)

model = SimpleCNN()
optimizer = torch.optim.Adam(
    model.parameters(), lr=config.learning_rate
)
criterion = nn.CrossEntropyLoss()

# Watch model (log gradients and parameters)
wandb.watch(model, log="all", log_freq=100)

# Training loop
for epoch in range(config.epochs):
    model.train()
    total_loss = 0
    correct = 0
    total = 0

    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
```

```
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        _, predicted = outputs.max(1)
        correct += predicted.eq(y_batch).sum().item()
        total += y_batch.size(0)

    # Log metrics per epoch
    wandb.log({
        "epoch": epoch,
        "train/loss": total_loss / len(train_loader),
        "train/accuracy": correct / total,
    })

# Log final model as artefact
artifact = wandb.Artifact("cnn-model", type="model")
torch.save(model.state_dict(), "model.pt")
artifact.add_file("model.pt")
wandb.log_artifact(artifact)

wandb.finish()
```

Remark 4.2. W&B automatically captures Git commit hashes, system information (GPU, OS, Python version), and console output. The web dashboard allows real-time comparison of runs with interactive charts.

4.5 Hydra for Configuration Management

Hydra (by Meta) is a configuration framework that simplifies managing complex configs with composition, overrides, and multi-run sweeps.

Hydra configuration example

```
# configs/config.yaml
defaults:
  - model: random_forest
  - data: iris
  - _self_

training:
  seed: 42
  test_size: 0.2

# configs/model/random_forest.yaml
n_estimators: 200
max_depth: 10
```

```yaml
# configs/model/gradient_boosting.yaml
n_estimators: 300
learning_rate: 0.1
max_depth: 5

# configs/data/iris.yaml
path: data/iris.csv
target_col: species
```

Using Hydra in Python

```python
import hydra
from omegaconf import DictConfig
import mlflow

@hydra.main(version_base=None, config_path="configs",
            config_name="config")
def train(cfg: DictConfig) -> float:
    mlflow.set_experiment("hydra-experiments")

    with mlflow.start_run():
        mlflow.log_params(dict(cfg.model))
        # ... training code ...
        mlflow.log_metric("accuracy", accuracy)

    return accuracy

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

Hydra command-line overrides and sweeps

```bash
# Override a parameter
python train.py model.n_estimators=500

# Switch model configuration
python train.py model=gradient_boosting

# Multi-run sweep
python train.py --multirun \
    model.n_estimators=100,200,500 \
    model.max_depth=5,10,20
```

4.6 Tool Comparison

Criterion	MLflow	W&B	Neptune	ClearML
Open source	Yes	Partial	Partial	Yes
Self-hosted	Yes	Yes	Yes	Yes
Cloud hosted	No	Yes	Yes	Yes
Free tier	N/A	Yes	Yes	Yes
Model registry	Yes	Yes	Yes	Yes
HPO sweeps	No	Yes	Yes	Yes
Real-time logs	No	Yes	Yes	Yes
Team collab	Basic	Excellent	Good	Good
Framework support	Broad	Broad	Broad	Broad
Learning curve	Low	Low	Low	Medium

Remark 4.3. **MLflow** is the de facto standard for self-hosted, open-source experiment tracking. **W&B** excels in visualisation and team collaboration for cloud-hosted workflows. Many teams use both: MLflow for the model registry and W&B for experiment visualisation.

4.7 Best Practices

Experiment tracking golden rules

1. **Log everything**: parameters, metrics, data version, code version, environment, random seeds.
2. **Use meaningful run names**: `rf-200trees-depth10`, not `run-42`.
3. **Organise experiments**: one experiment per task or project.
4. **Store artefacts**: save models, plots, and configs alongside metrics.
5. **Automate logging**: use `mlflow.autolog()` or W&B integrations.
6. **Compare systematically**: use the UI to compare runs, not memory or spreadsheets.

Common anti-patterns

- Logging only the final metric, not per-epoch curves
- Forgetting to log the data version used for training
- Using `print()` instead of a tracking tool
- Running experiments without setting random seeds
- Not saving the model artefact alongside the metrics

4.8 Mini-project: Tracked Experiment Campaign

Mini-project 4: Full experiment tracking

Building on the DVC-versioned project from Chapter 3:

1. Set up an MLflow tracking server (local or Docker).
2. Instrument your training script to log parameters, metrics, and the trained model.
3. Run at least 5 experiments varying hyperparameters.
4. Use the MLflow UI to identify the best run.
5. Register the best model in the MLflow Model Registry and promote it to *Production*.
6. Add W&B tracking alongside MLflow and compare the dashboards.
7. Use Hydra to manage experiment configurations and run a sweep over 3 hyperparameters.

4.9 Exercises

Exercise 4.1 (★ — MLflow basics)**.** Write a training script that logs at least 5 parameters and 3 metrics to MLflow. Launch the MLflow UI and take a screenshot of the comparison view for three different runs. Explain what each column represents.

Exercise 4.2 (★★ — W&B integration)**.** Rewrite the training script from the previous exercise using W&B instead of MLflow. Log training curves (loss and accuracy per epoch), a confusion matrix, and a model artefact. Use `wandb.sweep` to perform a hyperparameter search over learning rate, batch size, and number of layers. Compare the W&B and MLflow experiences.

Exercise 4.3 (★★★ — End-to-end tracked pipeline)**.** Build a complete tracked pipeline for image classification (e.g., CIFAR-10):

1. Use Hydra for configuration management with at least 3 config groups (model, data, training)
2. Log all experiments to both MLflow and W&B
3. Implement early stopping based on validation loss
4. Log per-epoch metrics, learning rate schedules, and sample predictions
5. Use the MLflow Model Registry to manage model versions
6. Write a comparison report using data exported from the tracking tools

4.10 Cheatsheet

Chapter 4 Summary

Tool / Concept	Key Commands / API
MLflow tracking	`mlflow.start_run()`, `mlflow.log_params()`, `mlflow.log_metric()`
MLflow registry	`mlflow.register_model()`, `client.transition_model_version_stage()`
MLflow server	`mlflow server --host 0.0.0.0 --port 5000`
W&B	`wandb.init()`, `wandb.log()`, `wandb.log_artifact()`
Hydra	`@hydra.main()`, `--multirun`, `model=gradient_boosting`
Best practice	Log everything; meaningful names; store artefacts

Chapter 5

Data Pipelines and Feature Stores

5.1 Why Data Pipelines?

In production ML, data rarely arrives clean and ready. A **data pipeline** automates the sequence of steps that transform raw data into features suitable for training or inference.

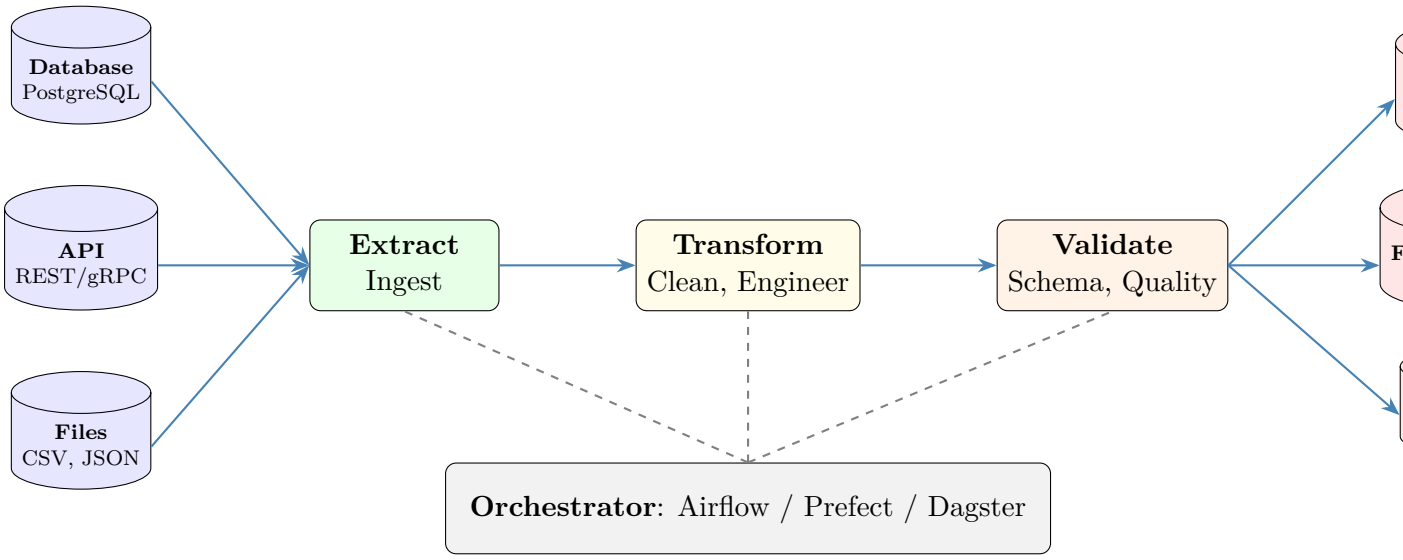
Definition 5.1 (Data pipeline)**.** A **data pipeline** is an automated, orchestrated sequence of data processing steps (extraction, transformation, validation, loading) that converts raw data sources into curated datasets ready for analysis or model training.

Remark 5.2. Without automated pipelines, data scientists spend up to 80% of their time on data wrangling—manually downloading, cleaning, and transforming data for each experiment.

5.2 ETL vs ELT

	ETL	**ELT**
Order	Extract → Transform → Load	Extract → Load → Transform
Where	Transformation on an intermediate server	Transformation inside the data warehouse
Best for	Structured data, relational sources	Large-scale, cloud data warehouses
Tools	Airflow, Prefect, Luigi	dbt, BigQuery, Snowflake
ML use case	Feature engineering pipelines	Analytics features from data warehouse

5.3 Pipeline Architecture



5.4 Orchestration with Airflow

Apache Airflow is the most widely used workflow orchestrator. It defines pipelines as Python DAGs (Directed Acyclic Graphs) and provides a web UI for monitoring, scheduling, and retry management.

Airflow DAG for an ML pipeline

```python
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta

default_args = {
    "owner": "ml-team",
    "retries": 2,
    "retry_delay": timedelta(minutes=5),
}

with DAG(
    dag_id="ml_training_pipeline",
    default_args=default_args,
    schedule_interval="@daily",
    start_date=datetime(2025, 1, 1),
    catchup=False,
    tags=["ml", "training"],
) as dag:

    def extract_data(**kwargs):
        """Extract data from source."""
        import pandas as pd
        df = pd.read_sql("SELECT * FROM events", conn)
        df.to_parquet("/data/raw/events.parquet")

    def transform_data(**kwargs):
```

```
        """Clean and engineer features."""
        import pandas as pd
        df = pd.read_parquet("/data/raw/events.parquet")
        df = df.dropna(subset=["target"])
        df["hour"] = pd.to_datetime(df["ts"]).dt.hour
        df.to_parquet("/data/processed/features.parquet")

    def train_model(**kwargs):
        """Train and log the model."""
        import mlflow
        # ... training with MLflow logging ...

    extract = PythonOperator(
        task_id="extract_data", python_callable=extract_data
    )
    transform = PythonOperator(
        task_id="transform_data", python_callable=transform_data
    )
    train = PythonOperator(
        task_id="train_model", python_callable=train_model
    )

    extract >> transform >> train
```

Airflow pitfalls

- Airflow is an **orchestrator**, not a data processing engine. Heavy computations should run on Spark, Dask, or dedicated workers.
- The scheduler can be resource-hungry; plan infrastructure accordingly.
- DAG files are parsed frequently—keep them lightweight (no heavy imports at module level).

5.5 Orchestration with Prefect

Prefect is a modern alternative to Airflow with a simpler API. Flows are regular Python functions decorated with `@flow` and `@task`.

Complete Prefect ML pipeline

```
from prefect import flow, task
from prefect.tasks import task_input_hash
from datetime import timedelta
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```python
import mlflow

@task(retries=3, cache_key_fn=task_input_hash,
      cache_expiration=timedelta(hours=1))
def extract_data(source: str) -> pd.DataFrame:
    """Extract data from source with caching."""
    df = pd.read_csv(source)
    print(f"Extracted {len(df)} rows")
    return df

@task
def validate_data(df: pd.DataFrame) -> pd.DataFrame:
    """Validate data quality."""
    assert len(df) > 100, "Dataset too small"
    assert df.isnull().mean().max() < 0.3, "Too many nulls"
    null_pct = df.isnull().mean()
    print(f"Max null %: {null_pct.max():.2%}")
    return df.dropna()

@task
def engineer_features(df: pd.DataFrame) -> pd.DataFrame:
    """Create features for training."""
    df["feature_ratio"] = df["col_a"] / (df["col_b"] + 1)
    df["feature_log"] = df["col_c"].apply(
        lambda x: max(x, 0)
    ).apply(lambda x: x ** 0.5)
    return df

@task
def train_and_evaluate(
    df: pd.DataFrame,
    target_col: str = "target",
    n_estimators: int = 200,
) -> dict:
    """Train model and return metrics."""
    X = df.drop(columns=[target_col])
    y = df[target_col]
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )

    with mlflow.start_run():
        model = RandomForestClassifier(
            n_estimators=n_estimators, random_state=42
        )
        model.fit(X_train, y_train)

        acc = accuracy_score(y_test, model.predict(X_test))
        mlflow.log_param("n_estimators", n_estimators)
        mlflow.log_metric("accuracy", acc)
        mlflow.sklearn.log_model(model, "model")
```

```python
    return {"accuracy": acc, "n_samples": len(df)}

@flow(name="ML Training Pipeline",
      description="End-to-end training pipeline")
def training_pipeline(
    data_source: str = "data/dataset.csv",
    n_estimators: int = 200,
):
    """Main pipeline flow."""
    raw_data = extract_data(data_source)
    clean_data = validate_data(raw_data)
    features = engineer_features(clean_data)
    metrics = train_and_evaluate(
        features, n_estimators=n_estimators
    )
    print(f"Pipeline complete. Accuracy: {metrics['accuracy']:.4f}")
    return metrics

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

Running and scheduling Prefect flows

```bash
# Run locally
python pipeline.py

# Start the Prefect server
prefect server start

# Deploy with a schedule
prefect deployment build pipeline.py:training_pipeline \
    --name "daily-training" \
    --cron "0 6 * * *"
prefect deployment apply training_pipeline-deployment.yaml

# Start a worker to execute scheduled runs
prefect worker start -p default-agent-pool
```

5.6 DVC Pipelines for ML

As seen in Chapter 3, DVC pipelines define reproducible ML workflows in `dvc.yaml`. Unlike Airflow/Prefect, DVC pipelines are **data-driven** (stages re-run only when inputs change) rather than **schedule-driven**.

Aspect	DVC Pipelines	Airflow/Prefect
Trigger	Data/code changes	Schedule or event
State tracking	File hashes	Database
Caching	Content-addressable	Time-based
Deployment	CLI (`dvc repro`)	Server + workers
Best for	Experimentation	Production orchestration

5.7 Feature Stores

A **feature store** is a centralised repository for storing, sharing, and serving ML features. It solves the problem of feature reuse and training-serving skew.

Definition 5.3 (Feature store)**.** A **feature store** is a data system that manages the lifecycle of ML features: definition, computation, storage, versioning, and serving for both training (batch) and inference (online, low-latency).

Training-serving skew

Training-serving skew occurs when features are computed differently at training time and inference time. For example, a feature computed as a 7-day rolling average in a batch training script may be computed as a simple average in the serving code. A feature store ensures the *same* transformation is used in both contexts.

5.7.1 Feast

Feast (Feature Store) is an open-source feature store that provides offline (batch) and online (low-latency) feature serving.

Feast feature definitions

```python
# feature_repo/features.py
from feast import Entity, FeatureView, Field, FileSource
from feast.types import Float32, Int64
from datetime import timedelta

# Define the entity (the business object)
customer = Entity(
    name="customer_id",
    join_keys=["customer_id"],
    description="Unique customer identifier",
)

# Define the data source
customer_source = FileSource(
    path="data/customer_features.parquet",
    timestamp_field="event_timestamp",
)

# Define features
customer_features = FeatureView(
```

```
    name="customer_features",
    entities=[customer],
    ttl=timedelta(days=90),
    schema=[
        Field(name="total_purchases", dtype=Int64),
        Field(name="avg_order_value", dtype=Float32),
        Field(name="days_since_last_order", dtype=Int64),
        Field(name="purchase_frequency", dtype=Float32),
    ],
    source=customer_source,
    online=True,
    tags={"team": "ml-platform"},
)
```

Using Feast for training and serving

```
from feast import FeatureStore
import pandas as pd

store = FeatureStore(repo_path="feature_repo/")

# --- Offline: get historical features for training ---
entity_df = pd.DataFrame({
    "customer_id": [1001, 1002, 1003],
    "event_timestamp": pd.to_datetime(["2025-01-15"] * 3),
})

training_df = store.get_historical_features(
    entity_df=entity_df,
    features=[
        "customer_features:total_purchases",
        "customer_features:avg_order_value",
        "customer_features:days_since_last_order",
        "customer_features:purchase_frequency",
    ],
).to_df()

print(training_df.head())

# --- Online: get features for real-time inference ---
# First, materialise features to the online store
# feast materialize 2024-01-01T00:00:00 2025-06-01T00:00:00

online_features = store.get_online_features(
    features=[
        "customer_features:total_purchases",
        "customer_features:avg_order_value",
    ],
    entity_rows=[{"customer_id": 1001}],
).to_dict()
```

```
print(online_features)
```

Feast CLI commands

```
# Initialise a feature repository
feast init feature_repo
cd feature_repo

# Apply feature definitions
feast apply

# Materialise features to the online store
feast materialize 2024-01-01T00:00:00 2025-06-01T00:00:00

# Serve features via REST API
feast serve
```

5.8 Orchestrator Comparison

Criterion	Airflow	Prefect	Dagster	Luigi
Language	Python	Python	Python	Python
DAG definition	Python	Python decorators	Python + assets	Python classes
UI	Good	Excellent	Excellent	Basic
Scheduling	Cron, event	Cron, event	Cron, sensor	Cron
Caching	No	Yes	Yes	Yes
Data lineage	Limited	Good	Excellent	Limited
Cloud managed	Yes	Yes	Yes	No
Community	Very large	Growing	Growing	Small
Learning curve	Medium	Low	Medium	Low

Choosing an orchestrator

- **Airflow**: battle-tested, huge ecosystem, best for large teams with complex scheduling needs.
- **Prefect**: modern API, easy to start, best for teams that want Python-native pipelines without boilerplate.
- **Dagster**: asset-centric paradigm, excellent for data engineering teams that think in terms of data assets.
- **DVC**: best for ML experimentation where pipelines are re-run on data/code changes, not on a schedule.

5.9 Best Practices

Pipeline design golden rules

1. **Idempotency**: running a pipeline twice with the same input must produce the same output.
2. **Atomicity**: each step either fully succeeds or fully fails (no partial writes).
3. **Data validation**: validate schemas and distributions at every stage boundary.
4. **Retry logic**: transient failures (network, API rate limits) should be retried with backoff.
5. **Logging and alerting**: every step should log its progress and alert on failure.
6. **Separation of concerns**: orchestration logic (DAG definition) should be separate from business logic (transformations).

Pipeline anti-patterns

- Running heavy computation inside the orchestrator process
- Hardcoding file paths and connection strings
- No data validation between pipeline stages
- Pipelines that cannot be re-run safely (non-idempotent writes)
- Feature computation duplicated between training and serving code

5.10 Mini-project: Orchestrated ML Pipeline

Mini-project 5: End-to-end orchestrated pipeline

Build a complete data-to-model pipeline:

1. Write a Prefect flow with 4 tasks: `extract`, `validate`, `engineer_features`, `train`.
2. Add data validation checks (schema, null percentage, class balance).
3. Integrate MLflow tracking in the training task.
4. Add retry logic and caching for the extraction task.
5. Deploy the pipeline with a daily schedule.
6. (Bonus) Define 2–3 features in Feast and serve them for training and online inference.

5.11 Exercises

Exercise 5.1 (★ — Prefect basics)**.** Write a Prefect flow with three tasks: (1) download a CSV from a URL, (2) compute basic statistics (mean, std, null count per column), (3) save a summary report to a JSON file. Add retry logic to the download task. Run the flow and inspect the Prefect UI.

Exercise 5.2 (★★ — Pipeline with validation)**.** Build a data pipeline (Prefect or Airflow) for a tabular ML task that includes:

1. Data extraction from two sources (CSV + SQLite database)
2. Data joining and cleaning
3. Schema validation using `pandera` or `great-expectations`
4. Feature engineering (at least 3 derived features)
5. Train/test split with the split ratio logged

Verify that the pipeline fails gracefully when given malformed data.

Exercise 5.3 (★★★ — Feature store integration)**.** Build a feature store using Feast for a customer churn prediction task:

1. Define at least 5 features across 2 feature views
2. Materialise features to the online store
3. Write a training script that pulls historical features from Feast
4. Write a serving script that pulls online features for real-time prediction
5. Verify consistency between offline and online feature values
6. Document the feature definitions and their business meaning

5.12 Cheatsheet

Chapter 5 Summary

Tool / Concept	Key Points
ETL vs ELT	ETL transforms before loading; ELT transforms inside the warehouse
Airflow	DAGs in Python, cron scheduling, `>>` for dependencies
Prefect	`@flow`, `@task` decorators, built-in caching and retries
DVC pipelines	`dvc.yaml`, `dvc repro`, data-driven re-execution
Feast	`feast apply`, `feast materialize`, `store.get_historical_features()`
Best practice	Idempotent, validated, retryable, logged

CHAPTER 5. DATA PIPELINES AND FEATURE STORES

Chapter 6

Large-Scale Training — Distributed Training

6.1 Why Distributed Training?

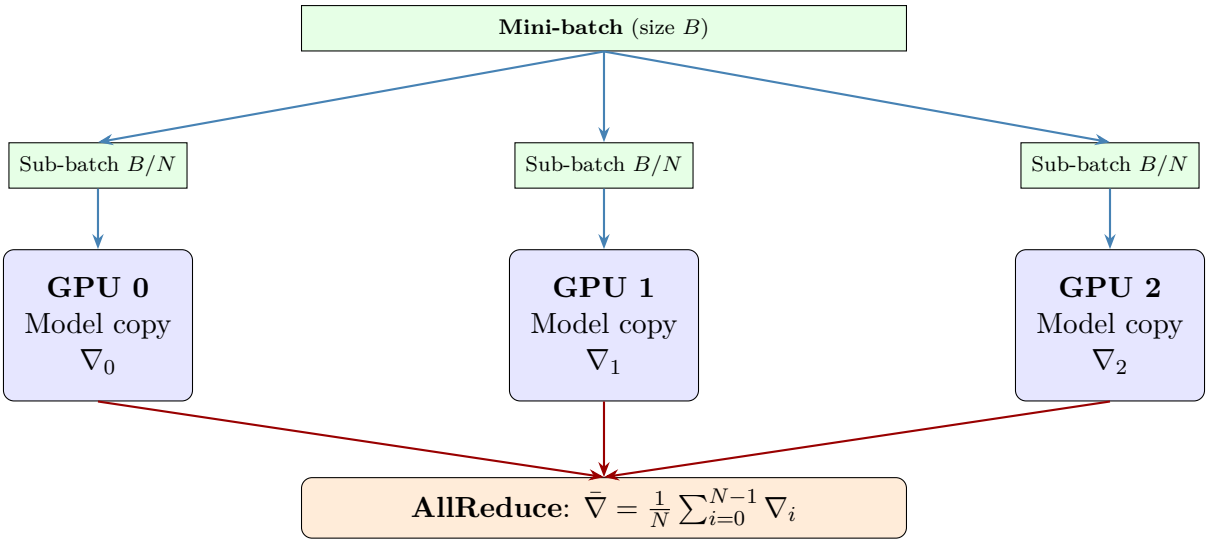
Modern deep learning models (LLMs, vision transformers, diffusion models) have billions of parameters and require terabytes of training data. A single GPU cannot fit the model or process the data in a reasonable time. Distributed training splits the workload across multiple devices.

Definition 6.1 (Distributed training)**. Distributed training** is the practice of training a model across multiple GPUs, machines, or accelerators by partitioning the computation (data, model, or both) and synchronising the results.

6.2 Data Parallelism vs Model Parallelism

	Data Parallelism	**Model Parallelism**
Principle	Same model on each device, different data	Different parts of the model on different devices
When	Model fits in one GPU memory	Model too large for one GPU
Communication	Gradient synchronisation (AllReduce)	Activation exchange between devices
Scaling	Near-linear with N GPUs	Sub-linear (pipeline bubbles)
Complexity	Low	High
Tools	DDP, DeepSpeed ZeRO	Megatron-LM, pipeline parallelism

6.2.1 Data parallelism — visual overview



All GPUs update weights with $\bar{\nabla}$

6.3 Mathematics of Data-Parallel SGD

Consider N workers, each computing gradients on a sub-batch of size $b = B/N$ where B is the global batch size.

Data-parallel SGD

Each worker k computes:

$$\nabla_k = \frac{1}{b} \sum_{i \in \mathcal{B}_k} \nabla_\theta \ell(f_\theta(x_i), y_i)$$

where $\mathcal{B}_k$ is the sub-batch assigned to worker k.
The averaged gradient is:

$$\bar{\nabla} = \frac{1}{N} \sum_{k=0}^{N-1} \nabla_k = \frac{1}{B} \sum_{i \in \mathcal{B}} \nabla_\theta \ell(f_\theta(x_i), y_i)$$

which is identical to computing the gradient on the full batch $\mathcal{B}$.
The parameter update is:

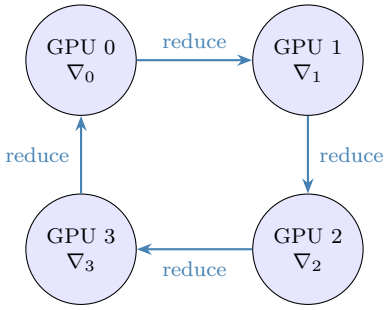
$$\theta_{t+1} = \theta_t - \eta \, \bar{\nabla}$$

Linear scaling rule (Goyal et al., 2017): when multiplying the batch size by N, multiply the learning rate by N and use a warm-up period:

$$\eta_{\text{distributed}} = N \cdot \eta_{\text{single}}$$

Remark 6.2. The AllReduce operation ensures that all workers have identical gradients (and thus identical model weights) after each step. This is equivalent to training on a single device with batch size $B = N \cdot b$.

6.3.1 AllReduce communication



Ring AllReduce: $O\left(\frac{2(N-1)}{N} \cdot M\right)$ communication

6.4 PyTorch Distributed Data Parallel (DDP)

DistributedDataParallel (DDP) is PyTorch's recommended approach for multi-GPU training. It uses NCCL for GPU-to-GPU communication and overlaps gradient computation with communication for efficiency.

Complete PyTorch DDP training script

```
import os
import torch
import torch.nn as nn
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data import DataLoader, DistributedSampler
from torchvision import datasets, transforms

def setup(rank, world_size):
    """Initialise the distributed process group."""
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "12355"
    dist.init_process_group(
        backend="nccl", rank=rank, world_size=world_size
    )
    torch.cuda.set_device(rank)

def cleanup():
    dist.destroy_process_group()

class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2),
```

```
        )
        self.classifier = nn.Sequential(
            nn.Linear(64 * 7 * 7, 256), nn.ReLU(),
            nn.Linear(256, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        return self.classifier(x)

def train(rank, world_size, epochs=10):
    setup(rank, world_size)

    # Data
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,)),
    ])
    dataset = datasets.MNIST(
        "data", train=True, download=True, transform=transform
    )
    sampler = DistributedSampler(
        dataset, num_replicas=world_size, rank=rank
    )
    loader = DataLoader(
        dataset, batch_size=64, sampler=sampler, num_workers=4
    )

    # Model
    model = SimpleCNN().to(rank)
    model = DDP(model, device_ids=[rank])

    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    criterion = nn.CrossEntropyLoss()

    # Training loop
    for epoch in range(epochs):
        sampler.set_epoch(epoch)  # Shuffle differently each epoch
        model.train()
        total_loss = 0

        for batch_idx, (data, target) in enumerate(loader):
            data, target = data.to(rank), target.to(rank)

            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
```

```
            total_loss += loss.item()

        if rank == 0:
            avg_loss = total_loss / len(loader)
            print(f"Epoch {epoch}: loss = {avg_loss:.4f}")

    # Save model (only on rank 0)
    if rank == 0:
        torch.save(
            model.module.state_dict(), "model_ddp.pt"
        )

    cleanup()

if __name__ == "__main__":
    world_size = torch.cuda.device_count()
    torch.multiprocessing.spawn(
        train, args=(world_size,), nprocs=world_size
    )
```

Launching DDP training

```
# Single node, multiple GPUs
torchrun --nproc_per_node=4 train_ddp.py

# Multi-node (2 nodes, 4 GPUs each)
# On node 0:
torchrun --nproc_per_node=4 --nnodes=2 --node_rank=0 \
    --master_addr=node0 --master_port=12355 train_ddp.py
# On node 1:
torchrun --nproc_per_node=4 --nnodes=2 --node_rank=1 \
    --master_addr=node0 --master_port=12355 train_ddp.py
```

DDP pitfalls

- Always call `sampler.set_epoch(epoch)` to ensure proper shuffling across epochs.
- Save checkpoints only on rank 0 to avoid file corruption.
- Access the underlying model via `model.module` (not `model`) when saving or inspecting.
- Ensure all processes reach the same synchronisation points (no conditional code paths that differ across ranks).

6.5 Hugging Face Accelerate

Accelerate (by Hugging Face) provides a high-level abstraction over DDP, FSDP, and DeepSpeed. A single training script works on CPU, single GPU, multi-GPU, and multi-node without code changes.

Training with Accelerate

```python
from accelerate import Accelerator
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

accelerator = Accelerator()

# Model, optimizer, data (standard PyTorch)
model = SimpleCNN()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss()

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)),
])
dataset = datasets.MNIST(
    "data", train=True, download=True, transform=transform
)
loader = DataLoader(dataset, batch_size=64, shuffle=True)

# Prepare for distributed training (one line!)
model, optimizer, loader = accelerator.prepare(
    model, optimizer, loader
)

# Standard training loop (no rank checks needed)
for epoch in range(10):
    model.train()
    for data, target in loader:
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        accelerator.backward(loss)
        optimizer.step()

    if accelerator.is_main_process:
        print(f"Epoch {epoch} complete")

# Save
accelerator.save_model(model, "model_accelerate/")
```

Accelerate configuration and launch

```
# Interactive configuration
accelerate config
# Generates ~/.cache/huggingface/accelerate/default_config.yaml

# Launch training
accelerate launch train_accelerate.py

# Launch with overrides
accelerate launch --num_processes 4 --mixed_precision fp16 \
    train_accelerate.py
```

6.6 Mixed Precision Training (AMP)

Mixed precision training uses both FP16 (or BF16) and FP32 arithmetic. FP16 is used for forward and backward passes (faster, less memory), while FP32 is used for the master weights and loss scaling (numerical stability).

Mixed precision benefits

- **Memory**: FP16 tensors use $\frac{1}{2}$ the memory of FP32 $\Rightarrow$ larger batch sizes or larger models.
- **Speed**: modern GPUs (Ampere, Hopper) have dedicated FP16 / BF16 tensor cores, providing 2–8× throughput.
- **Loss scaling**: gradients in FP16 can underflow. A **loss scaler** multiplies the loss by a large factor before backpropagation, then divides gradients before the optimizer step.

PyTorch AMP example

```
import torch
from torch.amp import autocast, GradScaler

model = SimpleCNN().cuda()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
scaler = GradScaler("cuda")

for data, target in train_loader:
    data, target = data.cuda(), target.cuda()

    optimizer.zero_grad()

    # Forward pass in FP16
    with autocast("cuda"):
        output = model(data)
        loss = criterion(output, target)
```

```
    # Backward pass with loss scaling
    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()
```

Remark 6.3. BF16 (bfloat16) has the same exponent range as FP32, making it more numerically stable than FP16. It is supported on Ampere GPUs and later (A100, H100). When available, prefer BF16 over FP16 to avoid loss scaling altogether.

6.7 Hyperparameter Optimisation with Optuna

Optuna is a hyperparameter optimisation (HPO) framework that uses Bayesian optimisation (Tree-structured Parzen Estimators) to efficiently search the hyperparameter space.

Optuna hyperparameter search

```
import optuna
import torch
import torch.nn as nn
from sklearn.metrics import accuracy_score

def objective(trial):
    # Suggest hyperparameters
    lr = trial.suggest_float("lr", 1e-5, 1e-1, log=True)
    n_layers = trial.suggest_int("n_layers", 1, 4)
    hidden_size = trial.suggest_categorical(
        "hidden_size", [64, 128, 256, 512]
    )
    dropout = trial.suggest_float("dropout", 0.1, 0.5)
    optimizer_name = trial.suggest_categorical(
        "optimizer", ["Adam", "SGD", "AdamW"]
    )

    # Build model dynamically
    layers = []
    in_features = 784
    for i in range(n_layers):
        layers.append(nn.Linear(in_features, hidden_size))
        layers.append(nn.ReLU())
        layers.append(nn.Dropout(dropout))
        in_features = hidden_size
    layers.append(nn.Linear(in_features, 10))
    model = nn.Sequential(*layers).cuda()

    optimizer = getattr(torch.optim, optimizer_name)(
        model.parameters(), lr=lr
    )
```

```
    criterion = nn.CrossEntropyLoss()

    # Train
    for epoch in range(10):
        model.train()
        for data, target in train_loader:
            data, target = data.cuda(), target.cuda()
            optimizer.zero_grad()
            loss = criterion(model(data.view(data.size(0), -1)),
                             target)
            loss.backward()
            optimizer.step()

        # Pruning: report intermediate value
        val_acc = evaluate(model, val_loader)
        trial.report(val_acc, epoch)
        if trial.should_prune():
            raise optuna.TrialPruned()

    return val_acc

# Run the study
study = optuna.create_study(
    direction="maximize",
    pruner=optuna.pruners.MedianPruner(),
    storage="sqlite:///optuna.db",
    study_name="mnist-hpo",
)
study.optimize(objective, n_trials=100, timeout=3600)

# Best result
print(f"Best accuracy: {study.best_value:.4f}")
print(f"Best params: {study.best_params}")
```

HPO best practices

- Use **pruning** (early stopping of bad trials) to save compute—Optuna's median pruner is a good default.
- Use **log-scale** for learning rates and weight decay.
- Start with a **broad search**, then narrow around promising regions.
- Use a **persistent storage** (`sqlite:///optuna.db`) so studies can be resumed.
- Combine with distributed training: run each trial on a GPU.

6.8 Distributed Training Framework Comparison

Criterion	DDP	FSDP	DeepSpeed
Paradigm	Data parallel	Fully sharded	ZeRO stages 1–3
Memory efficiency	Baseline	High	Very high
Model size	≤ 1 GPU	> 1 GPU	> 1 GPU
Complexity	Low	Medium	Medium
Mixed precision	Manual (AMP)	Built-in	Built-in
Offloading (CPU/NVMe)	No	No	Yes
Framework	PyTorch	PyTorch	PyTorch + config
Best for	Standard training	Large models	Very large models

Remark 6.4. **FSDP** (Fully Sharded Data Parallel) shards model parameters, gradients, and optimizer states across GPUs—similar to DeepSpeed ZeRO Stage 3 but natively integrated into PyTorch. For most use cases, start with DDP; move to FSDP or DeepSpeed only when the model does not fit in a single GPU.

6.9 Best Practices

Distributed training golden rules

1. **Start simple**: single GPU → DDP → FSDP/DeepSpeed.
2. **Linear scaling rule**: scale learning rate with the number of GPUs and use warm-up.
3. **Use mixed precision**: always enable AMP (FP16 or BF16) for modern GPUs.
4. **Profile before optimising**: use PyTorch Profiler or `nvidia-smi` to identify bottlenecks.
5. **Checkpoint regularly**: save checkpoints every N steps, not just at epoch boundaries.
6. **Use Accelerate**: for portability across hardware configurations.

Common anti-patterns

- Training in FP32 on modern GPUs (wasting memory and speed)
- Grid search instead of Bayesian HPO (exponentially wasteful)
- Not using a `DistributedSampler` (data duplication across workers)
- Saving checkpoints on all ranks (file corruption, wasted I/O)
- Ignoring the linear scaling rule (divergence with large batch sizes)

6.10 Mini-project: Distributed Training Pipeline

Mini-project 6: Multi-GPU training

Train an image classifier on CIFAR-10 or Fashion-MNIST:

1. Write a single-GPU training script with mixed precision (AMP).
2. Convert it to DDP using `torchrun`.
3. Rewrite it using Hugging Face Accelerate and verify identical results.
4. Run an Optuna HPO sweep (at least 20 trials) over learning rate, batch size, and architecture (number of layers, hidden size).
5. Log all experiments to MLflow or W&B.
6. Compare wall-clock time: 1 GPU vs 2 GPUs vs 4 GPUs.
7. (Bonus) Try FSDP or DeepSpeed for a larger model (e.g., ResNet-50).

6.11 Exercises

Exercise 6.1 (★ — Mixed precision basics)**.** Take an existing single-GPU training script and add mixed precision training using `torch.amp`. Compare:

1. Training time per epoch (FP32 vs FP16 vs BF16)
2. Peak GPU memory usage (`torch.cuda.max_memory_allocated()`)
3. Final model accuracy

Report your findings in a table.

Exercise 6.2 (★★ — DDP implementation)**.** Implement a DDP training script from scratch (without Accelerate) for a ResNet-18 on CIFAR-10:

1. Initialise the process group and wrap the model with DDP
2. Use `DistributedSampler` and set the epoch correctly
3. Add gradient clipping and learning rate scheduling
4. Save checkpoints only on rank 0
5. Verify that training on 2 GPUs with batch size 32 per GPU produces similar results to 1 GPU with batch size 64

Exercise 6.3 (★★★ — Full-scale training system)**.** Build a complete training system for a medium-scale model:

1. Use Accelerate with automatic mixed precision
2. Implement Optuna HPO with pruning and a persistent SQLite study

3. Add DVC data versioning and MLflow experiment tracking
4. Implement checkpointing with automatic resume on failure
5. Write a Prefect flow that orchestrates data preparation, HPO, and final training with the best hyperparameters
6. Produce a final report comparing at least 10 configurations

6.12 Cheatsheet

Chapter 6 Summary

Tool / Concept	Key Points
Data parallelism	Same model on each GPU, AllReduce gradients, scale LR linearly
DDP	`torchrun --nproc_per_node=N`, `DistributedDataParallel`, `DistributedSampler`
Accelerate	`accelerator.prepare()`, `accelerate launch`, hardware-agnostic
Mixed precision	`torch.amp.autocast`, `GradScaler`, BF16 preferred over FP16
Optuna	`study.optimize()`, `trial.suggest_*()`, pruning, persistent storage
FSDP / DeepSpeed	For models too large for a single GPU; shard parameters + gradients + optimizer

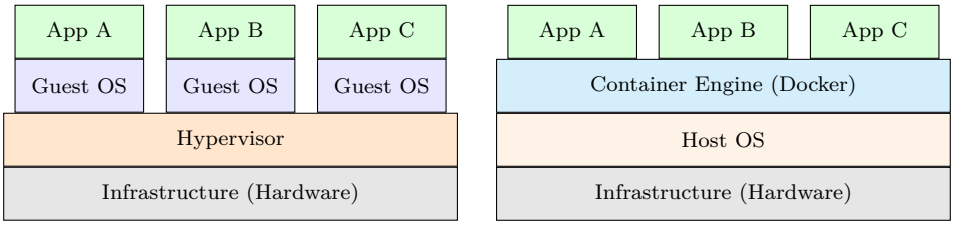
Chapter 7

Containerization — Docker for ML

7.1 Why Containers?

Virtual environments solve Python dependency isolation, but they do *not* capture system-level dependencies: OS libraries, CUDA drivers, compilers, or specific tool versions. Containers solve this by packaging an entire filesystem snapshot that runs identically everywhere.

Definition 7.1 (Container)**.** A **container** is a lightweight, standalone, executable unit of software that includes everything needed to run an application: code, runtime, system tools, libraries, and settings. Unlike virtual machines, containers share the host kernel, making them much faster to start and smaller in footprint.



Remark 7.2. Containers are *not* lightweight VMs. They share the host kernel and use Linux namespaces and cgroups for isolation. This makes them start in milliseconds rather than minutes, and use megabytes rather than gigabytes of overhead.

7.2 Docker Core Concepts

Docker vocabulary

Image A read-only template with instructions for creating a container. Built from a `Dockerfile`.

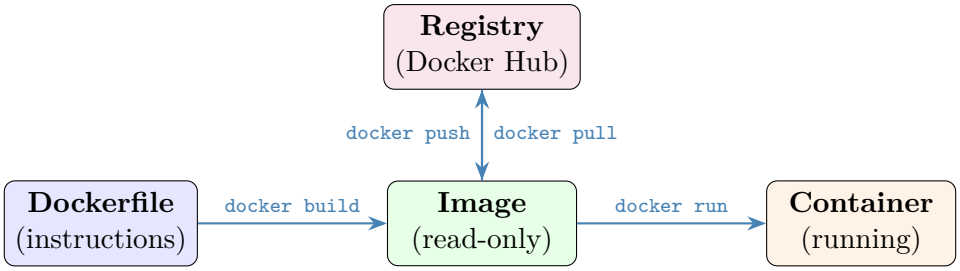
Container A running instance of an image. Ephemeral by default.

Layer Each instruction in a Dockerfile creates a layer. Layers are cached and shared between images.

Registry A repository for storing and distributing images (Docker Hub, GitHub Container Registry, AWS ECR).

Volume Persistent storage that survives container destruction.

Dockerfile A text file with instructions to assemble an image.



7.3 Essential Docker Commands

Docker command reference

```
# Build an image from a Dockerfile
docker build -t my-ml-app:v1 .

# Run a container
docker run -p 8000:8000 my-ml-app:v1

# Run interactively with GPU support
docker run -it --gpus all my-ml-app:v1 bash

# List running containers
docker ps

# Stop and remove a container
docker stop <container_id>
docker rm <container_id>

# List images
docker images

# Remove an image
docker rmi my-ml-app:v1

# View container logs
docker logs -f <container_id>

# Execute a command in a running container
docker exec -it <container_id> bash
```

7.4 Dockerfile for ML Projects

Complete ML Dockerfile with Python and CUDA

```
# Use NVIDIA CUDA base image for GPU support
FROM nvidia/cuda:12.1.0-cudnn8-runtime-ubuntu22.04

# Prevent interactive prompts during build
ENV DEBIAN_FRONTEND=noninteractive
ENV PYTHONUNBUFFERED=1
ENV PYTHONDONTWRITEBYTECODE=1

# Install system dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
    python3.11 \
    python3.11-venv \
    python3-pip \
    git \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Set Python 3.11 as default
RUN ln -sf /usr/bin/python3.11 /usr/bin/python

# Create non-root user
RUN useradd -m -s /bin/bash mluser
WORKDIR /app

# Copy and install dependencies first (layer caching)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY src/ ./src/
COPY models/ ./models/
COPY configs/ ./configs/

# Switch to non-root user
USER mluser

# Expose API port
EXPOSE 8000

# Health check
HEALTHCHECK --interval=30s --timeout=10s --retries=3 \
    CMD curl -f http://localhost:8000/health || exit 1

# Default command
CMD ["python", "-m", "uvicorn", "src.api:app", \
     "--host", "0.0.0.0", "--port", "8000"]
```

Remark 7.3. The order of `COPY` instructions matters for layer caching. Dependencies change less often than source code, so `requirements.txt` is copied and installed *before* the application code. This way, rebuilding after a code change does not re-install all packages.

7.5 Multi-stage Builds

Multi-stage builds produce smaller production images by separating the build environment from the runtime environment.

Multi-stage Dockerfile for ML

```
# ---------- Stage 1: Builder ----------
FROM python:3.11-slim AS builder

WORKDIR /build
COPY requirements.txt .
RUN pip install --no-cache-dir --prefix=/install \
    -r requirements.txt

# ---------- Stage 2: Training ----------
FROM python:3.11-slim AS trainer

WORKDIR /app
COPY --from=builder /install /usr/local
COPY src/ ./src/
COPY configs/ ./configs/
COPY data/ ./data/

RUN python src/train.py --config configs/config.yaml

# ---------- Stage 3: Production ----------
FROM python:3.11-slim AS production

RUN useradd -m mluser
WORKDIR /app

COPY --from=builder /install /usr/local
COPY --from=trainer /app/models/ ./models/
COPY src/api.py ./src/
COPY src/predict.py ./src/

USER mluser
EXPOSE 8000
CMD ["uvicorn", "src.api:app", "--host", "0.0.0.0", \
     "--port", "8000"]
```

Multi-stage build benefits

- The final image contains only runtime dependencies—no compilers, build tools, or training data.
- A typical ML image drops from 5–8 GB to 500 MB–1 GB.
- Smaller images mean faster deployments and reduced attack surface.

7.6 Docker Compose for ML Stacks

Docker Compose orchestrates multi-container applications with a single YAML file.

docker-compose.yml — ML application stack

```yaml
version: "3.9"

services:
  # ML API service
  ml-api:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8000:8000"
    volumes:
      - model-store:/app/models
    environment:
      - MODEL_PATH=/app/models/model.joblib
      - LOG_LEVEL=info
    depends_on:
      db:
        condition: service_healthy
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: 1
              capabilities: [gpu]

  # PostgreSQL for metadata
  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: mlops
      POSTGRES_USER: mluser
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    volumes:
      - pg-data:/var/lib/postgresql/data
```

```yaml
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U mluser"]
      interval: 5s
      timeout: 5s
      retries: 5

  # MLflow tracking server
  mlflow:
    image: ghcr.io/mlflow/mlflow:v2.10.0
    ports:
      - "5000:5000"
    command: >
      mlflow server
      --backend-store-uri postgresql://mluser:${DB_PASSWORD}@db/mlops
      --default-artifact-root /mlflow/artifacts
      --host 0.0.0.0
    volumes:
      - mlflow-artifacts:/mlflow/artifacts
    depends_on:
      db:
        condition: service_healthy

volumes:
  model-store:
  pg-data:
  mlflow-artifacts:
```

Docker Compose commands

```
# Start all services
docker compose up -d

# View logs
docker compose logs -f ml-api

# Rebuild after code change
docker compose up -d --build ml-api

# Stop all services
docker compose down

# Stop and remove volumes (destructive)
docker compose down -v
```

7.7 Best Practices

Docker best practices for ML

1. **Use `.dockerignore`**: exclude data, notebooks, `.git`, `__pycache__`, virtual environments.
2. **Layer caching**: copy dependency files before source code.
3. **Non-root user**: always switch to a non-root user with `USER`.
4. **Pin base image versions**: use `python:3.11.7-slim`, not `python:latest`.
5. **Minimize layers**: combine related `RUN` commands with `&&`.
6. **Clean up**: remove apt caches with `rm -rf /var/lib/apt/lists/*`.
7. **Use `--no-cache-dir`**: prevent pip from storing downloaded packages in the image.
8. **Health checks**: add `HEALTHCHECK` for production images.
9. **Label images**: use `LABEL` for metadata (version, maintainer).

.dockerignore

```
.git
.gitignore
__pycache__
*.pyc
.venv/
data/raw/
notebooks/
*.ipynb
.env
.dockerignore
Dockerfile
docker-compose.yml
README.md
```

Docker anti-patterns for ML

- Using `latest` tag for base images—builds become non-reproducible.
- Running containers as root in production.
- Copying the entire project tree including data and notebooks.
- Installing dev tools (Jupyter, pytest) in production images.
- Storing secrets in the Dockerfile or image layers.
- Building images without `.dockerignore`—context upload takes minutes.

- Using a single monolithic container for training, serving, and monitoring.

7.8 Mini-project: Containerized ML Application

Mini-project 7: Docker for ML

1. Write a `Dockerfile` for the structured ML project from Chapter 1.
2. Use multi-stage builds: one stage for training, one for serving.
3. Create a `.dockerignore` file.
4. Write a `docker-compose.yml` that includes the ML API, a PostgreSQL database, and an MLflow tracking server.
5. Verify that `docker compose up` starts all services and that the API responds to prediction requests.
6. Measure the image size before and after multi-stage optimization.

7.9 Exercises

Exercise 7.1 (★ — Basic Dockerfile)**.** Write a Dockerfile for a scikit-learn project that:

1. Uses `python:3.11-slim` as the base image
2. Installs dependencies from `requirements.txt`
3. Copies the source code
4. Runs the training script as the default command

Build the image and verify it runs correctly.

Exercise 7.2 (★★ — Multi-stage optimization)**.** Starting from a single-stage Dockerfile for a PyTorch project:

1. Measure the original image size with `docker images`
2. Convert to a multi-stage build separating build and runtime
3. Add a `.dockerignore` file
4. Compare the final image sizes and document the savings

Exercise 7.3 (★★★ — Full ML stack with Compose)**.** Create a complete Docker Compose stack for an ML project:

1. ML API (FastAPI) with GPU support
2. PostgreSQL database for predictions logging

3. MLflow tracking server backed by PostgreSQL
4. Prometheus for metrics collection
5. Grafana dashboard for monitoring

All services must have health checks, use named volumes for persistence, and communicate through a custom Docker network.

7.10 Cheatsheet

Chapter 7 Summary

Concept	Key Point
Image vs Container	Image = template (read-only); Container = running instance
`docker build`	Build image from Dockerfile
`docker run`	Create and start a container
Multi-stage	Separate build/runtime for smaller images
Layer caching	Copy `requirements.txt` before source code
`.dockerignore`	Exclude data, notebooks, `.git`
Non-root user	`USER mluser` for security
Docker Compose	Multi-container orchestration via YAML
GPU access	`docker run --gpus all` or Compose `deploy`

Chapter 8

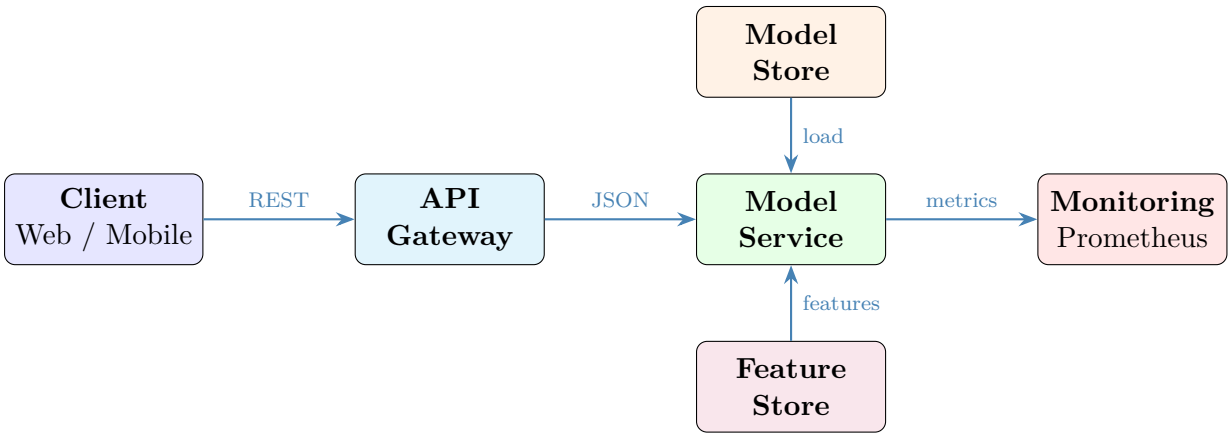
Model Deployment — REST APIs, FastAPI, Streamlit

8.1 Deployment Patterns

Once a model is trained, it must be made available for inference. Three major patterns exist:

Definition 8.1 (Deployment pattern)**.** A **deployment pattern** defines how a trained model serves predictions: in batch on stored data, in real time via an API, or continuously on a data stream.

Pattern	Latency	Use Case	Example
Batch	Minutes–hours	Nightly scoring, report generation	Spark job, Airflow
Online (REST)	Milliseconds	Real-time predictions, user-facing applications	FastAPI, TorchServe
Streaming	Sub-second	Fraud detection, real-time recommendations	Kafka, Flink



8.2 REST API Fundamentals for ML

A REST API exposes the model through HTTP endpoints. The typical contract for an ML service:

Endpoint	Method	Status	Description
`/health`	GET	200	Service health check
`/predict`	POST	200	Single prediction
`/predict/batch`	POST	200	Batch prediction
`/model/info`	GET	200	Model metadata (version, metrics)

8.3 Model Serialization

Before serving, the model must be serialized to a portable format.

Format	Framework	Speed	Notes
joblib / pickle	scikit-learn	Fast	Python-only, security risk
ONNX	Any → ONNX Runtime	Very fast	Cross-framework, optimized
TorchScript	PyTorch	Fast	No Python dependency at inference
SavedModel	TensorFlow	Fast	TF Serving compatible

Model serialization examples

```python
import joblib
import torch
import onnxruntime as ort

# --- scikit-learn with joblib ---
joblib.dump(model, "model.joblib")
model = joblib.load("model.joblib")

# --- PyTorch with TorchScript ---
scripted = torch.jit.script(model)
scripted.save("model.pt")
loaded = torch.jit.load("model.pt")

# --- ONNX export from PyTorch ---
dummy_input = torch.randn(1, 10)
torch.onnx.export(model, dummy_input, "model.onnx",
                  input_names=["input"],
                  output_names=["output"],
                  dynamic_axes={"input": {0: "batch"}})

# --- ONNX inference ---
session = ort.InferenceSession("model.onnx")
result = session.run(None, {"input": input_array})
```

8.4 FastAPI for ML Serving

FastAPI is a modern, high-performance Python web framework ideal for ML serving: automatic validation via Pydantic, async support, and auto-generated OpenAPI documentation.

Complete FastAPI ML serving application

```python
"""ML model serving API with FastAPI."""
import time
from contextlib import asynccontextmanager
from pathlib import Path

import joblib
import numpy as np
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field

# ---------- Pydantic schemas ----------

class PredictionRequest(BaseModel):
    features: list[float] = Field(
        ..., min_length=1,
        description="Input feature vector"
    )

class PredictionResponse(BaseModel):
    prediction: int
    probability: float
    model_version: str
    latency_ms: float

class HealthResponse(BaseModel):
    status: str
    model_loaded: bool
    model_version: str

class BatchRequest(BaseModel):
    instances: list[list[float]] = Field(
        ..., min_length=1,
        description="List of feature vectors"
    )

class BatchResponse(BaseModel):
    predictions: list[int]
    probabilities: list[float]
    count: int

# ---------- Application ----------

MODEL_PATH = Path("models/model.joblib")
```

```python
MODEL_VERSION = "1.0.0"
ml_model = None

@asynccontextmanager
async def lifespan(app: FastAPI):
    """Load model on startup, cleanup on shutdown."""
    global ml_model
    if not MODEL_PATH.exists():
        raise FileNotFoundError(f"Model not found: {MODEL_PATH}")
    ml_model = joblib.load(MODEL_PATH)
    print(f"Model loaded from {MODEL_PATH}")
    yield
    ml_model = None

app = FastAPI(
    title="ML Prediction API",
    version=MODEL_VERSION,
    lifespan=lifespan,
)

@app.get("/health", response_model=HealthResponse)
async def health():
    """Health check endpoint."""
    return HealthResponse(
        status="healthy",
        model_loaded=ml_model is not None,
        model_version=MODEL_VERSION,
    )

@app.post("/predict", response_model=PredictionResponse)
async def predict(request: PredictionRequest):
    """Single prediction endpoint."""
    if ml_model is None:
        raise HTTPException(503, "Model not loaded")
    start = time.perf_counter()
    X = np.array(request.features).reshape(1, -1)
    pred = int(ml_model.predict(X)[0])
    proba = float(ml_model.predict_proba(X).max())
    latency = (time.perf_counter() - start) * 1000
    return PredictionResponse(
        prediction=pred,
        probability=proba,
        model_version=MODEL_VERSION,
        latency_ms=round(latency, 2),
    )

@app.post("/predict/batch", response_model=BatchResponse)
async def predict_batch(request: BatchRequest):
    """Batch prediction endpoint."""
    if ml_model is None:
        raise HTTPException(503, "Model not loaded")
```

```
    X = np.array(request.instances)
    preds = ml_model.predict(X).tolist()
    probas = ml_model.predict_proba(X).max(axis=1).tolist()
    return BatchResponse(
        predictions=preds,
        probabilities=[round(p, 4) for p in probas],
        count=len(preds),
    )
```

Running and testing the API

```
# Run the API
uvicorn src.api:app --host 0.0.0.0 --port 8000 --reload

# Test health endpoint
curl http://localhost:8000/health

# Test prediction
curl -X POST http://localhost:8000/predict \
  -H "Content-Type: application/json" \
  -d '{"features": [5.1, 3.5, 1.4, 0.2]}'

# Interactive docs
# Open http://localhost:8000/docs in a browser
```

8.5 Streamlit for ML Demos

Streamlit enables rapid creation of interactive ML demos without front-end knowledge.

Streamlit ML demo application

```
"""Interactive ML demo with Streamlit."""
import streamlit as st
import requests
import pandas as pd

API_URL = "http://localhost:8000"

st.title("ML Model Explorer")
st.markdown("Interactive prediction demo")

# Sidebar for input
st.sidebar.header("Input Features")
feature_1 = st.sidebar.slider("Sepal Length", 4.0, 8.0, 5.1)
feature_2 = st.sidebar.slider("Sepal Width", 2.0, 4.5, 3.5)
feature_3 = st.sidebar.slider("Petal Length", 1.0, 7.0, 1.4)
feature_4 = st.sidebar.slider("Petal Width", 0.1, 2.5, 0.2)
```

```
if st.button("Predict"):
    response = requests.post(
        f"{API_URL}/predict",
        json={"features": [feature_1, feature_2,
                           feature_3, feature_4]},
    )
    if response.status_code == 200:
        result = response.json()
        st.success(f"Prediction: {result['prediction']}")
        st.metric("Confidence", f"{result['probability']:.2%}")
        st.metric("Latency", f"{result['latency_ms']:.1f} ms")
    else:
        st.error(f"API error: {response.status_code}")
```

8.6 Serving Framework Comparison

Criterion	FastAPI	Flask	TorchServe	Triton
Ease of use	High	High	Medium	Low
Performance	High	Medium	High	Very high
Auto-scaling	Manual	Manual	Built-in	Built-in
Multi-model	Manual	Manual	Yes	Yes
GPU batching	Manual	No	Yes	Yes
Framework lock-in	None	None	PyTorch	None
OpenAPI docs	Auto	Manual	No	No
Best for	Custom APIs	Simple APIs	PyTorch prod	High-perf prod

Remark 8.2. FastAPI is the best starting point for most ML projects: it is simple, performant, and flexible. Migrate to TorchServe or Triton when you need advanced features like dynamic batching, multi-model serving, or GPU-optimized inference at scale.

8.7 Best Practices and Anti-patterns

Deployment best practices

1. **Version your models**: include the model version in every response.
2. **Validate inputs**: use Pydantic schemas with constraints.
3. **Health checks**: always expose a `/health` endpoint.
4. **Measure latency**: return inference time in the response.
5. **Graceful startup**: load the model before accepting requests (lifespan events).
6. **Log predictions**: store inputs and outputs for monitoring and debugging.
7. **Separate model and API**: the API code should not contain training logic.

Deployment anti-patterns

- Loading the model on every request instead of at startup.
- No input validation—malformed data causes cryptic errors.
- Returning raw numpy arrays instead of JSON-serializable types.
- Hardcoding model paths without environment variable overrides.
- No error handling—a failed prediction crashes the server.
- Running the API without HTTPS in production.

8.8 Mini-project: End-to-end Deployment

Mini-project 8: Deploy an ML model

1. Train a scikit-learn classifier and serialize it with `joblib`.
2. Build a FastAPI application with `/health`, `/predict`, and `/predict/batch` endpoints.
3. Add Pydantic input validation with meaningful error messages.
4. Write a Streamlit front-end that calls the API.
5. Containerize both services with Docker Compose.
6. Export the model to ONNX format and compare inference latency against joblib.

8.9 Exercises

Exercise 8.1 (★ — Basic API)**.** Create a FastAPI application that serves a pre-trained `scikit-learn` model. The API must have a `/health` endpoint and a `/predict` endpoint with Pydantic validation. Test with `curl` or the automatic Swagger documentation.

Exercise 8.2 (★★ — ONNX conversion)**.** Take a trained PyTorch model and:

1. Export it to ONNX format with dynamic batch size
2. Build a FastAPI endpoint that uses ONNX Runtime for inference
3. Benchmark the latency of PyTorch vs ONNX Runtime on 100 requests
4. Document the performance difference

Exercise 8.3 (★★★ — Production-grade API)**.** Build a production-ready ML API with:

1. FastAPI with async endpoints
2. A/B testing support (serve two model versions, route by header)
3. Request logging to a database (async writes)
4. Rate limiting with `slowapi`
5. Prometheus metrics endpoint (`/metrics`)
6. Docker Compose deployment with Nginx reverse proxy

8.10 Cheatsheet

Chapter 8 Summary

Concept	Key Point
Batch vs Online	Batch = scheduled; Online = real-time REST API
FastAPI	High-perf Python API with Pydantic + async
Pydantic	Input/output validation via type annotations
Model formats	joblib (sklearn), ONNX (cross-framework), TorchScript
Streamlit	Rapid ML demos with pure Python
Health check	Always expose `GET /health`
Lifespan events	Load model at startup, not per request

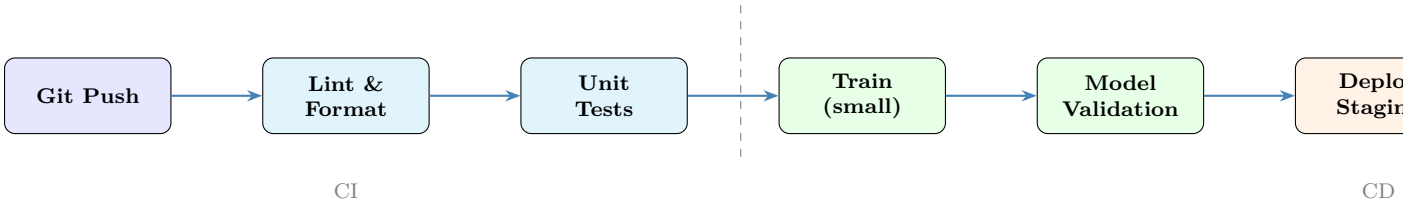
Chapter 9

CI/CD for Machine Learning

9.1 Why CI/CD for ML?

Traditional CI/CD pipelines test code and deploy applications. ML pipelines must additionally validate data, test model quality, and manage artifacts that are much larger than compiled binaries.

Definition 9.1 (CI/CD for ML)**. Continuous Integration** for ML verifies that code changes do not break the training pipeline, that data schemas are respected, and that model performance meets minimum thresholds. **Continuous Deployment** automates the promotion of validated models to staging and production environments.



Remark 9.2. The key difference between CI/CD for software and CI/CD for ML is the **model validation** step: the pipeline must verify that the newly trained model meets performance thresholds before deployment. A model that passes all unit tests can still be useless if accuracy has regressed.

9.2 Pre-commit Hooks

Pre-commit hooks run checks *before* code enters the repository, catching issues at the earliest possible stage.

.pre-commit-config.yaml

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.5.0
    hooks:
      - id: trailing-whitespace
```

```yaml
      - id: end-of-file-fixer
      - id: check-yaml
      - id: check-added-large-files
        args: ['--maxkb=1000']

  - repo: https://github.com/psf/black
    rev: 24.2.0
    hooks:
      - id: black

  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.3.0
    hooks:
      - id: ruff
        args: [--fix]

  - repo: https://github.com/pre-commit/mirrors-mypy
    rev: v1.8.0
    hooks:
      - id: mypy
        additional_dependencies: [numpy, pandas-stubs]
```

Setting up pre-commit

```bash
# Install pre-commit
pip install pre-commit

# Install hooks in the repository
pre-commit install

# Run on all files (first time)
pre-commit run --all-files

# Update hooks to latest versions
pre-commit autoupdate
```

9.3 GitHub Actions for ML

GitHub Actions is the most popular CI/CD platform for open-source ML projects. Workflows are defined in YAML files under `.github/workflows/`.

.github/workflows/ml-ci.yml — Complete ML CI pipeline

```yaml
name: ML CI Pipeline

on:
  push:
```

```yaml
    branches: [main, develop]
  pull_request:
    branches: [main]

env:
  PYTHON_VERSION: "3.11"
  MLFLOW_TRACKING_URI: "sqlite:///mlflow.db"

jobs:
  lint-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: ${{ env.PYTHON_VERSION }}
          cache: pip

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install -r requirements-dev.txt

      - name: Lint with ruff
        run: ruff check src/ tests/

      - name: Format check with black
        run: black --check src/ tests/

      - name: Type check with mypy
        run: mypy src/ --ignore-missing-imports

      - name: Unit tests
        run: pytest tests/unit/ -v --tb=short

  train-and-validate:
    needs: lint-and-test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: ${{ env.PYTHON_VERSION }}
          cache: pip

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

```yaml
      - name: Set up DVC
        uses: iterative/setup-dvc@v2

      - name: Pull data with DVC
        run: dvc pull
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_KEY }}

      - name: Train model (small dataset)
        run: |
          python src/train.py \
            --config configs/ci_config.yaml \
            --max-samples 1000

      - name: Validate model performance
        run: |
          python src/evaluate.py \
            --model models/model.joblib \
            --min-accuracy 0.85 \
            --min-f1 0.80

      - name: Upload model artifact
        uses: actions/upload-artifact@v4
        with:
          name: trained-model
          path: models/model.joblib
          retention-days: 7

  integration-test:
    needs: train-and-validate
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres:16-alpine
        env:
          POSTGRES_DB: test_db
          POSTGRES_USER: test_user
          POSTGRES_PASSWORD: test_pass
        ports:
          - 5432:5432
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: ${{ env.PYTHON_VERSION }}

      - name: Download model artifact
```

```yaml
        uses: actions/download-artifact@v4
        with:
          name: trained-model
          path: models/

      - name: Install dependencies
        run: pip install -r requirements.txt

      - name: Run API integration tests
        run: pytest tests/integration/ -v
```

9.4 Model Validation in CI

Model validation ensures that newly trained models meet minimum quality standards before deployment.

src/evaluate.py — Model validation script

```python
"""Model validation for CI pipeline."""
import argparse
import json
import sys

import joblib
import pandas as pd
from sklearn.metrics import (
    accuracy_score, f1_score, precision_score, recall_score
)


def validate_model(
    model_path: str,
    test_data_path: str = "data/test.csv",
    min_accuracy: float = 0.85,
    min_f1: float = 0.80,
) -> bool:
    """Validate model against minimum thresholds."""
    model = joblib.load(model_path)
    df = pd.read_csv(test_data_path)
    X_test = df.drop("target", axis=1)
    y_test = df["target"]

    y_pred = model.predict(X_test)

    metrics = {
        "accuracy": accuracy_score(y_test, y_pred),
        "f1": f1_score(y_test, y_pred, average="weighted"),
        "precision": precision_score(y_test, y_pred,
```

```
                                    average="weighted"),
        "recall": recall_score(y_test, y_pred,
                               average="weighted"),
    }

    # Save metrics report
    with open("metrics.json", "w") as f:
        json.dump(metrics, f, indent=2)

    print("=== Model Validation Report ===")
    for name, value in metrics.items():
        status = "PASS" if value >= locals().get(
            f"min_{name}", 0.0) else "FAIL"
        print(f"  {name}: {value:.4f} [{status}]")

    passed = (
        metrics["accuracy"] >= min_accuracy
        and metrics["f1"] >= min_f1
    )
    print(f"\nOverall: {'PASSED' if passed else 'FAILED'}")
    return passed


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--model", required=True)
    parser.add_argument("--test-data", default="data/test.csv")
    parser.add_argument("--min-accuracy", type=float, default=0.85)
    parser.add_argument("--min-f1", type=float, default=0.80)
    args = parser.parse_args()

    success = validate_model(
        args.model, args.test_data,
        args.min_accuracy, args.min_f1,
    )
    sys.exit(0 if success else 1)
```

9.5 DVC in CI

Data Version Control (DVC) integrates with CI pipelines to pull versioned data and reproduce training pipelines.

DVC integration in GitHub Actions

```
- name: Set up DVC
  uses: iterative/setup-dvc@v2

- name: Configure DVC remote
  run: |
```

```
      dvc remote modify myremote access_key_id \
        ${{ secrets.AWS_ACCESS_KEY_ID }}
      dvc remote modify myremote secret_access_key \
        ${{ secrets.AWS_SECRET_KEY }}

  - name: Pull data
    run: dvc pull

  - name: Reproduce pipeline
    run: dvc repro

  - name: Compare metrics
    run: |
      dvc metrics diff --md >> $GITHUB_STEP_SUMMARY
```

Remark 9.3. Use `dvc metrics diff` in pull requests to automatically show how a code change affects model performance. The CML (Continuous Machine Learning) tool by Iterative can post these comparisons as PR comments.

9.6 CI/CD Platform Comparison

Criterion	GitHub Actions	GitLab CI	Jenkins	CML
Hosted	Yes	Yes	Self-hosted	Cloud
GPU runners	Paid	Yes	Self-hosted	Yes
Config format	YAML	YAML	Groovy/YAML	YAML
ML-specific	No	No	No	Yes
Marketplace	Large	Medium	Large	Small
Free tier	Generous	Generous	Free	Free
Best for	Open source	GitLab users	Enterprise	ML teams

9.7 Best Practices and Anti-patterns

CI/CD best practices for ML

1. **Fast CI**: use a small data subset for CI training (full training offline or nightly).
2. **Model performance gates**: fail the pipeline if metrics drop below thresholds.
3. **Cache dependencies**: use pip/conda caching to speed up builds.
4. **Secrets management**: never hardcode API keys; use repository secrets.
5. **Artifact management**: upload trained models as CI artifacts for traceability.

6. **Separate CI config**: use a lightweight `ci_config.yaml` with fewer epochs and smaller data.

CI/CD anti-patterns

- Training on the full dataset in CI—pipelines take hours.
- No model validation step—broken models reach production.
- Storing large data files in the Git repository.
- Skipping linting and type checking for ML code.
- No integration tests for the serving API.
- Manual deployment after CI passes—defeats the purpose of CD.

9.8 Mini-project: ML CI/CD Pipeline

Mini-project 9: CI/CD for ML

1. Set up `pre-commit` with black, ruff, and mypy hooks.
2. Write a GitHub Actions workflow that:
   - Runs linting and unit tests
   - Trains a model on a small subset
   - Validates model performance against thresholds
3. Add a model validation script that exits with code 1 if metrics are below thresholds.
4. Configure DVC to pull test data in the CI pipeline.
5. Add a deployment step to staging (Docker build + push to a registry).
6. Verify the full pipeline runs on a pull request.

9.9 Exercises

Exercise 9.1 (★ — Pre-commit setup)**.** Configure `pre-commit` for an ML project with hooks for: black, ruff, mypy, trailing whitespace, and large file detection (max 500 KB). Run it on an existing project and fix all issues.

Exercise 9.2 (★★ — GitHub Actions workflow)**.** Write a GitHub Actions workflow for an ML project that:

1. Runs on push to `main` and on pull requests
2. Uses a matrix strategy to test on Python 3.10 and 3.11

3. Caches pip dependencies
4. Runs unit tests and generates a coverage report
5. Uploads the coverage report as an artifact

Exercise 9.3 (★★★ — Complete ML CI/CD)**.** Build an end-to-end CI/CD pipeline that:

1. Pulls versioned data with DVC
2. Trains a model and logs metrics to MLflow
3. Compares metrics against the current production model
4. Builds a Docker image and pushes to GitHub Container Registry
5. Deploys to a staging environment
6. Posts a metric comparison table as a PR comment using CML

9.10 Cheatsheet

Chapter 9 Summary

Concept	Key Point
CI for ML	Lint + test + train (small) + validate metrics
CD for ML	Auto-deploy if model passes validation gates
Pre-commit	Local hooks: black, ruff, mypy, large file check
GitHub Actions	YAML workflows in `.github/workflows/`
Model validation	Script that fails CI if accuracy/F1 too low
DVC in CI	`dvc pull` + `dvc repro` + `dvc metrics diff`
Secrets	Use repository secrets, never hardcode

Chapter 10

Model Monitoring in Production

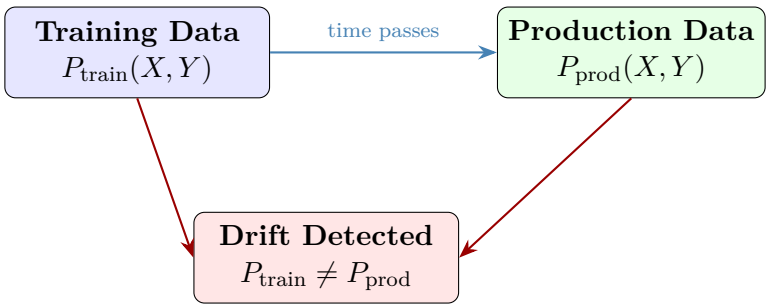
10.1 Why Monitor ML Models?

Unlike traditional software, ML models degrade silently. A deployed model can return valid HTTP 200 responses while producing increasingly wrong predictions. Monitoring detects this degradation before it impacts business outcomes.

Definition 10.1 (Model drift)**. Model drift** is the degradation of model performance over time due to changes in the statistical properties of the input data, the target variable, or the relationship between them.

10.2 Types of Drift

Type	Definition	Example
Data drift	Input distribution $P(X)$ changes	User demographics shift after a marketing campaign
Concept drift	Relationship $P(Y\|X)$ changes	Customer churn factors change during a recession
Covariate shift	Training $P(X)$ differs from production $P(X)$	Model trained on urban data deployed to rural areas
Label drift	Target distribution $P(Y)$ changes	Fraud rate increases from 1% to 5%



10.3 Drift Detection Methods

10.3.1 Kullback–Leibler divergence

The KL divergence measures how one probability distribution differs from a reference distribution:

$$D_{\mathrm{KL}}(P \,\|\, Q) = \sum_{x} P(x) \ln \frac{P(x)}{Q(x)} \tag{10.1}$$

$D_{\mathrm{KL}} = 0$ means $P = Q$; larger values indicate greater divergence. Note that KL divergence is *not* symmetric.

10.3.2 Population Stability Index (PSI)

PSI is a symmetric measure commonly used in credit scoring and ML monitoring:

$$\mathrm{PSI} = \sum_{i=1}^{B} \left(p_i^{\mathrm{prod}} - p_i^{\mathrm{train}}\right) \ln \frac{p_i^{\mathrm{prod}}}{p_i^{\mathrm{train}}} \tag{10.2}$$

where B is the number of bins and p_i is the proportion in bin i.

PSI Value	Interpretation
< 0.10	No significant shift
0.10–0.25	Moderate shift—investigate
> 0.25	Major shift—retrain

10.3.3 Kolmogorov–Smirnov test

The two-sample KS test compares the empirical cumulative distribution functions (ECDFs) of two samples:

$$D = \sup_{x} \left| F_{\mathrm{train}}(x) - F_{\mathrm{prod}}(x) \right| \tag{10.3}$$

The null hypothesis H_0: both samples come from the same distribution. A small p-value (< 0.05) indicates significant drift.

10.4 Drift Detection with Evidently AI

Data drift detection with Evidently

```python
"""Drift detection using Evidently AI."""
import pandas as pd
from evidently import ColumnMapping
from evidently.report import Report
from evidently.metric_preset import DataDriftPreset
from evidently.metrics import (
    DatasetDriftMetric,
    DataDriftTable,
    ColumnDriftMetric,
)
```

```python
# Load reference (training) and current (production) data
reference = pd.read_csv("data/train.csv")
current = pd.read_csv("data/production_batch.csv")

column_mapping = ColumnMapping(
    target="target",
    numerical_features=["feature_1", "feature_2", "feature_3"],
    categorical_features=["category_a", "category_b"],
)

# Generate drift report
report = Report(metrics=[
    DataDriftPreset(),
    DatasetDriftMetric(),
    DataDriftTable(),
    ColumnDriftMetric(column_name="feature_1"),
])

report.run(
    reference_data=reference,
    current_data=current,
    column_mapping=column_mapping,
)

# Save as HTML dashboard
report.save_html("drift_report.html")

# Get results programmatically
result = report.as_dict()
dataset_drift = result["metrics"][1]["result"]["dataset_drift"]
print(f"Dataset drift detected: {dataset_drift}")
```

Evidently test suite for CI

```python
"""Drift tests for CI pipeline."""
from evidently.test_suite import TestSuite
from evidently.tests import (
    TestShareOfDriftedColumns,
    TestColumnDrift,
    TestMeanInNSigmas,
)

suite = TestSuite(tests=[
    TestShareOfDriftedColumns(lt=0.3),  # < 30% drifted
    TestColumnDrift("feature_1"),
    TestColumnDrift("feature_2"),
    TestMeanInNSigmas("feature_1", n=3),
])
```

```
suite.run(reference_data=reference, current_data=current,
          column_mapping=column_mapping)
suite.save_html("drift_tests.html")

# Use in CI: exit 1 if tests fail
test_results = suite.as_dict()
all_passed = all(
    t["status"] == "SUCCESS"
    for t in test_results["tests"]
)
if not all_passed:
    raise SystemExit("Drift tests FAILED")
```

10.5 Prometheus and Grafana

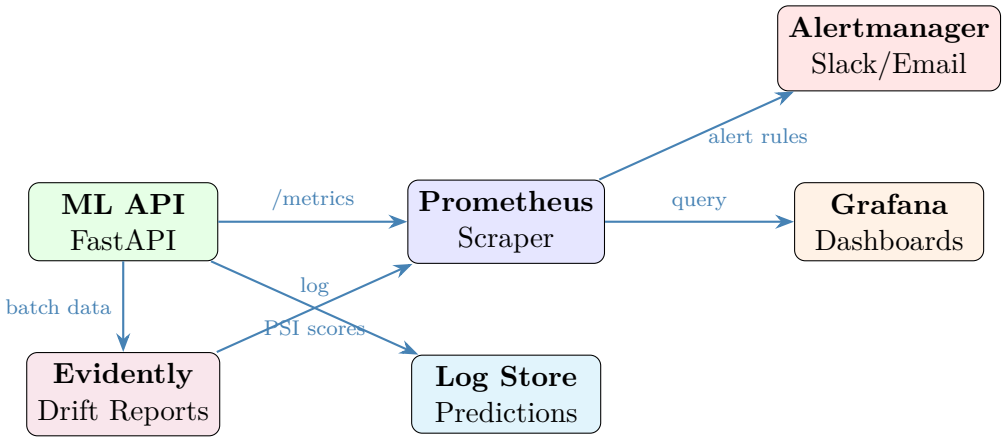
Prometheus collects time-series metrics; Grafana visualizes them in dashboards. Together they form the standard monitoring stack for ML APIs.

Prometheus metrics in FastAPI

```
"""FastAPI with Prometheus metrics."""
from prometheus_client import (
    Counter, Histogram, Gauge, generate_latest
)
from fastapi import FastAPI, Response

# Define metrics
PREDICTIONS_TOTAL = Counter(
    "ml_predictions_total",
    "Total predictions",
    ["model_version", "prediction_class"],
)
PREDICTION_LATENCY = Histogram(
    "ml_prediction_latency_seconds",
    "Prediction latency in seconds",
    buckets=[0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1.0],
)
MODEL_CONFIDENCE = Histogram(
    "ml_prediction_confidence",
    "Prediction confidence score",
    buckets=[0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 0.99],
)
DRIFT_SCORE = Gauge(
    "ml_drift_score",
    "Current drift score (PSI)",
)

app = FastAPI()
```

```python
@app.post("/predict")
async def predict(request: PredictionRequest):
    with PREDICTION_LATENCY.time():
        result = model.predict(request.features)
    PREDICTIONS_TOTAL.labels(
        model_version="1.0.0",
        prediction_class=str(result.label),
    ).inc()
    MODEL_CONFIDENCE.observe(result.confidence)
    return result

@app.get("/metrics")
async def metrics():
    """Prometheus metrics endpoint."""
    return Response(
        content=generate_latest(),
        media_type="text/plain",
    )
```

10.6 Monitoring Architecture



Key metrics to monitor

Operational Request latency (p50, p95, p99), throughput (req/s), error rate, uptime.

Model quality Prediction distribution, confidence distribution, feature drift (PSI per column).

Data quality Missing value rate, schema violations, out-of-range values, input volume changes.

Infrastructure CPU/GPU utilization, memory usage, model load time.

10.7 Alerting Strategy

Alert fatigue

Too many alerts lead to alert fatigue—teams start ignoring them. Define clear severity levels and only page on-call for critical issues:

- **Critical**: service down, error rate $> 5\%$
- **Warning**: latency p95 > 500 ms, drift PSI > 0.25
- **Info**: drift PSI > 0.10, prediction volume change $> 30\%$

10.8 Monitoring Tool Comparison

Criterion	Evidently AI	WhyLabs	NannyML
Open source	Yes	Freemium	Yes
Drift detection	Yes	Yes	Yes
Performance estimation	No	No	Yes (CBPE)
Real-time	Batch	Real-time	Batch
Dashboard	HTML reports	Cloud UI	Built-in
CI integration	Test suites	API	API
Best for	Batch monitoring	Streaming	No-label monitoring

Remark 10.2. NannyML's key differentiator is **CBPE** (Confidence-Based Performance Estimation): it estimates model performance *without ground truth labels*, which are often delayed days or weeks in production.

10.9 Best Practices and Anti-patterns

Monitoring best practices

1. **Log all predictions**: store inputs, outputs, confidence, and timestamps.
2. **Establish baselines**: compute reference metrics on validation data before deployment.
3. **Monitor features individually**: aggregate drift masks which features changed.
4. **Automate drift checks**: run Evidently test suites on a schedule (daily or weekly).
5. **Connect monitoring to retraining**: trigger retraining when drift exceeds thresholds.
6. **Dashboard per model**: each deployed model gets its own Grafana dashboard.

Monitoring anti-patterns

- Monitoring only infrastructure (CPU, memory) but not model quality.
- No baseline reference data for drift comparison.
- Alerting on every small fluctuation—causes alert fatigue.
- Waiting for user complaints to discover model degradation.
- Not logging prediction inputs—impossible to debug issues.
- Monitoring in production but not in staging.

10.10 Mini-project: Monitoring Pipeline

Mini-project 10: Model monitoring

1. Deploy a trained model with FastAPI and add Prometheus metrics (latency, predictions count, confidence histogram).
2. Write a script that simulates production traffic with gradual data drift.
3. Set up Evidently to generate drift reports comparing training data to production batches.
4. Configure Prometheus to scrape the API's `/metrics` endpoint.
5. Build a Grafana dashboard with panels for latency, throughput, prediction distribution, and drift score.
6. Configure an alert rule that fires when PSI exceeds 0.25.

10.11 Exercises

Exercise 10.1 (★ — Drift detection basics)**.** Using Evidently AI, generate a data drift report comparing two synthetic datasets. Create a reference dataset with `np.random.normal(0, 1)` and a drifted dataset with `np.random.normal(0.5, 1.2)`. Interpret the PSI values and the KS test results.

Exercise 10.2 (★★ — Custom monitoring)**.** Add Prometheus metrics to the FastAPI application from Chapter 8:

1. A counter for total predictions per class
2. A histogram for prediction latency
3. A histogram for confidence scores
4. A gauge for the latest drift score

Write a Prometheus scrape configuration and verify the metrics appear.

Exercise 10.3 (★★★ — Full monitoring stack)**.** Build a complete monitoring system:

1. Deploy the ML API with Docker Compose
2. Add Prometheus, Grafana, and Alertmanager as services
3. Create a Grafana dashboard with at least six panels
4. Simulate concept drift by gradually changing the data distribution
5. Configure alerts that fire when drift is detected
6. Document the alerting and retraining decision process

10.12 Cheatsheet

Chapter 10 Summary

Concept	Key Point
Data drift	$P(X)$ changes over time
Concept drift	$P(Y\|X)$ changes over time
PSI	< 0.10 OK; 0.10–0.25 investigate; > 0.25 retrain
KS test	Compares ECDFs; $p < 0.05$ means significant drift
Evidently AI	Open-source drift reports and test suites
Prometheus	Time-series metrics collection
Grafana	Dashboard visualization and alerting
Key metrics	Latency, throughput, confidence, drift score

Chapter 11

Reproducibility in Research — Standards and Best Practices

11.1 The Reproducibility Crisis in ML

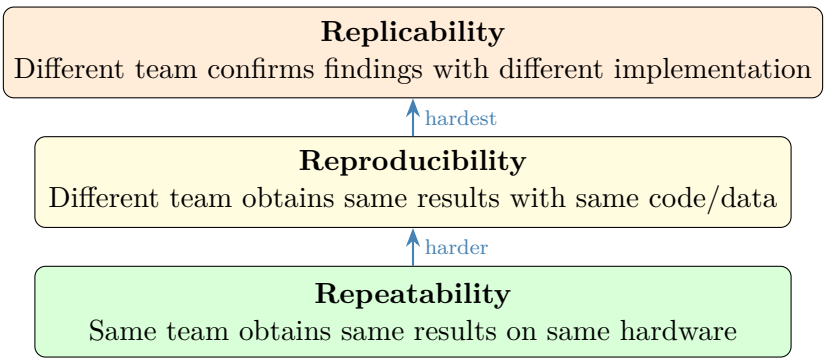
A growing body of evidence shows that many published ML results cannot be reproduced. Surveys report that over 60% of ML experiments lack sufficient detail for independent verification. The causes are multiple: missing code, undocumented hyperparameters, unreported hardware configurations, and non-deterministic training procedures.

Definition 11.1 (Reproducibility)**. Reproducibility** is the ability for an independent team to obtain the same results using the *same* code and data. It is distinct from *replicability*, which aims to confirm findings using *different* code or data.

11.2 Levels of Reproducibility

The ACM distinguishes three levels with increasing stringency:

Level	Definition	Requirements
Repeatability	Same team, same setup	Fixed seeds, versioned code, deterministic pipeline
Reproducibility	Different team, same artifacts	Published code, data, environment specs, clear instructions
Replicability	Different team, different setup	Detailed methodology, model cards, independent confirmation



CHAPTER 11. REPRODUCIBILITY IN RESEARCH — STANDARDS AND BEST PRACTICES

11.3 Random Seed Management

Non-determinism is the primary obstacle to repeatability. ML pipelines have multiple sources of randomness that must all be controlled.

Complete random seed management

```python
"""Comprehensive seed management for reproducibility."""
import os
import random

import numpy as np
import torch


def set_seed(seed: int = 42) -> None:
    """Set all random seeds for reproducibility.

    Controls randomness in:
    - Python's random module
    - NumPy
    - PyTorch (CPU and CUDA)
    - CUDA convolution algorithms
    - Hash-based operations
    """
    # Python
    random.seed(seed)
    os.environ["PYTHONHASHSEED"] = str(seed)

    # NumPy
    np.random.seed(seed)

    # PyTorch CPU
    torch.manual_seed(seed)

    # PyTorch CUDA (all GPUs)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

    # Deterministic algorithms (may reduce performance)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

    # PyTorch 2.0+ deterministic mode
    torch.use_deterministic_algorithms(True)
    os.environ["CUBLAS_WORKSPACE_CONFIG"] = ":4096:8"


# Use at the very start of training
set_seed(42)
```

Limits of determinism on GPU

Even with all seeds fixed, some GPU operations remain non-deterministic by default:

- Atomic operations in CUDA (e.g., `scatter_add`)
- cuDNN auto-tuner selects different algorithms per run
- Multi-threaded data loading with `DataLoader` (`num_workers > 0`)

Setting `torch.use_deterministic_algorithms(True)` forces deterministic alternatives but may slow down training by 10–20%.

Deterministic DataLoader

```python
"""Reproducible data loading."""
import torch
from torch.utils.data import DataLoader


def seed_worker(worker_id: int) -> None:
    """Seed each DataLoader worker for reproducibility."""
    worker_seed = torch.initial_seed() % 2**32
    np.random.seed(worker_seed)
    random.seed(worker_seed)


generator = torch.Generator()
generator.manual_seed(42)

train_loader = DataLoader(
    dataset,
    batch_size=32,
    shuffle=True,
    num_workers=4,
    worker_init_fn=seed_worker,
    generator=generator,
)
```

11.4 Model Cards

A **model card** (Mitchell et al., 2019) is a standardized document that accompanies a trained model, providing transparency about its intended use, performance, and limitations.

Model card template

1. **Model Details**: architecture, version, training date, framework, license.

2. **Intended Use**: primary use cases, out-of-scope uses.
3. **Training Data**: dataset description, size, preprocessing, known biases.
4. **Evaluation Data**: test set description, evaluation protocol.
5. **Metrics**: accuracy, F1, AUC—disaggregated by demographic group.
6. **Ethical Considerations**: potential harms, biases, fairness analysis.
7. **Caveats and Recommendations**: known limitations, failure modes.

Model card generation with Python

```python
"""Generate a model card as structured YAML."""
import yaml
from datetime import date

model_card = {
    "model_details": {
        "name": "Fraud Detection Classifier v2.1",
        "architecture": "XGBoost",
        "framework": "xgboost==2.0.3",
        "training_date": str(date.today()),
        "license": "MIT",
    },
    "intended_use": {
        "primary": "Detect fraudulent credit card transactions",
        "out_of_scope": [
            "Non-financial fraud detection",
            "Real-time streaming (latency > 100ms)",
        ],
    },
    "training_data": {
        "dataset": "Internal transactions 2022-2024",
        "size": "2.4M transactions",
        "fraud_rate": "1.2%",
        "preprocessing": "SMOTE oversampling, standard scaling",
    },
    "metrics": {
        "overall": {"accuracy": 0.987, "f1": 0.912, "auc": 0.975},
        "by_region": {
            "north_america": {"f1": 0.923},
            "europe": {"f1": 0.908},
            "asia": {"f1": 0.891},
        },
    },
    "limitations": [
        "Lower performance on transactions > $10,000",
        "Not validated for cryptocurrency transactions",
        "Requires retraining quarterly due to drift",
```

```
    ],
}

with open("model_card.yaml", "w") as f:
    yaml.dump(model_card, f, default_flow_style=False, sort_keys=False)
```

11.5 Datasheets for Datasets

Datasheets for Datasets (Gebru et al., 2021) standardize dataset documentation. Key sections:

Datasheet sections

Motivation Why was the dataset created? Who funded it?

Composition What do instances represent? How many? Any missing data?

Collection process How was data acquired? Consent process?

Preprocessing What cleaning or transformations were applied?

Uses Intended tasks? Tasks to avoid?

Distribution How is the dataset shared? License?

Maintenance Who maintains it? Update frequency? Deprecation policy?

11.6 NeurIPS Reproducibility Checklist

Major ML conferences now require authors to complete a reproducibility checklist. The NeurIPS checklist includes:

NeurIPS-style reproducibility checklist

1. Code is submitted (or will be released upon acceptance).
2. All hyperparameters are reported, including search ranges.
3. Training and evaluation datasets are described or cited.
4. Compute resources used are documented (GPU type, hours).
5. Number of runs and variance/confidence intervals are reported.
6. Random seeds are specified.
7. Data preprocessing steps are fully described.
8. Software dependencies are listed with versions.
9. Statistical significance tests are included where relevant.

10. Negative results and failure cases are discussed.

Remark 11.2. Reporting the mean and standard deviation over multiple runs (e.g., 5 runs with different seeds) is far more informative than a single best result. A model with $92.1 \pm 0.3\%$ accuracy is more trustworthy than one that claims 93.0% from a single lucky seed.

11.7 Practical Reproducibility Toolkit

Reproducibility configuration

```yaml
# configs/experiment.yaml
experiment:
  name: "fraud-detection-v2"
  seed: 42
  deterministic: true

data:
  path: "data/transactions.csv"
  dvc_hash: "a1b2c3d4"  # DVC file hash for traceability
  test_size: 0.2
  stratify: "target"

model:
  type: "xgboost"
  params:
    n_estimators: 500
    max_depth: 8
    learning_rate: 0.05
    random_state: 42

training:
  n_runs: 5  # Multiple runs for confidence intervals
  seeds: [42, 123, 456, 789, 1024]

environment:
  python: "3.11.7"
  gpu: "NVIDIA A100"
  cuda: "12.1"
  requirements_hash: "sha256:e5f6..."
```

Multi-run experiment with statistics

```python
"""Run experiment multiple times for robust evaluation."""
import json
import numpy as np
from scipy import stats
```

```python
def run_experiment(seed: int, config: dict) -> dict:
    """Run a single experiment with a given seed."""
    set_seed(seed)
    # ... training code ...
    return {"accuracy": acc, "f1": f1, "auc": auc}


seeds = [42, 123, 456, 789, 1024]
results = [run_experiment(s, config) for s in seeds]

# Aggregate results
for metric in ["accuracy", "f1", "auc"]:
    values = [r[metric] for r in results]
    mean = np.mean(values)
    std = np.std(values, ddof=1)
    ci = stats.t.interval(
        0.95, len(values) - 1,
        loc=mean, scale=stats.sem(values),
    )
    print(f"{metric}: {mean:.4f} +/- {std:.4f} "
          f"(95% CI: [{ci[0]:.4f}, {ci[1]:.4f}])")
```

11.8 Best Practices and Anti-patterns

Reproducibility best practices

1. **Fix all seeds**: Python, NumPy, PyTorch, CUDA, data loaders.
2. **Version everything**: code (Git), data (DVC), experiments (MLflow), environments (Docker).
3. **Report multiple runs**: mean $\pm$ std over ≥ 3 seeds.
4. **Write model cards**: document intended use, limitations, and biases.
5. **Publish code and data**: use GitHub and Zenodo or Hugging Face Datasets.
6. **Log compute resources**: GPU type, training time, energy consumption.
7. **Use deterministic algorithms**: accept the performance cost for critical experiments.

Reproducibility anti-patterns

- Cherry-picking the best run out of many.
- "Code available upon request"—almost never provided.
- Not reporting hyperparameter search ranges.

- Using `latest` Docker tags or unpinned dependencies.
- Ignoring non-determinism because "results are approximately the same".
- Not documenting data preprocessing steps.

11.9 Mini-project: Reproducible Experiment

Mini-project 11: Reproducibility

1. Implement a complete `set_seed()` function covering Python, NumPy, and PyTorch.
2. Run the same experiment 5 times with different seeds and report mean ± std with 95% confidence intervals.
3. Write a model card for the trained model following the template above.
4. Write a datasheet for the dataset used.
5. Create a `README.md` with step-by-step reproduction instructions.
6. Verify that a classmate can reproduce your results within ±1% by following your instructions alone.

11.10 Exercises

Exercise 11.1 (★ — Seed management)**.** Train a small neural network on MNIST *without* setting any random seed. Run training 5 times and record the variance in test accuracy. Then implement full seed management and repeat. Compare the variance in both cases.

Exercise 11.2 (★★ — Model card creation)**.** Choose a pre-trained model from Hugging Face Hub. Write a comprehensive model card including:

1. Model details and architecture
2. Intended use and out-of-scope uses
3. Performance metrics disaggregated by demographic group (if applicable)
4. Ethical considerations and known biases
5. Reproduction instructions

Exercise 11.3 (★★★ — Full reproducibility audit)**.** Take an existing ML project (your own or an open-source one) and conduct a reproducibility audit:

1. Attempt to reproduce the reported results from scratch
2. Document every issue encountered (missing seeds, undocumented steps, version conflicts)

3. Fix all reproducibility issues
4. Add a model card, datasheet, and NeurIPS-style checklist
5. Run multi-seed experiments and report confidence intervals
6. Package the project with Docker for environment reproducibility

11.11 Cheatsheet

Chapter 11 Summary

Concept	Key Point
Repeatability	Same team, same setup, same results
Reproducibility	Different team, same code/data, same results
Replicability	Different team, different code, same conclusions
Random seeds	Fix Python, NumPy, PyTorch, CUDA, DataLoader
Model card	Standardized doc: use, metrics, limitations, biases
Datasheet	Dataset documentation: composition, collection, uses
Multi-run	Report mean $\pm$ std over ≥ 3 seeds
Determinism	`torch.use_deterministic_algorithms(True)`

Appendix A

Appendix: MLOps Tool Comparison

This appendix provides a comprehensive comparison of tools across the major MLOps categories. Use these tables as a reference when selecting tools for your projects.

A.1 Environment Management

Tool	Language	Locking	System Deps	Speed	GPU
venv + pip	Python	pip-tools	No	Fast	Manual
Conda	Python/R/C	Built-in	Yes	Slow	Yes
Mamba	Python/R/C	Built-in	Yes	Fast	Yes
Poetry	Python	Built-in	No	Medium	Manual
uv	Python	Built-in	No	Very fast	Manual
PDM	Python	Built-in	No	Fast	Manual
Pixi	Any	Built-in	Yes	Fast	Yes

A.2 Version Control

Tool	Type	Open Source	Cloud	Best For
Git	Code	Yes	No	Universal code versioning
DVC	Data + models	Yes	Optional	Data versioning alongside Git
Git LFS	Large files	Yes	GitHub	Binary files in Git repos
LakeFS	Data (S3-like)	Yes	Yes	Data lake versioning
Delta Lake	Data (tables)	Yes	Databricks	Versioned tabular data
Pachyderm	Data pipelines	Yes	Yes	Data-driven pipelines

A.3 Experiment Tracking

Tool	Open Source	UI	Collab	Self-host	Pricing
MLflow	Yes	Yes	Limited	Yes	Free
Weights & Biases	No	Excellent	Yes	Enterprise	Freemium
Neptune	No	Yes	Yes	No	Freemium
CometML	No	Yes	Yes	Enterprise	Freemium
Aim	Yes	Yes	Limited	Yes	Free
ClearML	Yes	Yes	Yes	Yes	Freemium
TensorBoard	Yes	Basic	No	Yes	Free

A.4 Pipeline Orchestration

Tool	DAG Def.	K8s	GPU	UI	Best For
Airflow	Python	Yes	Plugin	Yes	General data pipelines
Prefect	Python	Yes	Yes	Yes	Modern data workflows
Dagster	Python	Yes	Yes	Yes	Data-asset-centric
Kubeflow	YAML/Python	Native	Yes	Yes	ML on Kubernetes
Metaflow	Python	No	Yes	Limited	ML research to prod
ZenML	Python	Yes	Yes	Yes	MLOps framework
DVC Pipeline	YAML	No	No	Limited	Simple ML pipelines

A.5 Containerization and Infrastructure

Tool	Purpose	GPU	Scale	Best For
Docker	Containerization	Yes	Single host	Dev and CI
Docker Compose	Multi-container	Yes	Single host	Local stacks
Kubernetes	Container orchestration	Yes	Cluster	Production
Helm	K8s package manager	–	Cluster	K8s deployments
NVIDIA Container Toolkit	GPU in containers	Yes	–	GPU workloads
Podman	Rootless containers	Yes	Single host	Secure envs

A.6 Model Serving and Deployment

Tool	Perf.	Batching	Multi-model	GPU	Best For
FastAPI	High	Manual	Manual	Manual	Custom APIs
Flask	Medium	No	No	No	Simple APIs
TorchServe	High	Yes	Yes	Yes	PyTorch models
TF Serving	High	Yes	Yes	Yes	TensorFlow models
Triton	Very high	Yes	Yes	Yes	Multi-framework
BentoML	High	Yes	Yes	Yes	ML serving platform
Seldon Core	High	Yes	Yes	Yes	K8s ML serving
KServe	High	Yes	Yes	Yes	Serverless ML on K8s
Ray Serve	High	Yes	Yes	Yes	Scalable serving
vLLM	Very high	Yes	No	Yes	LLM serving

A.7 CI/CD Platforms

Platform	Hosted	Self-host	GPU	ML-specific	Config
GitHub Actions	Yes	Yes	Paid	No	YAML
GitLab CI	Yes	Yes	Yes	No	YAML
Jenkins	No	Yes	Plugin	No	Groovy
CircleCI	Yes	No	Yes	No	YAML
CML (Iterative)	Yes	–	Yes	Yes	YAML
Azure Pipelines	Yes	Yes	Yes	No	YAML

A.8 Monitoring

Tool	Open Source	Drift	Real-time	No-label	Best For
Evidently AI	Yes	Yes	Batch	No	Batch drift reports
WhyLabs	Freemium	Yes	Yes	No	Streaming monitoring
NannyML	Yes	Yes	Batch	Yes (CBPE)	No-label perf. est.
Prometheus	Yes	No	Yes	–	Infra metrics
Grafana	Yes	No	Yes	–	Dashboards
Arize AI	No	Yes	Yes	Yes	Enterprise ML obs.
Fiddler AI	No	Yes	Yes	Yes	Explainable monitoring

A.9 Feature Stores

Tool	Open Source	Online	Offline	Best For
Feast	Yes	Yes	Yes	Open-source feature store
Tecton	No	Yes	Yes	Enterprise real-time
Hopsworks	Yes	Yes	Yes	Full ML platform
Databricks FS	No	Yes	Yes	Databricks users
AWS SageMaker FS	No	Yes	Yes	AWS ecosystem

A.10 Quick Decision Guide

Recommended starter stack

For a team beginning its MLOps journey, a practical starting stack is:

Environment Conda (GPU) or uv (CPU-only)

Version control Git + DVC

Experiment tracking MLflow (self-hosted) or W&B (cloud)

Containerization Docker + Docker Compose

Serving FastAPI (custom) or BentoML (standardized)

CI/CD GitHub Actions + CML

Monitoring Evidently AI + Prometheus + Grafana

Orchestration Add Prefect or Dagster when pipelines grow

Start simple and add complexity only when the team outgrows simpler tools. Premature adoption of Kubernetes or Kubeflow creates more overhead than value for small teams.

Bibliography

[1] C. Huyen. *Designing Machine Learning Systems*. O'Reilly Media, 2022.

[2] M. Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

[3] N. Gift, A. Deza. *Practical MLOps*. O'Reilly Media, 2021.

[4] M. Treveil et al. *Introducing MLOps*. O'Reilly Media, 2020.

[5] MLflow Documentation. https://mlflow.org/docs/latest/index.html, 2024.

[6] DVC Documentation. https://dvc.org/doc, 2024.

[7] Weights & Biases Documentation. https://docs.wandb.ai/, 2024.

[8] Docker Documentation. https://docs.docker.com/, 2024.

[9] FastAPI Documentation. https://fastapi.tiangolo.com/, 2024.

[10] D. Sculley et al. Hidden Technical Debt in Machine Learning Systems. *NeurIPS*, 2015.

[11] A. Paleyes, R. Urma, N. Lawrence. Challenges in Deploying Machine Learning: a Survey of Case Studies. *ACM Computing Surveys*, 2022.

[12] J. Pineau et al. Improving Reproducibility in Machine Learning Research. *JMLR*, 22(164):1–20, 2021.