From Notebook to Production

Deploying Finance ML: Serialization, APIs, Dashboards, RAG, and MLOps

Day 7, Full Day

Where We Are: Day 7 of 7

Six days in, you have built real finance models. Today we make them usable by someone other than you.

- Day 1: cleaned and explored a financial dataset (returns, missing data, outliers)
- Day 2-4: trained and evaluated supervised models (regression, classification, neural networks)
- Day 5: clustered stocks with K-Means, compressed portfolios with PCA, measured text similarity with embeddings
- Day 6: explained transformers and prototyped a retrieval pipeline over earnings calls

Today's question

A trained model living inside a Jupyter notebook serves nobody. How do you turn it into a service a loan officer, a portfolio manager, or an analyst can actually call?

Deployment is the bridge from "it works in my notebook" to "it works for the bank"

The Model That Never Left the Notebook

Most machine learning models die in the notebook they were born in.

- The data scientist trains a model, gets 0.87 accuracy, screenshots the confusion matrix, closes the laptop
- Six months later, nobody can re-run it: the data path changed, a library version moved, the preprocessing steps live only in the author's head
- The business never used it because it was never reachable: there was no button, no endpoint, no app

Finance analogy

A credit-scoring model that only runs on one analyst's machine is like a lending policy that lives in one person's memory. It cannot be audited, cannot be applied consistently, and disappears when that person leaves.

The goal of deployment: make the model reachable, reproducible, and auditable

Training-Serving Skew: The Silent Killer

The model in production sees data that is shaped differently from the data it trained on.

- In training: a clean pandas DataFrame with 500 rows, all five features present, scaled with a fitted `StandardScaler`
- In production: one applicant arrives at a time, as a JSON object, with no scaler attached, possibly with a missing field or a value outside the training range
- If the serving code does not apply the *exact same* preprocessing the model was trained with, the predictions are quietly wrong, never crashing, just degrading

The rule

Whatever transformations happened before `model.fit()` must happen, identically, before `model.predict()` in production. The model and its preprocessing must travel together.

Training-serving skew never throws an error; it just makes the model lie

What “Deployment” Actually Means

Deployment is the set of choices that turn a trained model object into something the rest of the world can use.

- **Serialize** the trained model so it survives the notebook closing (Section 2)
- **Wrap** it in an API so software can call it: send features, get a prediction (Section 3)
- **Wrap** it in a dashboard so a human can interact with it: drag a slider, see a chart (Section 4)
- **Ground** a language model in your documents so it answers from real text, not hallucination (Section 5)
- **Package and ship** it to a server where it runs without your laptop (Section 6)
- **Monitor** it so you know when it stops working (Section 7)

Six layers, each one small; together they are the difference between a demo and a product

The Problem: Re-Training Is Wasteful and Non-Reproducible

A trained model is a Python object holding millions of fitted numbers. When the process exits, that object is gone.

- Re-training the credit-scoring MLP every time the API restarts wastes minutes and burns compute
- Worse: re-training is not reproducible unless every random seed, library version, and data row is pinned. The model you serve at noon may not be the model you served at 9am
- You want to train *once*, freeze the result, and load that exact frozen object whenever you need a prediction

The solution: serialization

Serialization writes the trained model object to a file on disk in a format that can be read back later, on a different machine, into the same object. "Pickling" (Python's native term) and `joblib` are the two common tools.

Train once, serialize, then load the frozen model forever after

joblib: Freeze and Thaw

`joblib.dump` **writes the model to a file;** `joblib.load` **reads it back into the same object.**

- `joblib.dump(model, "credit_model.joblib")`: the fitted `MLPClassifier`, with all its weights, is now a file on disk
- `model = joblib.load("credit_model.joblib")`: in a fresh process, on a server, the model is back, ready to `predict` without any re-training
- `joblib` is preferred over plain `pickle` for scikit-learn models because it handles large NumPy arrays efficiently
- The round-trip is exact: `loaded_model.predict(X)` gives the identical output to the original `model.predict(X)`

Example

You train the Day 5 credit-scoring MLP, call `joblib.dump`, and commit the `.joblib` file alongside your code. The API server loads it on startup and answers `/predict` calls. The model never re-trains.

joblib.dump = freeze; joblib.load = thaw; the model survives across machines and time

The Trust Caveat: Loading a Model Runs Code

Deserializing a pickled object can execute arbitrary code. Only load model files you trust.

- The pickle format is not just data: it can contain instructions that run when the file is loaded
- A malicious `.joblib` or `.pkl` file downloaded from an untrusted source can run anything on your machine the moment you call `joblib.load`
- In a bank, this matters: a model artifact is part of your supply chain. It should come from your own training pipeline, stored in your own artifact registry, with a known hash
- Safer alternatives exist for pure-data serialization (ONNX, safetensors), but for scikit-learn models in this course, `joblib` from a trusted source is the standard

The rule

Treat a model file like executable code, because it is. Load only artifacts your own pipeline produced.

A model file is code in disguise: trust the source the way you trust a binary

What Must Travel With the Model

Serializing the model alone is not enough. The model is useless without the things that make its inputs valid.

- **The fitted preprocessing**: the `StandardScaler` (or pipeline) that was fitted on the training set. Serialize it too, or wrap model and scaler in a single scikit-learn `Pipeline` and serialize that
- **The feature list and order**: the API must build the input array in exactly the order the model expects (`income`, `debt_ratio`, `credit_score`, `employment_years`, `loan_amount`)
- **The version metadata**: which training run, which data snapshot, which library versions. When predictions look wrong six months later, this is what you check first

Finance analogy

Serializing a model is like archiving a signed risk-model document: the model, the data it was validated on, and the sign-off all go in the folder together. A model without its provenance is not auditable.

Ship the model, its preprocessing, its feature schema, and its provenance as one unit

What Is an API?

An API is a contract: “send me a request shaped like this, and I will send you a response shaped like that.”

- “API” stands for Application Programming Interface: the agreed way one piece of software talks to another
- A *web* API specifically: you send an HTTP request over the network, you get an HTTP response back
- The contract is fixed and documented: the caller does not need to know how the model works, only what to send and what comes back
- Once a model lives behind an API, *anything* can use it: a website, a mobile app, an Excel macro, another model

Finance analogy

An API is like a loan-officer terminal: you type in the applicant’s income, debt ratio, and credit score, press a button, and a decision comes back. You do not need to know the model inside; you need to know which fields to fill in.

An API hides the model behind a fixed input/output contract

HTTP Basics: GET, POST, and JSON

Two request types cover almost everything; the data travels as JSON.

- **GET**: "give me something." No body, just a URL. Used for reading. A health-check endpoint (`GET /health` returns `ok`) is a typical use
- **POST**: "here is some data, do something with it." Carries a body. A prediction request (`POST /predict` with the applicant's features) is a POST
- **JSON**: JavaScript Object Notation, a plain-text format of key-value pairs: `{"income": 60000, "credit_score": 700}`. It is the universal language of web APIs because every programming language can read and write it
- The response is also JSON: `{"approved": true, "probability": 0.83}`

Example

A risk dashboard sends `POST /predict` with a JSON body of the five credit features; the API responds with a JSON body containing the approval probability. Both sides agreed on the field names in advance.

GET to read, POST to send data; JSON is the wire format both ends understand

FastAPI: A Model Endpoint in a Few Lines

FastAPI is a Python framework for building web APIs. You declare an endpoint as a function, and it handles the HTTP plumbing.

- You write an ordinary Python function and decorate it: "this function answers `POST /predict`"
- Inside the function: load the JSON, build the feature array in the right order, call `model.predict_proba`, return a JSON dictionary
- FastAPI automatically generates interactive documentation at `/docs`, so a colleague can see exactly what to send without reading your code
- It is fast, it is widely used in industry, and it works with any model object you can call from Python

The pattern (in words, not code)

1. On startup, `joblib.load` the model once. 2. Define a `/predict` endpoint that accepts the five features. 3. Inside, assemble the input, predict, return the probability. 4. Run it with a server (`uvicorn`). The model is now a service.

FastAPI turns a Python function into a callable web endpoint, with docs for free

Input Validation: Reject Bad Data Before the Model Sees It

The model trusts its inputs. Production data does not deserve that trust. Validate at the door.

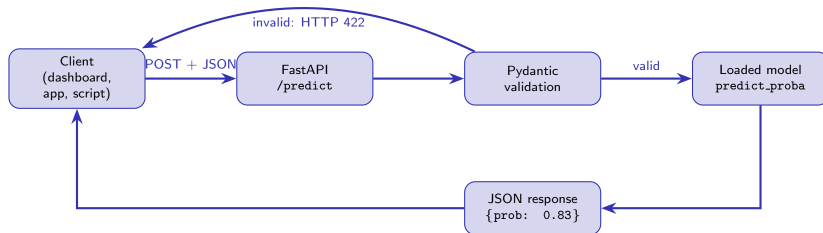
- A request arrives with `income: -5000` or `credit_score: 1200`: values impossible in the training data. The model will not crash; it will return a confident, meaningless number
- FastAPI uses *Pydantic* models to declare the expected shape and constraints: income must be a non-negative number, credit score must be between 300 and 850, all five fields must be present
- If the request violates the contract, FastAPI rejects it with HTTP 422 ("Unprocessable Entity") and a clear message, *before* the model is ever called
- This is the API's job: be the bouncer that keeps the model's inputs inside the range it was trained on

Finance analogy

Input validation is the form-checking step at a loan desk: before the application reaches the underwriter, the front office confirms the income field is filled, the SSN is the right length, the requested amount is plausible. Garbage gets bounced at intake.

Validate inputs at the API boundary; never let out-of-range data reach the model

The Request Flow, End to End



One request: the client sends JSON, the API validates it, the model predicts, JSON comes back. Bad input never reaches the model.

- The model is loaded once when the server starts, not per request: that is the difference between a fast API and a slow one
- Everything between the client and the model is glue: validation, array assembly, response formatting

Load the model once; validate every request; the model only sees clean input

What You Build in Colab A

Notebook: "FastAPI credit-scoring service" wraps the Day 5 MLP in a callable endpoint.

- Re-create the Day 5 credit-scoring data and model, then `joblib.dump` it
- Build a tiny demo API first (`POST /add` returns `a + b`) so the HTTP mechanics are clear before the real model
- Add a `/predict` endpoint with Pydantic validation on the five features
- Test it from Python: one valid request returns a probability (HTTP 200), one negative-income request returns HTTP 422
- See the auto-generated docs at `/docs`: the contract, written for you

Checkpoint you will hit

Before building `/predict`: "what shape is the request JSON?" and "what HTTP status does a negative-income request return?" Predict, then test.

By the end of Colab A you have a credit model you can call over HTTP

The Problem: APIs Serve Machines, Humans Want to Click

A `/predict` endpoint is perfect for software. A portfolio manager wants a screen with sliders and charts.

- Not every stakeholder writes code or sends HTTP requests. A risk officer wants to drag a control and watch the output change
- Building a full web front end (HTML, JavaScript, a server) is a project in itself: weeks of work, a different skill set
- Streamlit collapses that: you write a Python script, and it becomes an interactive web app. Every `st.slider` is a control; every `st.pyplot` is a live chart; every `st.dataframe` is a sortable table
- The same model, the same data, now reachable by a human through a browser

Finance analogy

A Streamlit dashboard for the Day 5 stock clusters is like a portfolio-manager terminal: drag the “number of regimes” slider, and the groupings of the 30 stocks re-draw in front of you. No code, no API call, just a control and a picture.

APIs are for software; dashboards are for people who need to see and steer

Streamlit: A Script Becomes an App

You write a top-to-bottom Python script. Streamlit re-runs it whenever a control changes and renders the output as a web page.

- `st.title("Stock Clusters")`: a heading on the page
- `k = st.slider("Number of clusters", 2, 8, 3)`: a slider; `k` holds whatever the user dragged it to
- `fig = make_cluster_plot(data, k); st.pyplot(fig)`: a matplotlib figure, drawn live, re-computed when `k` changes
- There is no front-end code, no callbacks, no server setup you write: Streamlit handles all of it. The mental model is "a script that re-runs on every interaction"

Example

Your dashboard script: load the 30-stock returns, show a slider for `k` and a dropdown for ticker, run K-Means with the chosen `k`, plot the clusters and the PCA biplot. A risk officer opens the URL and explores regimes by dragging.

Streamlit turns a linear Python script into a clickable web app

Caching: Do Not Re-Compute What Has Not Changed

Streamlit re-runs the whole script on every interaction. Without caching, that means re-loading data and re-fitting models on every click.

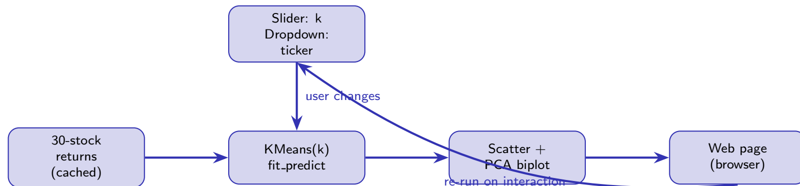
- The user drags the ticker dropdown: the entire script re-runs, including `pd.read_csv` and `KMeans().fit` on data that has not changed. The app feels sluggish
- `@st.cache_data` on the data-loading function: "run this once, remember the result, re-use it on every re-run unless the inputs change"
- `@st.cache_resource` for expensive objects like a loaded model: load it once, share it across interactions
- With caching, only the cheap part (re-drawing the chart for the new `k`) actually re-runs. The app feels instant

The rule

Wrap every expensive step (loading data, fitting a model, downloading a file) in a cache decorator. The first run pays the cost; every later run is free.

Cache the expensive steps; let only the cheap re-draw happen on each interaction

The Dashboard Flow



User drags the slider, KMeans re-runs with the new `k`, the plot re-draws, the page updates. The cached data load does not repeat.

- What you build in Colab B: this exact dashboard as an `app.py` file, plus the launch command, plus a deploy step
- Checkpoint: "when the user drags `k` from 3 to 6, what changes in the scatter plot?" Predict, then run

The interaction loop: control changes, script re-runs, only the uncached parts recompute

The Problem: A Language Model Alone Hallucinates

Ask a general language model "what was Acme Corp's Q3 revenue growth?" and it will produce a fluent, confident, possibly invented number.

- A language model knows patterns of language, not the contents of *your* earnings calls. It was trained on the public internet up to some cutoff, not on your firm's transcripts
- It does not say "I do not know"; it generates the most plausible-sounding continuation, which can be wrong
- In finance this is disqualifying: an analyst cannot act on a number the model made up. The answer must be traceable to a real source
- The fix is not a bigger model. The fix is to *give the model the relevant text* before it answers

Finance analogy

A bare language model is like an analyst answering from memory under time pressure: fluent, but unverifiable. A RAG system is the same analyst who, before answering, pulls the exact transcript passages and reads from them.

A language model alone is fluent and unmoored; grounding it in real text is the cure

RAG: Retrieval-Augmented Generation

RAG = Retrieval-Augmented Generation. Retrieve the relevant text, stuff it into the prompt, then generate an answer grounded in that text.

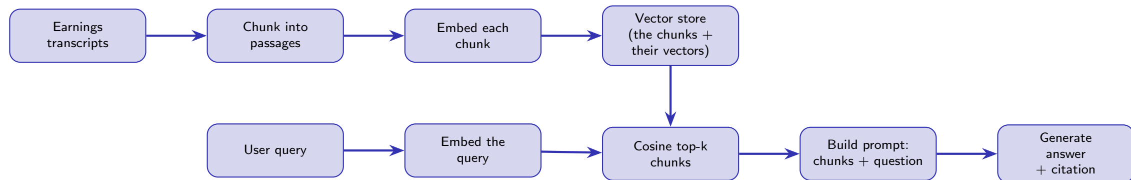
- **Retrieval** is the half you can trust: embed the documents, embed the question, find the document chunks closest in meaning. This is exactly the cosine-similarity skill from Day 5
- **Generation** is the half that needs grounding: a language model writes the final answer, but only from the retrieved chunks placed in its prompt, not from its training memory
- The prompt the model sees becomes: "Here are three passages from the earnings call. Using only these, answer: what was revenue growth?"
- The model can still phrase the answer naturally, but it is now anchored to text you can show the analyst

Example

Query: "revenue growth." Retrieval pulls the three transcript chunks mentioning revenue. The prompt = those chunks + the question. The model answers "revenue grew 12 percent year over year," citing the chunk it came from.

RAG = trusted retrieval (Day 5 embeddings) plus grounded generation (a small language model)

The RAG Pipeline



Documents are chunked and embedded once (offline). At query time: embed the question, retrieve the closest chunks, build a grounded prompt, generate.

- Steps 1-4 (chunk, embed, store) happen once when the documents arrive. Steps 5-8 (embed query, retrieve, prompt, generate) happen on every question

Index the documents once; embed-retrieve-prompt-generate on every query

Why Top-k, Not All Chunks

Tempting shortcut: stuff every document chunk into the prompt and let the model sort it out. This breaks two ways.

- **The context window is finite**: a language model can only read so much text at once. Twenty earnings calls do not fit. The prompt gets truncated, often dropping the relevant part
- **Relevance gets diluted**: even if it fit, burying the three relevant passages among two hundred irrelevant ones makes the model's job harder, not easier. Signal-to-noise collapses
- **Top-k is the fix**: retrieve only the `k` chunks closest to the query ($\mathtt{k} = 3$ to 5 is typical). The prompt stays small, every chunk in it is on-topic, the model has a clean task
- Choosing `k` is a real tradeoff: too small misses context, too large dilutes. Start at 3, tune by inspection

The rule

Never put all your documents in the prompt. Retrieve the top few most-similar chunks and put only those in. The retrieval step exists precisely to make this selection.

Top-k retrieval keeps the prompt small and on-topic; "all chunks" overflows and dilutes

Citations: The Answer Must Point to Its Source

A grounded answer with no citation is only half-grounded. The user needs to verify.

- When the model says "revenue grew 12 percent," the system should also surface *which retrieved chunk* that came from: the transcript passage, ideally with a speaker and timestamp
- This is what makes RAG usable in finance: the analyst clicks through to the source and confirms before acting. "Trust, but verify" becomes mechanical
- Practically: the retrieval step already knows which chunk it returned; carry that identifier through the prompt and into the displayed answer
- A RAG system that answers without citing is back to the hallucination problem, just with extra steps

Finance analogy

An equity research note that cites the exact page of the 10-K is trusted; the same claim with no footnote is an opinion. RAG without citations is an opinion engine.

Always carry the source chunk's identity into the answer; an uncited RAG answer is just a confident guess

What You Build in Colab C

Notebook: "RAG earnings chatbot" builds the full pipeline over the Day 5 earnings snippets and deploys it.

- Re-use the Day 5 earnings-call snippets and the same small embedding model (`paraphrase-MiniLM-L3-v2`, local, no API key)
- Build a tiny demo first: retrieval over three toy sentences, so "embed, find the closest" is concrete before the real pipeline
- Full pipeline: chunk, embed, retrieve top-k, build the grounded prompt, generate the answer with a small local model (`flan-t5-small`, runs on a free CPU)
- Wrong-way cell: retrieve *all* chunks and watch the prompt overflow, then fix it with top-k
- Deploy step: write an `app.py` and a `requirements.txt`, drag them into a Hugging Face Space, get a live URL

Checkpoint you will hit

"For the query revenue growth, which snippets get retrieved, and what does the final prompt look like?" Predict, then run.

By the end of Colab C you have a grounded earnings chatbot, deployed, with citations

The “Works on My Machine” Problem

Your API runs fine on your laptop. You hand it to a colleague; it breaks. Different Python version, a missing library, a different operating system.

- Software depends on its environment: the exact Python version, the exact package versions, sometimes system libraries. Your laptop has one combination; the server has another
- “It works on my machine” is the oldest excuse in software because the machine *is* part of the program, even though we pretend it is not
- For a finance model this is a control problem: the model you validated must be byte-for-byte the model in production, including its runtime
- The fix: package the code *together with* its environment, so wherever it runs, it runs the same

The solution: containers

A container bundles your code, your dependencies, and a minimal operating system into one image. Run that image anywhere, and you get exactly the environment you built. The receiving server does not need to recreate your setup.

Containers make “works on my machine” true everywhere by shipping the machine with the code

Docker: The Recipe and the Image

A `Dockerfile` is a recipe: "start from this base, copy in this code, install these packages, run this command." Building it produces an image you can run anywhere.

- **Base image**: start from `python:3.11-slim`, a minimal Linux with Python already installed. You do not build the world from scratch
- **Copy code**: copy your `app.py`, your model file, your `requirements.txt` into the image
- **Install dependencies**: run `pip install -r requirements.txt` inside the image, freezing exact versions
- **Run command**: declare how to start the app (`uvicorn app:app`). When someone runs the image, this is what executes
- `docker build` turns the recipe into the image; `docker run` starts a container from it. The same image runs on your laptop, a teammate's, or a cloud server, identically

Example

You write a six-line `Dockerfile` for the credit-scoring API, `docker build` it, and now the bank's platform team can run that exact image in their cluster without asking you what Python version you used.

Dockerfile = recipe; `docker build` **= bake the image;** `docker run` **= serve it, identically, anywhere**

Free-Tier Hosts: Where to Actually Put It

You do not need a credit card or an AWS account to put a model online. Several services have generous free tiers a student can use in class.

- **Hugging Face Spaces**: drag-and-drop an `app.py` and a `requirements.txt` in the browser, get a live URL in minutes. No GitHub repo required. Best for dashboards and chatbots. This is the path used in the notebooks
- **Streamlit Community Cloud**: connects to a *public* GitHub repo and deploys the Streamlit app on push. Free, but your code must be in a public repo, which your course models may not be
- **Render / Railway**: free-tier web services, good for FastAPI endpoints; the app sleeps when idle and wakes on the next request
- All of these are demo-grade: rate-limited, sleepy when idle, not for real production traffic. But they are exactly right for "ship something a beginner built, today"

The matching rule

Dashboard or chatbot → Hugging Face Spaces (drag-and-drop). FastAPI endpoint → Render or Railway. Public-repo project → Streamlit Community Cloud. Pick the host that fits the workload and the repo situation.

Free tiers are demo-grade but real; HF Spaces drag-and-drop is the no-friction default

From Notebook to Live URL: The Shortest Path

You do not have to learn Docker to ship a class project. The shortest path skips the container and uses a managed host.

- Write your app as a single `app.py` (Streamlit dashboard, or a small RAG chatbot interface)
- List its dependencies in a `requirements.txt` (`streamlit`, `scikit-learn`, `sentence-transformers`, `transformers`)
- Create a new Hugging Face Space, choose "Streamlit" or "Gradio", drag the two files in
- The Space builds the environment for you and gives you a public URL. Share it
- Docker is the next step up: when you outgrow the managed host, the `Dockerfile` lets you run the same app on any infrastructure. But it is not the entry ticket

In Colab B and C

Each notebook writes the `app.py` and `requirements.txt` to disk and walks through the Hugging Face Spaces upload. The in-Colab preview (via a tunnel) is shown but treated as throwaway; the live URL is the real deliverable.

`app.py` + `requirements.txt` + **a Hugging Face Space = a live URL, no Docker required**

Deployment Is Not the End: Models Decay

The day you deploy a model is the day it starts getting worse, because the world it learned stops standing still.

- A credit-scoring model trained on 2015-2019 applicants performs beautifully in 2019. In 2021, after a pandemic reshaped employment and lending, the same model is quietly mis-ranking applicants
- Nothing crashed. The API still returns probabilities. They are just less accurate, and you will not know unless you are watching
- "MLOps" is the practice of treating a deployed model like a living system that needs monitoring, alerting, and periodic re-validation, not a finished artifact
- In a bank this is mandatory, not optional: model risk management requires ongoing performance tracking and a documented re-validation schedule

The mindset shift

"Trained and deployed" is the start of the model's working life, not the end of the project. The maintenance phase is longer than the build phase.

A deployed model is a perishable asset; MLOps is the practice of watching it age

Two Kinds of Drift

Models go stale in two distinct ways. Telling them apart tells you what to do.

- **Data drift**: the *inputs* change distribution. The average applicant income shifts, a new product brings in a different customer mix, a region's data starts flowing. The model is fine; it is just seeing a population it was not trained on. Detect by tracking input feature distributions over time
- **Model drift (concept drift)**: the *relationship* between inputs and outcome changes. The same income and credit score that meant "low risk" in 2019 mean something different post-COVID. The model's learned mapping is now wrong. Detect by tracking prediction accuracy against actual outcomes as they arrive
- Data drift is a warning sign; model drift is a failure. Both eventually require re-training, but model drift requires it *urgently*

Example

Your stock-clustering dashboard from Day 5: if a new asset class enters the universe, that is data drift. If a regime change makes "high volatility" no longer cluster the way it used to, that is concept drift in the structure itself.

Data drift: the inputs moved. Concept drift: the rules moved. The second is the emergency

Monitoring: What to Watch on a Live Model

You cannot fix what you cannot see. A deployed model needs a dashboard about itself.

- **Input distributions**: are the feature values arriving in the ranges the model expects? A sudden spike in out-of-range requests means something upstream changed
- **Prediction distributions**: is the model suddenly approving 95 percent of applicants when it used to approve 60 percent? That shift is a red flag even before you have ground truth
- **Latency and error rates**: is the API responding in milliseconds, or is it timing out? Are requests returning 422 (bad input) or 500 (something broke)?
- **Accuracy against actuals**: when the real outcome arrives (the loan defaulted or did not), compare it to what the model predicted. This is the gold standard, but it is delayed

The rule

Log every prediction with its inputs and timestamp. Then you can compute all of the above retrospectively, and reconstruct what the model was doing on any past day. Without logging, you are blind.

Log inputs, predictions, latency, and outcomes; the model needs a monitor as much as a database does

Retraining, Versioning, and CI/CD

When monitoring says the model has drifted, you re-train. Doing that safely needs a process, not a heroic Tuesday-night re-run.

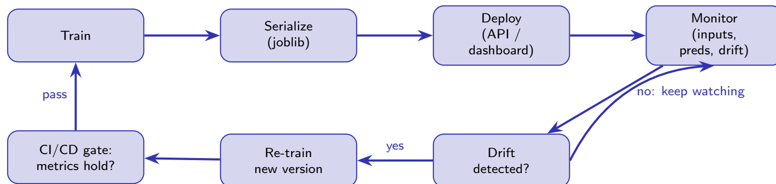
- **Retraining triggers**: scheduled (every quarter, regardless), or threshold-based (when a drift metric crosses a line). A bank typically does both: a calendar re-validation plus an alert-driven one
- **Versioning**: every model is a numbered, dated artifact (`credit_model_v7_2026Q2.joblib`), stored in a registry, with the training data snapshot and metrics recorded. You can always say which model was live on which day
- **CI/CD for ML**: "Continuous Integration / Continuous Deployment." Before a new model version goes live, an automated pipeline runs it against a held-out test set, checks the metrics did not regress, and only then promotes it. Same idea as running tests before merging code
- Roll-back must be one command: if `v8` misbehaves in production, you revert to `v7` immediately

Finance analogy

Promoting a new model version through a CI/CD gate is like a model passing internal validation before the risk committee signs off. No model reaches production without passing the gate, and any model can be pulled.

Trigger → re-train → test the new version → promote if metrics hold → keep the rollback handy

The Full Lifecycle, One Picture



Train, serialize, deploy, monitor. When drift appears, re-train, pass the gate, and the new version takes over. The loop never ends.

- Everything you learned this week sits inside this loop: the model from Days 2-6, serialized today, served today, and from now on, watched

The MLOps loop: build → serve → watch → rebuild; deployment is one station, not the terminus

One Thing Per Day

You walked in not knowing Python. Here is what you walk out able to do, one capability per day of this course.

1. **Day 1**: Clean and explore a financial dataset: load price data, compute return series, detect outliers, handle missing values
2. **Day 2**: Diagnose a model: read a confusion matrix, interpret precision and recall, reason about bias and variance, split data honestly
3. **Day 3**: Train and evaluate a classifier on finance data: fit it, measure it, compare it to a baseline
4. **Day 4**: Build a neural network: stack layers, train with backpropagation, recognize and curb overfitting
5. **Day 5**: Find structure without labels: cluster assets with K-Means, compress portfolios with PCA, measure document similarity with sentence embeddings
6. **Day 6**: Explain how a transformer turns text into meaning, and prototype a retrieval pipeline over real documents
7. **Day 7**: Ship it: serialize a model, serve it behind a FastAPI endpoint, build a Streamlit dashboard, deploy a grounded RAG chatbot, and reason about monitoring and drift

---

Seven days, seven capabilities; together they are an end-to-end finance ML toolkit

Where to Go Next

This course was the on-ramp. A few honest directions if you want to keep going.

- **Deepen one model class**: pick gradient boosting (XGBoost) or deep learning and go deep. Breadth got you here; depth is the next move
- **Learn the data layer**: SQL, time-series databases, feature stores. In practice, getting clean data is most of the job
- **Real MLOps tooling**: experiment tracking (MLflow, Weights and Biases), pipeline orchestration, model registries. The concepts from Section 7, made concrete
- **Finance specifics**: backtesting frameworks, risk model validation standards, the regulatory context (model risk management guidance). The domain has its own rigor
- **Build a portfolio piece**: take one of this week's notebooks, extend it, deploy it publicly, write it up. A live URL and a clear README is worth more than a certificate

Closing

You can train a model, serve it, and explain when it will fail. That is the whole pipeline. Everything else is depth.

Breadth was the goal of this week; depth, data, and a deployed portfolio piece are the next steps

Thank you.

From a financial dataset on Day 1 to a deployed, monitored finance ML service on Day 7.