From Notebook to Production

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

Day 7 Post-Course Handout

Data Science with Python: BSc Intensive (Finance Focus)

Joerg Osterrieder

May 15, 2026

This handout is the written companion to Day 7 of the course. Days 1 to 6 taught you to clean data, train models, build a neural network, cluster assets, and prototype a retrieval pipeline over text. Day 7 is the bridge from "it works in my notebook" to "it works for the bank." Each section opens with the problem it solves, explains the idea in plain English, gives a finance analogy, and points back at one of the day's worked notebooks: the FastAPI credit-scoring service, the Streamlit stock-cluster dashboard, and the RAG earnings chatbot. Short code fragments are included where they make a concept concrete; the full runnable versions live in the Day 7 Colab notebooks. Read it before the day for orientation, and after the day to consolidate.

Contents

1 The Notebook Graveyard: Why Deploy at All?

The Model That Never Left the Notebook

You spent two days building a credit-scoring model. It reaches 0.87 accuracy on a held-out set. You screenshot the confusion matrix, paste it into a slide, and close the laptop. Six months later a colleague asks to run it. The data path moved, a library version changed, the preprocessing steps live only in your head, and the only "interface" to the model is a Jupyter cell that no longer executes. The business never used the model, because the business could never reach it. There was no button, no endpoint, no app: just a dead notebook in a folder.

Think First

Before reading on: name three things, other than the trained model object itself, that a colleague would need in order to get the same prediction you got six months ago. (Hint: think about what happened to the raw input between loading the data and calling `model.fit`.)

1.1 What "deployment" actually means

Most machine learning models die in the notebook they were born in. Training a model is the visible, satisfying part: you watch the loss go down, you read off a metric, you feel done. But a trained model object, sitting in the memory of a Python process, is useless the moment that process exits. Deployment is the set of unglamorous choices that turn that 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 other software can call it: send features, get a prediction back (Section 3).
- **Wrap** it in a dashboard so a human can interact with it: drag a slider, see a chart update (Section 4).
- **Ground** a language model in your own documents so it answers from real text instead of hallucinating (Section 5).
- **Package and ship** it to a server where it runs without your laptop (Section 6).
- **Monitor** it so you find out when it stops working (Section 7).

Six layers, each one small. Together they are the difference between a demo and a product.

Deployment: Everything that happens between "a trained model object exists in my notebook" and "a person or a piece of software, somewhere else, can get a prediction from it reliably." Deployment is not one tool; it is a chain of choices about storage, interface, packaging, and monitoring.

Common Misconception

"My model reached 0.90 accuracy, so it is done." Accuracy on a held-out set is a lab measurement, not a deployment. A model that cannot be called from another program, that depends on a Python environment only you control, and that disappears when you close the notebook is not deployed: it is a prototype. Deployment begins where training ends. The metric is a starting condition, not a finish line.

1.2 Training-serving skew: the silent killer

There is one failure mode that deserves its own warning, because it never throws an error: it just makes the model quietly wrong. It is called *training-serving skew*.

In training, your model sees a clean pandas DataFrame: 500 rows, all five features present, scaled by a `StandardScaler` that was fitted on exactly that data. In production, the world looks different. One applicant arrives at a time, as a JSON object. There is no scaler attached. A field might be missing. A value might be outside the range anything in the training data ever took. If the serving code does not apply the *exact same* preprocessing the model trained with, the model still returns a number. It is just the wrong number, returned confidently, every time.

Training-serving skew: Any difference between the data pipeline used to train a model and the data pipeline used to serve it in production. The classic case: a feature was scaled or encoded during training but the serving code feeds raw values. The model does not crash; its accuracy silently degrades.

Key Idea

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

Finance Analogy

A credit-scoring model that runs only on one analyst's machine is like a lending policy that lives in one person's memory. It cannot be audited, it cannot be applied consistently across branches, and it disappears the day that person leaves. Deployment is what makes a model an institutional asset rather than a personal one: reachable, reproducible, and reviewable.

Worked Example: What Training-Serving Skew Looks Like

A credit-scoring MLP was trained on five features scaled with `StandardScaler`: income was transformed from roughly \$50,000 to approximately -0.1 (one standard deviation below the training mean). At serving time, the preprocessing step was omitted. The raw value 50 000 was fed directly to the model. The model returned a default probability of 0.91 (high risk) instead of 0.08 (low risk), a factor-of-eleven error with no warning and no exception. The only signal was that every applicant in that batch was flagged high-risk. These are illustrative estimates; the exact flip depends on the model's fitted weights.

Section takeaway. A model in a notebook serves nobody. Deployment makes it reachable, reproducible, and auditable; the first thing it must preserve is the exact preprocessing the model was trained with.

2 Serialization: Saving a Trained Model

Re-Training Is Wasteful and Not Reproducible

Your credit-scoring API restarts (a deploy, a crash, a routine reboot). If the only way to get the model back is to re-train it, every restart burns minutes of compute and, worse, may not reproduce the same model: unless every random seed, library version, and data row is pinned, the model you serve at noon is not byte-for-byte the model you served at 9 a.m. You want to train *once*, freeze the result, and load that exact frozen object whenever a prediction is needed.

Think First

A fitted `MLPClassifier` is, internally, a collection of NumPy arrays of weights plus some bookkeeping. If you could write those arrays to a file and read them back into an identical object on another machine, you would never need to re-train. What could go wrong with "just save the object to a file"? (Hint: who decides what code runs when the file is read back?)

2.1 joblib: freeze and thaw

A trained model is a Python object holding many fitted numbers. When the process exits, the object is gone. *Serialization* writes that object to a file on disk in a format that can be read back later, on a different machine, into the same object. Python's built-in mechanism is called *pickling*; for scikit-learn models the preferred tool is `joblib`, which handles large NumPy arrays efficiently.

Code

```
import joblib

# After training your Day-5 credit-scoring MLP:
joblib.dump(clf, "credit_model.joblib")        # freeze: model is
    now a file

# Later, in a fresh process (an API server, a different machine):
clf = joblib.load("credit_model.joblib")        # thaw: model is
    back, no re-training
clf.predict_proba(X_one_applicant)              # identical output
     to the original
```

Serialization (pickling): Converting an in-memory object into a stream of bytes that can be stored in a file and later reconstructed into an equivalent object. `joblib.dump` writes the file; `joblib.load` reads it back. The round-trip is exact: the loaded model gives the same predictions as the original.

Worked Example

You train the Day-5 credit-scoring MLP on the five features (income, debt ratio, credit score, employment years, loan amount). `joblib.dump(clf, "credit_model.joblib")` writes a roughly few-hundred-kilobyte file. You commit that file alongside your code. When the API server boots, it calls `joblib.load` once, and from then on every `/predict` request is answered by the loaded model with no re-training. The model that handled request #1 is the same model that handles request #10,000.

Worked Example: What the Serialized File Actually Looks Like

A trained `MLPClassifier` on five features serializes to roughly 8 KB raw (about 2 800 floating-point weights plus bookkeeping). `gzip` compresses it to roughly 3 KB. The first byte of the file is `0x80`, the pickle protocol marker; the next four bytes encode the protocol version. The file is not a CSV of weights: it is a Python-specific binary stream that can recreate the full object, including class hierarchy and callable methods. Opening it in a text editor shows garbage; opening it with `joblib.load` gives back the model with identical `predict_proba` output to nine decimal places.

2.2 The trust caveat: loading a model runs code

Here is the "what could go wrong" from the discovery question. The pickle format is not just data. It can contain instructions that execute when the file is loaded. A malicious `.joblib` or `.pkl` file, downloaded from an untrusted source, can run arbitrary code on your machine the moment you call `joblib.load` on it.

Common Misconception

"A `.joblib` file is just numbers, so loading one is safe." It is not. Deserializing a pickled object can execute code. Treat a model artifact the way you treat an executable binary: only run one whose source you trust. In a bank, a model file is part of your software supply chain; it should come from your own training pipeline, be stored in your own artifact registry, and be identifiable by a known hash. There are formats designed for pure-data serialization (ONNX, safetensors), but for scikit-learn models in this course, `joblib` from a trusted source is the standard.

2.3 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. Three things must be archived together with it:

1. **The fitted preprocessing.** The `StandardScaler` (or the whole pipeline) that was fitted on the training set. Serialize it too, or, cleaner, wrap model and scaler in a single scikit-learn `Pipeline` and serialize that one object.
2. **The feature schema: names and order.** The serving code must build the input array in exactly the order the model expects. "Income, debt ratio, credit score, employment years, loan amount" is not a comment; it is a contract. Swap two columns and the model silently mis-scores everyone.
3. **The provenance metadata.** Which training run produced this artifact, which data snapshot it used, which library versions. When predictions look wrong six months later, this is the first thing you check.

Finance Analogy

Serializing a model is like archiving a signed risk-model document. You do not file the model in isolation; you file it together with the data it was validated on and the sign-off that approved it. A model without its provenance is not auditable, and an unauditable model is, for a regulated institution, the same as no model.

Alternatives: When Not to Use joblib

- **joblib** (this course): fast, handles large NumPy arrays, the scikit-learn default. Use when you trust the source and stay within Python.
- **ONNX** (Open Neural Network Exchange): cross-language format. Export a scikit-learn or PyTorch model to ONNX and call it from C++, Java, or a browser. Correct choice when the serving environment is not Python.
- **safetensors** (Hugging Face): memory-mapped, no arbitrary code execution on load. Designed for neural-network weight tensors; cannot represent a full scikit-learn pipeline, but is the safe-by-default format for transformer models.
- **Plain pickle**: simple, ubiquitous, and a security risk if the file comes from outside your own pipeline. Avoid for anything that crosses a trust boundary.

Section takeaway. Train once, `joblib.dump`, then `joblib.load` forever after. Ship the model, its fitted preprocessing, its feature schema, and its provenance as one bundle, and only load model files whose source you trust.

3 REST APIs and FastAPI: Making the Model Callable

A Model Object Is Not a Service

A loaded model lives inside one Python process. A risk dashboard, an Excel macro, a mobile app, another model: none of them can call a Python variable that exists on your laptop. You need a way for software running anywhere to send the model some features over the network and get a prediction back, without knowing anything about how the model works internally. That "way" is an API.

Think First

A colleague's spreadsheet needs the credit model's probability for one applicant. You will not give them your notebook. What is the minimum contract you and they must agree on so their code can ask your model a question and understand the answer?

3.1 What is an API?

API stands for Application Programming Interface: the agreed way one piece of software talks to another. A *web* API specifically: the caller sends an HTTP request over the network, and gets an HTTP response back. The contract is fixed and documented. The caller does not need to know the model is an MLP, or that you scaled the inputs, or which library you used. They

need to know which fields to send and which fields come back. Once a model lives behind an API, anything can use it: a website, an app, a macro, another service.

API (web API): A network-accessible contract of the form "send a request shaped like this, receive a response shaped like that." The caller and the service agree on the URL, the request format, and the response format in advance, and nothing else needs to be shared.

Finance Analogy

An API is a loan-officer terminal. The officer types in the applicant's income, debt ratio, and credit score, presses a button, and a decision comes back. They do not see the model inside the terminal; they see a form to fill in and a result. Your `/predict` endpoint is that terminal, and the JSON contract is the form.

Common Misconception

"My app can call `/predict` and the server will remember that applicant for next time." No. A REST API endpoint is *stateless*: each request is independent, the server stores nothing between calls, and the next request has no memory of the previous one. If you need to persist predictions, log them yourself (a database, a file). The `/predict` endpoint does one thing: accept features, return a number, forget both immediately. Confusing an API with a database is the most common conceptual error when building a first service.

3.2 HTTP basics: GET, POST, and JSON

Two request types cover almost everything a model service needs, and the data travels as JSON.

- **GET** means "give me something." No body, just a URL. Used for reading. A health-check endpoint, `GET /health` returning `ok`, is the textbook example: a monitoring system pings it to confirm the service is alive.
- **POST** means "here is some data, do something with it." It carries a body. A prediction request, `POST /predict` with the applicant's features in the body, is a POST.
- **JSON** (JavaScript Object Notation) is 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}`.

HTTP request / response: The message a client sends to a web service (request) and the message it gets back (response). A request has a method (GET, POST, ...), a path (`/predict`), and optionally a body. A response has a status code (200 for success, 422 for invalid input, 500 for a server error) and usually a body.

3.3 FastAPI: a model endpoint in a few lines

FastAPI is a Python framework for building web APIs. You declare an endpoint as an ordinary Python function and decorate it; FastAPI handles the HTTP plumbing. It also auto-generates interactive documentation at `/docs`, so a colleague can see exactly what to send without reading your source. The pattern, in words: on startup, `joblib.load` the model once; define a `/predict` endpoint that accepts the five features; inside it, assemble the input array in the right order, call `predict_proba`, return a JSON dictionary; run it with a server (`uvicorn`).

Code

```
from fastapi import FastAPI
import joblib

app = FastAPI()
model = joblib.load("credit_model.joblib")    # loaded ONCE, at
    startup

@app.post("/predict")
def predict(income: float, debt_ratio: float, credit_score: float,
            employment_years: float, loan_amount: float):
    X = [[income, debt_ratio, credit_score, employment_years,
        loan_amount]]
    prob = float(model.predict_proba(X)[0, 1])
    return {"approved": prob > 0.5, "probability": round(prob, 3)}
```

(This is the idea, not the production version. The real notebook adds Pydantic input validation, below, and bundles the scaler with the model. A handout shows the skeleton; the Colab shows the full thing.)

3.4 Input validation: reject bad data at the door

The model trusts its inputs. Production data has not earned that trust. A request can arrive with `income: -5000` or `credit_score: 1200`: values that nothing in the training data ever looked like. The model will not crash; it will return a confident, meaningless number. The API's job is to be the bouncer.

FastAPI uses *Pydantic* models to declare the expected shape and constraints of a request: income is a non-negative number, credit score is between 300 and 850, all five fields are present. If a request violates the contract, FastAPI rejects it with HTTP 422 ("Unprocessable Entity") and a clear message, *before* the model is ever called.

Code

```
from pydantic import BaseModel, Field

class CreditRequest(BaseModel):
    income: float = Field(ge=0)                      # non-negative
    debt_ratio: float = Field(ge=0, le=1)            # a fraction
    credit_score: float = Field(ge=300, le=850)      # the usual
        range
    employment_years: float = Field(ge=0)
    loan_amount: float = Field(ge=0)

@app.post("/predict")
def predict(req: CreditRequest):
    X = [[req.income, req.debt_ratio, req.credit_score,
          req.employment_years, req.loan_amount]]
    prob = float(model.predict_proba(X)[0, 1])
    return {"approved": prob > 0.5, "probability": round(prob, 3)}
```

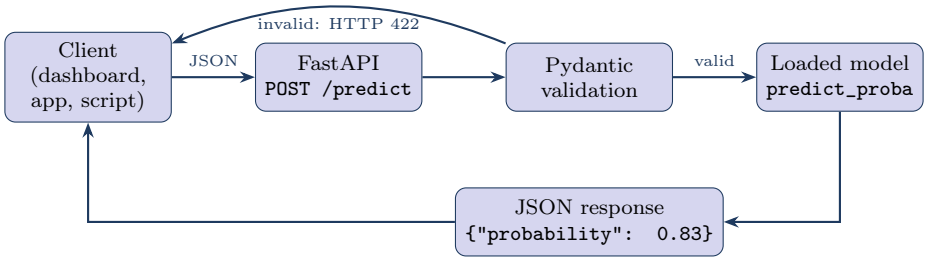
Common Misconception

"Validation is the model's problem; if the input is weird the model will figure it out." No. A model extrapolates badly and silently outside its training range. A negative income does not make `predict_proba` throw; it makes it return a number that looks just as plausible as a real one. Catch the impossible input at the API boundary, where you can return a clear error, not at the model, where the error is invisible.

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 identifier is the right length, the requested amount is plausible. Garbage is bounced at intake, not discovered three steps later. The Pydantic model is your intake clerk.

3.5 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. Two details carry most of the weight here. First, the model is loaded *once* when the server starts, not on every request: re-loading a model from disk per request is the single most common performance mistake in a first deployment, and it can make an API ten or a hundred times slower than it needs to be. Second, everything between the client and the model is glue: validation, array assembly, response formatting. The interesting machine learning happened in training; the API is plumbing, and plumbing should be boring.

What You Build in Colab A: the FastAPI Credit-Scoring Service

The Day 7 "FastAPI credit-scoring service" notebook re-creates the Day-5 credit data and MLP, then `joblib.dumps` the bundle. It builds a tiny demo API first, a `POST /add` endpoint that returns `a + b`, run inside Colab with `nest_asyncio` and `uvicorn` on `127.0.0.1`, so the HTTP mechanics are clear before the real model. Then it adds `/predict` with the Pydantic schema above. You test it from Python, not from a clickable web page: a `requests.post` with valid features returns HTTP 200 and a JSON probability; a `requests.post` with `income` negative returns HTTP 422 and the validation message. By the end you have a credit model you can call over HTTP. The checkpoints you hit, with the answer to predict before running: "what shape is the request JSON?" (an object with the five named float fields) and "what status does a negative-income request return?" (422, because the Pydantic constraint rejects it before the model runs).

Alternatives: FastAPI vs Other API Frameworks

- **FastAPI** (this course): auto-validation via Pydantic, auto-generated `/docs`, async support. Correct choice for a new Python ML service.
- **Flask**: flexible, minimal, no built-in validation. Requires manual input checking and documentation. Good for very simple scripts; fragile at scale.
- **gRPC**: binary protocol, 2 to 5 times faster than HTTP/JSON for high-throughput microservice-to-microservice calls. Steep learning curve, not browser-friendly. Correct when latency matters at volume, not for a first deployment.
- **Django REST Framework**: full-featured, includes authentication, ORM, admin panel. Correct for a large web application; overkill for a single `/predict` endpoint.

Section takeaway. An API hides the model behind a fixed JSON contract: POST features in, get a prediction out. Load the model once at startup, validate every request with Pydantic, and let the model see only clean input.

4 Streamlit: Making the Model Interactive

APIs Serve Machines; Humans Want to Click

A `/predict` endpoint is perfect for software. A portfolio manager is not software. They want a screen with a slider for "number of regimes" and a chart that re-draws as they drag it. Building a real web front end (HTML, JavaScript, a server, callbacks) is a project in itself: weeks of work and a different skill set. You need a way to turn the Python analysis you already wrote into a clickable web app without becoming a front-end developer.

Think First

Your Day-5 K-Means analysis of 30 stocks is a linear Python script: load returns, standardize, fit K-Means with `k=4`, plot the clusters, plot the PCA biplot. If a tool re-ran that whole script every time a user changed a control, the dashboard would work, but what would feel wrong about it, and what cheap fix would make it feel instant?

4.1 A script becomes an app

Streamlit collapses the front-end project into nothing. You write an ordinary top-to-bottom Python script. Streamlit re-runs it whenever a control changes and renders the output as a web page. Every `st.slider` is a control; every `st.pyplot` is a live chart; every `st.dataframe` is a sortable table. There is no front-end code you write, no callbacks, no server you configure. The mental model is one sentence: *a Python script that re-runs on every interaction.*

Code

```
import streamlit as st

st.title("Stock Clusters")
k = st.slider("Number of clusters", 2, 8, 3)    # a slider; k holds
    its value
ticker = st.selectbox("Highlight ticker", tickers)

fig = make_cluster_plot(returns, k, highlight=ticker)    # your Day
   -5 plotting code
st.pyplot(fig)                                          # drawn live; re-
   runs when k changes
```

Streamlit: A Python library that turns a script into a web app. Widgets such as `st.slider` and `st.selectbox` produce interactive controls; output functions such as `st.pyplot` and `st.dataframe` render results. The whole script re-executes top to bottom every time the user interacts with a widget.

Finance Analogy

A Streamlit dashboard for the Day-5 stock clusters is a portfolio-manager terminal. Drag the "number of regimes" slider from 3 to 6, and the groupings of the 30 stocks re-draw in front of the manager. No code, no API call, no waiting for a developer: a control and a picture, which is exactly what a non-programmer stakeholder wants.

4.2 Caching: do not re-compute what has not changed

Here is the "what feels wrong" from the discovery question. Streamlit re-runs the whole script on every interaction. Without caching, that means re-loading the CSV and re-fitting K-Means every time the user nudges the ticker dropdown, even though the data has not changed. The app feels sluggish.

The fix is two decorators. `@st.cache_data` on a function that returns data (a DataFrame, an array) means "run this once, remember the result, re-use it on every re-run unless the inputs change." `@st.cache_resource` is the same idea for expensive objects you do not want copied, like a loaded model: load it once, share it across interactions. With caching, only the cheap part actually re-runs: re-drawing the scatter plot for the new `k`. The first run pays the cost; every later run is free.

Code

```
import streamlit as st

@st.cache_data                          # run once, cache the
    DataFrame
def load_returns():
    return build_30_stock_returns()   # the Day-5 data-generation,
        called once

@st.cache_resource                      # cache the heavy object, do
    not copy it
def get_model():
    return joblib.load("credit_model.joblib")

returns = load_returns()                # instant after the first
    call
k = st.slider("Number of clusters", 2, 8, 3)
st.pyplot(make_cluster_plot(returns, k))    # only this re-runs on
    each drag
```

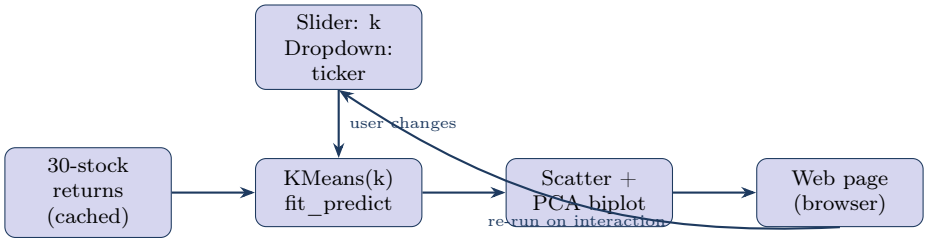
Common Misconception

"The dashboard is slow because Streamlit is slow." Almost always it is slow because an expensive step (loading data, fitting a model, downloading a file) runs on every interaction instead of once. Wrap that step in `@st.cache_data` or `@st.cache_resource` and the same dashboard becomes instant. Caching is not an optimization you add later; for any non-trivial Streamlit app it is part of writing it correctly.

Worked Example: Cache Hit-Rate in Practice

Suppose the user drags the cluster slider 10 times. Without caching, each drag re-loads the CSV and re-fits K-Means: $10 \times 2.0\,\text{s} = 20.0\,\text{s}$ total. With `@st.cache_data` applied to the data-loading function, run 1 is a cache miss (2.0 s); runs 2 to 10 are cache hits, each a hashmap lookup returning the already-computed result, at around 0.01 ms each. Total: $\approx 200.9\,\text{ms}$, a 99.95 % reduction. The script logic is unchanged; the decorator is the only addition. For real datasets where the load step takes 30 seconds or more, the difference between a cached and an uncached dashboard is the difference between usable and not.

4.3 The interaction loop



The user drags the slider, K-Means re-runs with the new `k`, the plot re-draws, the page updates. The cached data load does not repeat. That is the whole loop: a control changes, the script re-runs, only the uncached parts recompute.

What You Build in Colab B: the Streamlit Stock-Cluster Dashboard

The Day 7 "Streamlit stock-cluster dashboard" notebook re-creates the Day-5 30-stock, 252-day return data, then builds the dashboard as a real `app.py` file (written to disk with `%%writefile`). It shows a three-line demo app first (`st.title` plus one `st.slider` plus `st.write`) so the "script becomes app" idea is concrete. Then it shows the wrong way: an `app_slow.py` that re-fits K-Means on every interaction, immediately followed by the `@st.cache_data` fix. The complete `app.py` has a slider for `k` (2 to 8), a dropdown for ticker, a live scatter plot of the clustering, and the PCA biplot. The notebook shows the flaky in-Colab preview path (a tunnel such as `cloudflared`) and then the path that actually works for shipping: write `app.py` and `requirements.txt`, drag both into a Hugging Face Space in the browser, get a public URL. The checkpoints: "when the user drags `k` from 3 to 6, what changes in the scatter plot?" (more, smaller clusters; some points change color) and "why does the app feel sluggish without `@st.cache_data`?" (the data load and the fit re-run on every widget change instead of once).

Alternatives: Streamlit vs Other Dashboard Libraries

- **Streamlit** (this course): Python-first, entire app in one script, minimal learning curve. Correct for data scientists who want a result today.
- **Dash** (Plotly): more layout control, callback-based reactivity. Better for complex dashboards with many interacting panels; steeper learning curve than Streamlit.
- **Gradio**: one-liner ML demos. Define inputs and outputs; Gradio draws the interface. Correct for sharing a single model prediction form, not for multi-chart analysis.
- **Panel** (HoloViz): multi-page apps, stronger notebook integration. More powerful than Streamlit for complex layouts; more complex to learn.
- **React + FastAPI**: production-grade, full layout control, used by real software teams. Correct when UX quality matters more than developer speed; weeks of work, not hours.

Section takeaway. Streamlit turns a linear Python script into a clickable web app: widgets become controls, output functions become live charts, and the script re-runs on every interaction. Cache the expensive steps so only the cheap re-draw happens each time.

5 RAG in Production: Grounded Generation

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. It 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 that is disqualifying: an analyst cannot act on a number a model made up, and 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.

Think First

You have 200 chunks of earnings-call transcript and a question: "what was revenue growth?" One tempting plan is to paste all 200 chunks into the prompt and let the model find the answer. Name two distinct reasons that plan breaks, even if you could fit all the text in. (Hint: one is about size, one is about signal.)

5.1 RAG = Retrieval-Augmented Generation

RAG stands for Retrieval-Augmented Generation. The recipe: retrieve the relevant text, stuff it into the prompt, then generate an answer grounded in that text. It has two halves with very different trust profiles.

- **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-over-sentence-embeddings skill from Day 5: nothing is generated, nothing is invented, you are just looking things up.
- **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 model can still phrase the answer naturally, but it is now anchored to text you can show the analyst.

The prompt the model actually sees becomes something like: "Here are three passages from the earnings call. Using only these, answer: what was revenue growth?" The model answers in fluent prose, but the prose is constrained to the supplied passages.

Retrieval-Augmented Generation (RAG): A pattern for question-answering over your own documents: (1) embed the documents and store the vectors; (2) at query time, embed the question and retrieve the most similar document chunks; (3) build a prompt containing those chunks plus the question; (4) have a language model generate the answer from that prompt. The generation is "grounded" because it is constrained to the retrieved text.

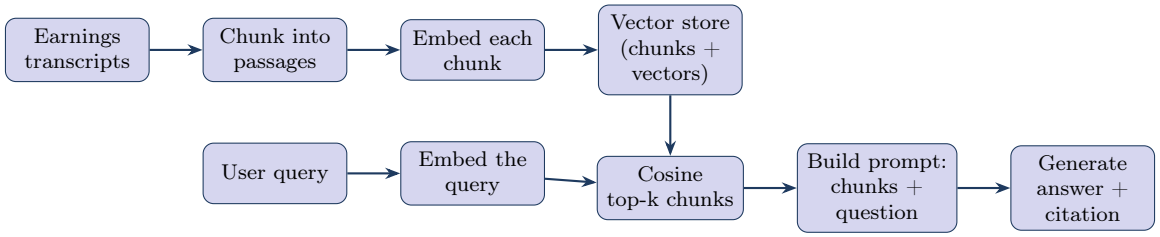
Finance Analogy

A bare language model is 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. The skill is the same; the difference is whether the answer points to a source the next person can check.

Common Misconception

"An embedding is just a fancy word-count vector." It is not. A word-count (or TF-IDF) vector is *sparse*: each dimension is one vocabulary entry, almost all values are zero, and two documents with no words in common have zero overlap. A sentence embedding is *dense*: 384 or 768 floating-point values encode the *meaning* of the whole sentence, not its words. Two sentences with zero word overlap can have a cosine similarity of 0.85 because they mean the same thing in different words ("revenue grew sharply" and "sales surged"). This is exactly what makes retrieval work: the query does not need to share words with the relevant chunks; it only needs to share meaning.

5.2 The pipeline: chunk, embed, retrieve, prompt, generate



Steps 1 to 4 (chunk, embed, store) happen once, offline, when the documents arrive. Steps 5 to 8 (embed query, retrieve, build prompt, generate) happen on every question. "Chunking" means splitting a long document into passages small enough to embed and to fit several of them in a prompt: a sentence, a paragraph, a fixed window of tokens. The "vector store" in a small project is just a list of chunks and a matrix of their embeddings; in a large one it is a specialized database, but the idea is identical.

5.3 Why top-k, not all chunks

This is the discovery question made explicit. Stuffing every chunk into the prompt breaks two ways:

1. **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 part that mattered.
2. **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.

The fix is top-k retrieval: keep only the `k` chunks closest to the query, with `k` typically 3 to 5. 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 it. Start at 3 and tune by reading the outputs.

Top-k retrieval: Returning only the `k` document chunks whose embeddings are most similar to the query embedding (highest cosine similarity), rather than all of them. The retrieval step exists precisely to make this selection: it is the filter that keeps the generator's prompt small and relevant.

Common Misconception

"More context is always better, so retrieve everything." No. Beyond the relevant passages, extra context is pure noise that competes for the model's attention and risks overflowing the prompt. RAG works *because* retrieval is selective. If you find yourself wanting all the chunks, that is a sign your chunks are too small or your retrieval is too weak, not that you should skip the filter.

Retrieval by Hand: Cosine Similarity Picks the Chunks

Suppose, after embedding, six earnings-call chunks have these cosine similarities to the query "what was revenue growth?" (similarity runs from -1 to 1; higher means closer in meaning):

Chunk Topic	C1 revenue	C2 margins	C3 guidance	C4 revenue	C5 hiring	C6 revenue
Cosine to query	0.81	0.22	0.34	0.74	0.09	0.69

With $k = 3$, retrieval keeps the three highest: C1 (0.81), C4 (0.74), C6 (0.69). The prompt becomes those three revenue-related passages plus the question. Note what is excluded: C2 (margins), C3 (guidance), C5 (hiring) score low and stay out, which is exactly right. Now set $k = 6$: every chunk goes in, including the three off-topic ones, and the model must answer a revenue question while reading about hiring and margins. The prompt is twice as long and half as relevant. The numbers make the rule concrete: retrieval is the step that turns "all six chunks" into "the three that matter," and k is the dial that controls how aggressive that filter is.

5.4 Citations: the answer must point to its source

A grounded answer with no citation is only half-grounded. When the model says "revenue grew 12 percent," the system should also surface *which retrieved chunk* that came from: the transcript passage, ideally with the speaker and a position. The retrieval step already knows which chunk it returned; carry that identifier through the prompt and into the displayed answer. This is what makes RAG usable in finance: the analyst clicks through to the source and confirms before acting. "Trust, but verify" becomes mechanical. 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. The citation is not a nice-to-have; it is the feature that distinguishes "grounded" from "confident."

5.5 Using a small local model

You do not need a paid API to do the generation half. `google/flan-t5-small` is a small instruction-tuned model (a few hundred megabytes) that runs on a free CPU and is loaded through `transformers.pipeline("text2text-generation")`. It is not eloquent, but it follows the instruction "answer from this context" well enough to demonstrate real RAG: retrieval plus generation, not retrieval plus template-filling. If even that download stalls on a slow connection, an honest fallback is an extractive responder that returns the single top retrieved snippet with a one-line framing, clearly labeled as degraded. The point of the fallback is to keep the lesson moving; the point of the model is to show that the generated answer is genuinely synthesized from the retrieved passages.

Code

```
from transformers import pipeline
generator = pipeline("text2text-generation", model="google/flan-t5
    -small")  # load once

def rag_answer(question, chunks, chunk_embeddings, k=3):
    # 1. retrieve: cosine top-k chunks for this question
    top = top_k_chunks(question, chunks, chunk_embeddings, k)
    # 2. build a grounded prompt: only the retrieved chunks plus
        the question
    context = "\n".join(f"[{i+1}] {c}" for i, c in enumerate(top))
    prompt = f"Context:\n{context}\n\nUsing only the context,
        answer: {question}"
    # 3. generate, and report which chunks were used
    answer = generator(prompt, max_new_tokens=80)[0]["
        generated_text"]
    return answer, top                              # the answer AND
        its sources
```

What You Build in Colab C: the RAG Earnings Chatbot

The Day 7 "RAG earnings chatbot" notebook re-uses the Day-5 earnings-call snippets and the same small embedding model (`paraphrase-MiniLM-L3-v2`, local, no API key). It starts with retrieval over three toy sentences so "embed, find the closest" is concrete. Then the full pipeline: chunk each snippet into sentences, embed every chunk once, and for a query embed it and take the cosine top-k. The wrong-way cell retrieves *all* chunks and shows the prompt ballooning; the fix is top-k. It then builds the grounded prompt (retrieved context plus the question, nothing else) and generates with `flan-t5-small`, returning the answer together with the snippets it came from. A second checkpoint compares `k=3` against `k=10` so you see the prompt grow and the relevant fraction shrink. Finally it writes a tiny Gradio `app.py` and a `requirements.txt` and walks through dragging both into a Hugging Face Space, browser only, no GitHub repo. The checkpoint to predict before running: "for the query revenue growth, which snippets get retrieved, and what does the final prompt look like?" (the chunks that mention revenue; the prompt is those chunks, numbered, followed by the question).

Section takeaway. RAG = trusted retrieval (Day-5 embeddings) plus grounded generation (a small local model). Retrieve only the top-k chunks so the prompt stays small and on-topic, and always carry the source chunk's identity into the answer.

6 Containers and Cloud: Shipping It

"Works on My Machine"

Your API runs fine on your laptop. You hand it to a colleague and it breaks: a different Python version, a missing library, a different operating system. Software depends on its environment, and the environment 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. You need a way to package the code together with its environment, so wherever it runs, it runs the same.

Think First

Suppose you could put your `app.py`, your model file, your exact list of package versions, and a minimal operating system into a single sealed bundle, and any server could run that bundle without installing anything itself. What would you call such a bundle, and what is the recipe that produces it?

6.1 Docker: the recipe and the image

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. The recipe for an image is a `Dockerfile`: "start from this base, copy in this code, install these packages, run this command." `docker build` turns the recipe into the image; `docker run` starts a container from it. The same image runs identically on your laptop, a teammate's, or a cloud server.

Container / image / Dockerfile: A *container* is an isolated process running with its own bundled filesystem and dependencies. An *image* is the frozen, shippable bundle a container is started from. A `Dockerfile` is the text recipe that builds an image: a base image, files to copy in, packages to install, and the command to run.

Code: a minimal Dockerfile for the credit-scoring API

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY app.py credit_model.joblib ./
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

Six lines. `docker build -t credit-api .` bakes the image; `docker run -p 8000:8000 credit-api` serves it. The bank's platform team can now run that exact image in their cluster without asking which Python version you used.

Finance Analogy

Containerizing a model is shipping it in a sealed box with the exact runtime inside. The receiving desk does not assemble anything, does not hunt for the right version of a dependency, does not discover on a Friday that their Python is too new. They open the box and the same thing you tested is the same thing that runs. For a regulated model, "the same thing you tested" is the whole point.

Worked Example: What Is Actually Inside a Docker Image

A minimal `FROM python:3.11-slim` base image is around 120 MB. After `pip install` of the ML stack (NumPy, scikit-learn, FastAPI, uvicorn), the image grows to around 950 MB. The model artifact `credit_model.joblib` is around 8 KB: less than 0.001 % of the image. The dominant cost is the Python packages, not the model. Adding `-no-cache-dir` to the `pip install` instruction saves roughly 180 MB by discarding the pip download cache before the layer is committed. These are illustrative estimates; exact sizes depend on package versions, but the orders of magnitude are reliable: base is ∼120 MB, deps add ∼830 MB, model adds ∼8 KB.

6.2 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 free tiers a student can use in class. They 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."

- **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 the Day 7 notebooks use.
- **Streamlit Community Cloud.** Connects to a *public* GitHub repo and deploys the Streamlit app on every 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, so the first call after a quiet spell is slow.

Key Idea

The matching rule: dashboard or chatbot, use Hugging Face Spaces (drag-and-drop). FastAPI endpoint, use Render or Railway. Public-repo project, Streamlit Community Cloud also works. Pick the host that fits the workload and the repo situation; do not reach for Docker or a paid cloud until a managed host genuinely cannot do what you need.

6.3 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 entirely:

1. Write your app as a single `app.py` (a Streamlit dashboard, or a small RAG chatbot interface).
2. List its dependencies in a `requirements.txt`, one package name per line. The stock dashboard needs `streamlit`, `scikit-learn`, and `pandas`; the RAG chatbot additionally needs `sentence-transformers` and `transformers`.
3. Create a new Hugging Face Space, choose "Streamlit" or "Gradio," and drag the two files in.
4. 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 Colab 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.

Common Misconception

"Docker gives you a separate computer inside your computer." Not quite. A *virtual machine* (VM) has its own kernel: it boots a full operating system, which takes 20 to 60 seconds and consumes gigabytes of RAM for the kernel alone. A *container* shares the host OS kernel: only the filesystem and process namespace are isolated. A container starts in under a second, uses tens of megabytes of overhead, and runs the same kernel as everything else on the host. On Windows, Docker Desktop uses **WSL2** (Windows Subsystem for Linux 2) to provide a Linux kernel, which is why Linux containers run on Windows at all. The practical difference: a container is closer to a very isolated process than to a separate computer.

Section takeaway. A container ships the code together with its runtime, so "works on my machine" becomes "works everywhere"; a `Dockerfile` is the recipe. To ship a class project, though, the shortest path is `app.py` plus `requirements.txt` dragged into a Hugging Face Space, no Docker required.

7 The MLOps Lifecycle: Keeping It Alive

Deployment Is Not the End: Models Decay

A credit-scoring model trained on 2015 to 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. "Trained and deployed" is the start of a model's working life, not the end of the project. The maintenance phase is longer than the build phase.

Think First

You have a credit model live in production. Name one thing you could watch *today*, before any loan in this batch has had time to default, that would warn you the model might be going wrong. Then name the thing you can only check *later*, once outcomes arrive, that is the real gold standard. Why do you need both?

7.1 Two kinds of drift

Models go stale in two distinct ways, and telling them apart tells you what to do.

- **Data drift.** The *inputs* change distribution. Average applicant income shifts, a new product brings a different customer mix, a new region's data starts flowing. The model's learned mapping is still fine; it is just seeing a population it was not trained on. Detect it by tracking input feature distributions over time and comparing to the training distribution.
- **Concept drift (model drift).** The *relationship* between inputs and outcome changes. The same income and credit score that meant "low risk" in 2019 mean something different after a regime change. The model's learned mapping is now wrong. Detect it by tracking prediction accuracy against actual outcomes as they arrive.

Data drift is a warning sign; concept drift is a failure. Both eventually require re-training, but concept drift requires it *urgently*.

Data drift vs concept drift: Data drift: the distribution of the model's inputs changes, but the input-to-output relationship is unchanged. Concept drift: the input-to-output relationship itself changes. Data drift you can often tolerate for a while; concept drift means the model is now systematically wrong and must be re-trained.

Finance Analogy

Monitoring a deployed model is the annual model re-validation a risk team performs: every year, the model's recent performance is reviewed against actual outcomes, the input population is checked for shifts, and a decision is made on whether it still meets the standard. The deployed model needs that same discipline, not as a one-off audit but as a continuous one. A model is a perishable asset; MLOps is the practice of watching it age.

7.2 Monitoring: what to watch on a live model

You cannot fix what you cannot see. A deployed model needs a dashboard about itself, tracking four things:

1. **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.
2. **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.
3. **Latency and error rates.** Is the API responding in milliseconds, or timing out? Are requests returning 422 (bad input) or 500 (something broke)? A rising error rate is an operational problem regardless of model quality.
4. **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, sometimes by months: you act on the leading indicators first and confirm with this one.

Key Idea

Log every prediction, with its inputs, its output, and a timestamp. Then you can compute all four of the above retrospectively and reconstruct what the model was doing on any past day. Without logging, you are blind, and "the model seemed fine" is not an answer a risk committee accepts.

Common Misconception

"If accuracy on the holdout set was good, the deployed model is good." That number describes the past. The deployed model is judged on a population that did not exist when you measured it, under relationships that may have moved. A holdout score is a launch gate, not a guarantee; only ongoing monitoring tells you whether the model is still earning its place.

Reading a Monitoring Dashboard: Three Signals, One Story

Your credit-scoring API has been live for three months. The monitoring dashboard shows:

- **Approval rate:** steady at 58 to 61 percent for ten weeks, then jumps to 74 percent in the last two weeks.
- **Input distribution:** the mean `debt_ratio` of incoming applications fell from about 0.34 to about 0.21 over the same two weeks; other features look normal.
- **Accuracy against actuals:** unchanged, but the most recent loans have not had time to mature, so this number reflects applicants from two months ago, not the recent shift.

What happened? Almost certainly a new, lower-debt customer segment started flowing in (a marketing campaign, a partnership, a new product). That is *data drift*: the model's mapping is probably still correct, it is just seeing lower-risk applicants than before, so it approves more of them. The right response is not an emergency re-train; it is to confirm the new segment is genuine (not a data-pipeline bug feeding stale or wrong values), watch the accuracy-against-actuals number as those recent loans mature, and schedule a re-validation if the population stays shifted. Contrast this with the alarm case: if the approval rate jumped *and*, two months later, the default rate among those approved climbed well above what the model predicted, that would be *concept drift*, and that one is the Tuesday-night re-train.

7.3 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 the metrics recorded. You can always say which model was live on which day, which is exactly what an auditor will ask.
- **CI/CD for machine learning.** "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 unit tests before merging code: the gate is automatic, and nothing reaches production without passing it.
- **Rollback.** Reverting to the previous version must be one command. If `v8` misbehaves in production, you go back to `v7` immediately, then investigate at leisure.

Alternatives: Three Drift-Response Paths

When monitoring signals drift, the right response depends on the type and severity.

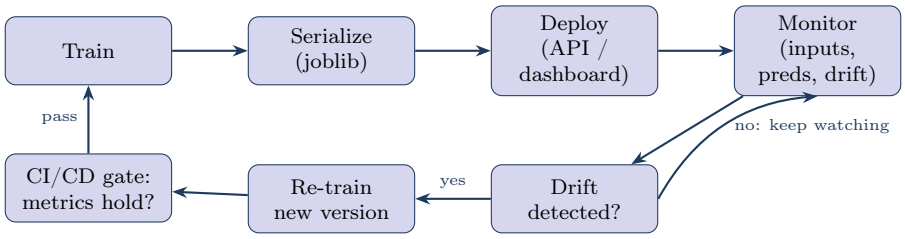
- **Retrain immediately** (emergency mode): fast response, but risks overfitting to a transient outlier or a data-pipeline glitch. Use only when the drift metric is large and sustained, not for a single anomalous week.
- **Rolling-window retrain** (standard mode): retrain on a sliding window of the most recent N months of data. Smooths gradual distributional shifts while discarding stale historical data. The standard approach for a credit model in a slowly-changing economy.
- **Human-review-first** (regulated mode): flag the drift, pause automated approvals above a threshold, and route borderline cases to a human analyst until re-validation is complete. Required in regulated finance for concept-drift events, where the bank must demonstrate it did not continue using a model it knew was wrong.

CI/CD for ML: An automated pipeline that, on a new model version, retrains (or loads the candidate), evaluates it on a fixed held-out set, compares its metrics to the current production model, and promotes it only if it does not regress. Combined with versioned artifacts and a one-command rollback, it makes model updates routine and reversible rather than risky and manual.

Finance Analogy

Promoting a new model version through a CI/CD gate is a model passing internal validation before the risk committee signs off. No model reaches production without clearing the gate, and any model in production can be pulled. The gate and the rollback are not bureaucracy; they are what let you change a live model at all without holding your breath.

7.4 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 it: the model from Days 2 to 6, serialized today, served today, and from now on, watched.

Section takeaway. A deployed model decays as the world moves: data drift moves the inputs, concept drift moves the rules. Log every prediction, monitor inputs, predictions, latency, and outcomes, and re-train through a CI/CD gate with a one-command rollback. Deployment is one station in the loop, not the terminus.

8 What You Can Now Do, and Where to Go Next

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

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 machine learning toolkit: you can take a dataset, train a model, serve it, and explain when it will fail. That is the whole pipeline. Everything else is depth.

A deployment checklist

When you take one of your own models from notebook to live URL, this is the short list to run through. It is the whole handout in one screen.

1. **Serialize the right bundle.** `joblib.dump` the model *and* its fitted preprocessing (a `Pipeline` is cleanest), and record the feature schema and the training run. Load only artifacts your own pipeline produced.
2. **Pin the preprocessing into the serving path.** Whatever happened before `model.fit` must happen, identically, before `model.predict` in production. Test that the serving code and the training code give the same prediction on the same row.
3. **Validate inputs at the boundary.** A Pydantic model (for an API) or a bounded widget (for a dashboard) that rejects impossible values before the model sees them. Decide what "impossible" means per feature and enforce it.
4. **Load the model once.** At startup for an API, behind `@st.cache_resource` for a Streamlit app. Never re-load it per request or per interaction.
5. **Cache the expensive steps.** In a dashboard, wrap data loading and model fitting in `@st.cache_data`. The first run pays; every later one is free.
6. **For RAG, retrieve top-k and cite.** Never put all chunks in the prompt; keep the k most similar, and carry each retrieved chunk's identity through to the displayed answer.
7. **Ship with the environment.** A `requirements.txt` at minimum; a `Dockerfile` if you need to run on arbitrary infrastructure. For a class project, `app.py` plus `requirements.txt` dragged into a Hugging Face Space is enough.
8. **Log every prediction.** Inputs, output, timestamp. Without the log you cannot answer "what was the model doing last Tuesday?", and that is a question someone will eventually ask.
9. **Watch four things.** Input distributions, prediction distributions, latency and error rates, accuracy against actuals. The first three are leading indicators; the last is the gold standard

but delayed.

10. **Plan the re-train and the rollback before you need them.** A trigger (scheduled and/or threshold-based), a versioned artifact registry, a CI/CD gate that checks metrics did not regress, and a one-command revert to the previous version.

Where to go next

- **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.

From a financial dataset on Day 1 to a deployed, monitored finance ML service on Day 7. You can train a model, serve it, and explain when it will fail; that is the pipeline, and you now own it end to end.