PHASE 7

MLOps, Deployment & Production Systems

Experiment tracking, serving, containers, monitoring, optimization, and CI/CD for ML

Phase goal

Turn models into reliable products. Track experiments, package and serve models, monitor for drift, optimize for latency/cost, and automate the whole lifecycle. This phase plays directly to your backend and infrastructure strengths.

Estimated time: 4–6 weeks | **Track:** Comprehensive ML & Deep Learning Curriculum

A self-paced, depth-first path from mathematical foundations to production deep learning.

Contents

1 Why MLOps is your edge

A model in a notebook creates zero value. Value comes from a model that serves predictions reliably, cheaply, and correctly — and keeps doing so as data shifts. This is software and systems engineering with a few ML-specific twists, which means your existing strengths (APIs, nginx, OAuth, containers, CI/CD) transfer almost wholesale. This is the phase where you can differentiate fastest from people who can only train models.

Key idea

The "ops" problems unique to ML: (1) the artifact is *data + code + model*, not just code, so all three must be versioned and reproducible; (2) models *decay* silently as the world changes (drift) — there is no exception or stack trace, just quietly worse predictions; (3) training and serving can subtly disagree (training/serving skew). Most of MLOps exists to manage these three realities.

2 Experiment tracking & reproducibility

Before deployment comes the discipline of knowing *what you trained and how*. Log every run's parameters, metrics, code version, data version, and artifacts.

- **MLflow** (open-source, self-hostable) or **Weights & Biases** (hosted, great UI) for tracking and the model registry.
- **Reproducibility hygiene:** pin dependencies, set random seeds, version data (DVC or dataset hashes), and capture the git SHA with each run.

```
import mlflow
with mlflow.start_run():
    mlflow.log_params({"lr": 1e-3, "depth": 6})
    mlflow.log_metric("val_auc", auc)
    mlflow.sklearn.log_model(model, "model")      # versioned artifact
    mlflow.set_tag("git_sha", current_git_sha())
```

Common pitfall

"It worked on my machine / in my notebook" is the enemy. If you cannot reproduce a result from a logged run — same data, same code, same environment — you do not really have that result. Build reproducibility in from the first experiment; retrofitting it after a model is in production is painful.

3 Packaging & serving

3.1 Inference patterns

Pattern	Use when
Batch / offline	predictions can be precomputed (e.g. nightly scores written to a DB)
Real-time / online	low-latency per-request prediction (an API behind your app)
Streaming	continuous event scoring (fraud on a transaction stream)
Edge / on-device	privacy, offline, or latency needs (mobile CoreML/TFLite)

3.2 Serving frameworks

Wrap the model in an API (**FastAPI**/Flask) for full control, or use a dedicated server (**TorchServe**, **Triton**, BentoML, KServe) for batching, versioning, and metrics out of the box. Export to **ONNX** for a framework-agnostic, optimized runtime.

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

app = FastAPI()
model = torch.jit.load("model.pt").eval()        # TorchScript artifact

class Req(BaseModel):
    features: list[float]

@app.post("/predict")
def predict(req: Req):
    x = torch.tensor([req.features])
    with torch.no_grad():
        prob = model(x).softmax(-1)[0].tolist()
    return {"probabilities": prob}

@app.get("/health")
def health():                                     # liveness/readiness probe
    return {"status": "ok"}
```

3.3 Containerization

Docker is the unit of deployment. For ML, mind image size (multi-stage builds), CUDA/-driver compatibility for GPU images, and pinning the exact inference dependencies. The same container runs in CI, staging, and production — eliminating environment skew.

Intuition

The serving stack should feel familiar: it is a stateless service behind a load balancer, with health checks, autoscaling, observability, and a CI/CD pipeline — exactly your backend wheelhouse. The ML-specific additions are the model artifact, its preprocessing, and drift monitoring.

4 Versioning & registries

A **model registry** (MLflow Registry, W&B, SageMaker) is the source of truth: it stores versioned model artifacts with stage labels (staging/production), lineage (which run/data produced it), and enables one-click rollback. Treat model promotion like a deploy: reviewed, gated, and reversible.

5 Monitoring

Once live, a model needs more monitoring than ordinary services, because it can fail *silently*.

- **Operational:** latency, throughput, error rate, resource use (the usual SRE signals).
- **Data drift:** input distribution shifts from training (monitor feature statistics; tests like PSI or KS).
- **Concept drift:** the input→output relationship changes (the world moved; yesterday's patterns no longer hold).
- **Performance:** the real metric (accuracy/AUC) once ground-truth labels arrive — often delayed, so use proxies meanwhile.
- **Prediction monitoring:** output distribution, confidence, and (for LLMs) safety/quality checks.

Common pitfall

The classic production failure is silent degradation: no errors, no alerts, just a model whose accuracy quietly slid because the input data drifted (a new product category, a changed upstream feature, seasonality). Alert on *input drift* as an early warning, since true performance labels often arrive days or weeks late. Also guard against the **training/serving skew** where preprocessing differs between offline training and online serving — a leading cause of "great offline, bad online."

6 Optimization for production

Making models fast and cheap is often as valuable as making them accurate.

- **Quantization:** use int8/fp16 instead of fp32 — big speed/memory wins with small accuracy loss (post-training or quantization-aware).
- **Distillation:** train a small "student" to mimic a large "teacher" — much smaller, nearly as good.
- **Pruning:** remove redundant weights/structures.
- **Graph/runtime optimization:** ONNX Runtime, TensorRT (fuse ops, pick optimal kernels).
- **Batching:** dynamic batching of concurrent requests raises throughput dramatically (tune against your latency SLO).

Intuition

A 4-bit quantized model that fits on one GPU and answers in 50 ms can be worth far more than a 2%-more-accurate model that needs four GPUs and 400 ms. In production, latency, cost, and reliability are first-class metrics — not afterthoughts. This is also where your earlier SVD/low-rank (Phase 0) and LoRA (Phase 4) intuitions pay off.

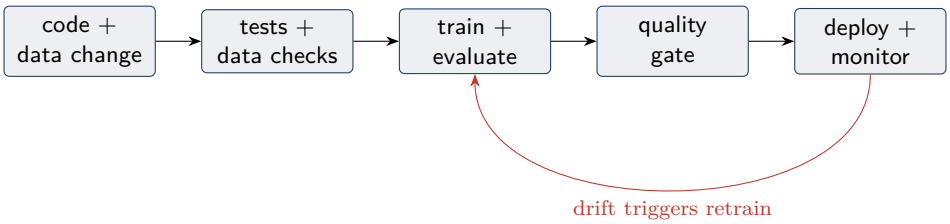
7 Feature stores & pipelines

A **feature store** (e.g. Feast) centralizes feature definitions and serves them consistently to both training (offline) and serving (online) — the standard cure for training/serving skew. You do

not need one for small projects, but understand the concept: "compute a feature once, use it everywhere, consistently."

8 CI/CD for ML

Extend familiar CI/CD with ML stages:



Add: data validation (Great Expectations), automated retraining triggers (on drift or schedule), a quality gate (don't deploy if the candidate underperforms the incumbent on a held-out set), and progressive rollout (canary / shadow / A-B, connecting back to Phase 1).

9 Checkpoint project

Productionize an earlier model end to end

Goal: take any model from Phases 1–4 and build the full production loop.

1. **Track:** retrain it under MLflow/W&B with logged params, metrics, and a registered model artifact.
2. **Serve:** wrap it in a FastAPI service with `/predict` and `/health`; validate inputs with Pydantic.
3. **Containerize:** a slim, pinned Docker image; run it locally and hit it with `curl`.
4. **Monitor:** log requests/predictions; compute a simple input-drift metric and expose Prometheus-style metrics.
5. **Automate:** a CI pipeline (GitHub Actions) that runs tests, builds the image, and (optionally) deploys; gate on a minimum eval score.
6. **Document:** a README with the architecture diagram, API contract, and a runbook (how to roll back, what alerts mean).

Definition of done: one command brings the service up; a load test reports latency/throughput; a simulated data shift trips your drift metric. **Stretch:** add ONNX export + quantization and report the latency improvement; add a canary deploy.

Phase 7

- **Designing Machine Learning Systems** (Chip Huyen) — the practitioner's bible for this phase.
- **Machine Learning Engineering** (Andriy Burkov) — free online; broad and practical.
- **Made With ML** (madewithml.com) — strong, free, code-first MLOps course.
- Google's *Rules of ML* and the *MLOps: Continuous delivery* whitepaper.
- Tools to get hands-on with: MLflow, FastAPI, Docker, ONNX Runtime, Prometheus/-Grafana, GitHub Actions.

10 Phase 7 self-check

- Explain why ML systems version data + code + model, not just code.
- Choose an inference pattern (batch/online/streaming/edge) for a given use case.
- Stand up a containerized prediction API with health checks.

- Distinguish data drift, concept drift, and training/serving skew, and say how you'd detect each.
- Pick an optimization (quantization/distillation/batching) for a latency or cost target.
- Describe a CI/CD pipeline for ML with a quality gate and a retraining trigger.