Integrate the Model

2026-05-09 · hopeful nectarine Specht

Train hard, ship easy.

The Why

Block 3 left us with a reproducible environment on both machines. The virtual environment is ready, the dependencies are pinned, the project foundation is in place. A foundation that would work for all sorts of different projects by the way. Let's get our project some specific functionality.

We build PixelWise. This block is the first that produces application logic in the form of a neural network, in the following referred to as model. We assume the model is already trained and available, and our task is to integrate it into the codebase and make it callable.

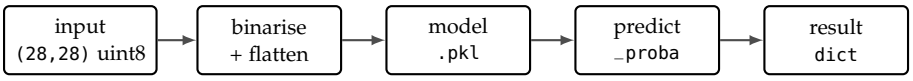

Figure 1:
The inference pipeline for one image: the caller binarises a raw `(28,28)` `uint8` array at threshold 128 and flattens it into a `(1,784)` row vector before feeding it to the loaded `.pkl` model. The model's internal `Binarizer` is a redundant idempotent guard on already-binarised input, and `predict_proba` turns the result into a dictionary of class scores.

The gap between a trained model and a working system is larger than most students expect. A `.pkl` file sitting in a folder is not a product. It needs to be loaded safely, called with the exact input format it was trained on, and wrapped in an interface the rest of the application can depend on, suffice to say validated. The perfect mdoel however, is not useful if it is not integrated into the system, so it can be exposed and put to work.

Hands On Experience

Consider this scenario: a colleague hands you a trained model file and a one-line note, "ship it". This block teaches you how to, and why you should, ask the right questions to your colleague.

This fourth block integrates a trained model into PixelWise and builds the inference layer. By the end you will have

- understood what a model file is and why it is not source code,
- loaded the model safely, learned about and verified its class contract at startup,
- implemented a batch-first inference interface in `app/classifier.py`,
- written a smoke test that proves the integration honours the contract,
- and read the canonical model card alongside your introspection output to verify the contract.

Verification and validation are two different questions. Verification asks whether the system was built right: does the integration return what the model is trained to produce, with the expected shape, dtype, and class set? Validation asks whether the right system was built: is this model the right choice for the problem in the first place and is the performance sufficient? This block is squarely about verification. Validation belongs to pre-integration.

Optional Exercise: Catch Up to `v0.2`

CATCHING UP VIA TAG. This catch-up extends the one in Block 3. If you have not yet done the Block 3 catch-up, do that first: it sets up your GitHub repository, SSH key, Git identity, and remote URL, and lands you at `v0.1`, the end-of-Block 2 state. The exercise here picks up from there and rewinds to `v0.2`, the end-of-Block 3 state, so you arrive at the starting point this block expects: a provisioned VM with the virtual environment, pinned dependencies, and `.env.example` in place.

TAG MAP. `v0.2` marks the end of Block 3, `v0.3` will mark the end of this block. Browse the full set at https://github.com/schutera/pixelwise/tags.

TWO LAYERS TO RESTORE. Block 3 produced both Git-tracked files (`setup-server.sh`, `requirements.txt`, `.env.example`, plus an updated `.gitignore`) and unversioned state (system `apt` packages, the `.venv/` directory, real `.env` values). `git reset --hard v0.2` restores the first; the second has to be reproduced by replaying Block 3's setup commands. Both VMs need both layers, so run the sequence below on each.

RUN THE SAME SEQUENCE ON BOTH VMS. Rewind the repository to `v0.2`, reinstall system packages, recreate the virtual environment, reinstall pinned dependencies, and copy `.env.example` to `.env`. The dev sequence runs in `~/pixelwise`; the prod sequence is identical except the directory is `/opt/pixelwise` and you reach it via `ssh produser@192.168.56.11` first. The exact commands are in the margin.

YOU ARE NOW at the state where Block 3 ended, with both machines aligned, ready to start Block 4. Did you realize how seamless this was, that is thanks to our solid git practices, and our dependency and setup scripts.

SEQUENCE ON `dev`.

```
cd ~/pixelwise
git fetch origin --tags
git reset --hard v0.2
bash setup-server.sh
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
cp .env.example .env
```

Then edit `.env` with real values.

SEQUENCE ON `prod`.

```
ssh produser@192.168.56.11
cd /opt/pixelwise
git fetch origin --tags
git reset --hard v0.2
bash setup-server.sh
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
cp .env.example .env
```

What a Model File Actually Is

A `.pkl` file is a build artefact, not source code. You distribute it; you do not commit it to your repo. Just as a compiled binary is produced by a compiler, a model file is produced by a training script. The training script and its configuration is the source of truth: It takes data, runs an algorithm, and writes weights, hyperparameters, and any preprocessing into a single serialised object.

Training from scratch every time is unrealistic. Even our small classifier takes seconds; a production scale model takes days of GPU time and terabytes of data. The training team tunes, the inference team integrates, and the artefact is the contract between them.

Versioning makes that contract durable. Files carry a version in the filename or in metadata shipped alongside. A retrain on more data bumps the patch version; an architecture change or a different class set bumps the major version. Integration code pins to a specific version and updates deliberately, the same discipline you would apply to any external API. Open source tooling that automates this includes DVC for git-style data and model tracking, MLflow and ClearML as self-hostable experiment registries, Aim as a lighter-weight tracker, and the Hugging Face Hub itself, which is free for public repositories and versions every push as a git revision.

Pulling a model from the Hub.

```
from huggingface_hub import
hf_hub_download
path = hf_hub_download(
  repo_id="distilbert-base-uncased",
  filename="model.safetensors",
  revision="v1.0",
)
```

Hugging Face is the GitHub of models. The Hugging Face Hub at `https://huggingface.co` hosts hundreds of thousands of public model repositories, each with versioned weights, a model card describing it. Downloading a specific revision is a one-line call against the `huggingface_hub` library. The same workflow runs against private repositories with a token, which is how many companies share proprietary models internally. For this course the artefact lives in the course-maintained model repository at `https://github.com/schutera/pixelwise-model`, with a tagged release per model version and the model card sitting next to it.

The training script `train.py` lives in the course `pixelwise-model` repo alongside the artefact it produces, not in PixelWise. The artefact and the script that produced it travel together, so any consumer of a tagged release can audit how that exact `.pkl` came to be. Students who want to understand how the artefact was produced can run it: MNIST downloads automatically via `sklearn.datasets.fetch_openml`, and training takes ∼30 seconds on CPU. Further reading on training practices: Karpathy, A Recipe for Training Neural Networks: `https://karpathy.github.io/2019/04/25/recipe/` Google, Rules of Machine Learning: `https://developers.google.com/machine-learning/guides/rules-of-ml`

Loading a `.pkl` from an untrusted source can execute arbitrary code. `pickle` and `joblib` deserialise objects by reconstructing them, including any code embedded in the file. This is convenient when you produced the file yourself; it is dangerous when the file came from somewhere you do not control. Treat `.pkl` files the way you treat executables: only run what you trust.

Exercise: Pull the Model into Dev

TREAT THE MODEL ARTEFACT AS SOMETHING YOU LOAD, not something you commit to PixelWise. The course hosts the model `pkl` as a tagged release in the central `pixelwise-model` repo; you pull it into `models/` of the main project, you do not redistribute it.

THE COURSE ARTEFACT. at `https://github.com/schutera/pixelwise-model` `digit_classifier_v1.pkl` is a LogisticRegression pipeline trained on MNIST digits 1 to 9 (9 classes, ∼54,000 training samples, public domain). Class 0 is intentionally held back; students can add it later as a v2 release without collecting any new data.

PIN THE MODEL VERSION in PixelWise. Add to `.env.example` so others know which release the integration expects, and `.env` so your code knows:

```
MODEL_REPO=https://github.com/schutera/pixelwise-model.git
MODEL_VERSION=v1.0
```

WHY PULL FROM A SEPARATE REPO? The model has its own release cadence and its own contract. Pinning a specific tag in PixelWise's `.env` makes the integration explicit: a v2 only takes effect when the integration code chooses to. The optional exercise at the end of the block walks through publishing your own v2 to a fork of the model repo.

TEACH `setup-server.sh` TO FETCH THE MODEL. Pulling the artefact by hand once is fine for understanding what happens; encoding the same steps in `setup-server.sh` means that re-running the script on `prod`, on a fresh VM, or after a snapshot restore brings the model back without any extra commands. Open `setup-server.sh` in nano and append a model-pull block after the existing `apt install` lines:

```
nano setup-server.sh
```

Add the following at the bottom of the file, save with `Ctrl+O`, and exit with `Ctrl+X`:

```
# Pull the pinned model artefact
if [ -f .env ]; then
    set -a; source .env; set +a
    if [ -n "${MODEL_REPO:-}" ] && \
       [ -n "${MODEL_VERSION:-}" ]; then
        mkdir -p models/
        rm -rf /tmp/pixelwise-model
        git clone --depth 1 --branch "$MODEL_VERSION" \
            "$MODEL_REPO" /tmp/pixelwise-model
        cp /tmp/pixelwise-model/*.pkl models/
        cp /tmp/pixelwise-model/MODELCARD.md models/
        rm -rf /tmp/pixelwise-model
    fi
fi
```

WHY ALL THE GUARDS? The `if [ -f .env ]` check lets the script run on a VM that has not yet been configured (for example, immediately after the Block 3 catch-up). The `rm -rf /tmp/pixelwise-model` before `git clone` keeps the script idempotent: a leftover temp directory from a previous run does not block the next one. Together these turn the model pull into a re-runnable provisioning step rather than a one-shot command.

RUN THE UPDATED SCRIPT ON `dev`:

```
bash setup-server.sh
ls models/
git status
```

IF `digit_classifier_v1.pkl` APPEARS IN `git status`, your `.gitignore` from Block 3 needs an update. Add `models/*.pkl` on its own line and rerun `git status`. A model file accidentally committed once is in the history forever, even if the next commit deletes it.

CONFIRM THE ARTEFACT LOADS in a Python shell on dev:

```
source .venv/bin/activate
python3
>>> import joblib
>>> model = joblib.load("models/digit_classifier_v1.pkl")
>>> type(model)
```

If `joblib.load` returns a `Pipeline` object without raising, the artefact is sitting at `models/digit_classifier_v1.pkl` on your dev VM, ready to be scrutinised next.

COMMIT THE SCRIPT AND `.env.example`, then mirror the change to `prod`:

```
git add setup-server.sh .env.example
git commit -m "Pull model artefact in setup-server.sh"
git push

ssh produser@192.168.56.11
cd /opt/pixelwise
git pull origin main
cp .env.example .env       # Don't forget to edit .env to fir production
bash setup-server.sh
ls models/
```

MACHINE SETUP AND MODEL PROVISIONING BECOMES A SINGLE COMMAND. After this block, both `dev` and `prod` are set up by one command, `bash setup-server.sh`, regardless of whether the machine setup or the model artefact, or both. Optionally also do that for the virtual environment and the system packages.

The Model Contract and Its Card

Every model was trained on inputs in a specific format. The format is part of the contract: a 28-by-28 greyscale image, pixel values in $[0, 1]$, flattened to 784 features. Most violations of this contract are loud: a wrong shape raises `ValueError` on the first call, a wrong dtype trips an assertion in the pipeline, a missing class never appears in the predictions at all. The integration layer's job is to catch these before the model reaches production.

Silent failures live in the gap between syntactic and semantic correctness. Same dtype, same dimensions, but pixel range $[0, 255]$ instead of $[0, 1]$, or raw greyscale instead of binarised. The model returns a class label and a confidence score with no way to tell that the input did not come from the training distribution. This is the train/serve skew bug class, and the only fix is an explicit preprocessing step that mirrors training.

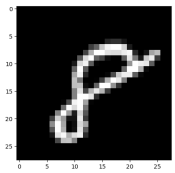

Figure 2: An MNIST sample: a handwritten digit on a 28×28 greyscale grid, the input shape every consumer of `digit_classifier_v1` must respect.

The MNIST model expects:

- a 28×28 greyscale image,
- pixel values normalised to $[0, 1]$,
- flattened to a 784-element vector.

The sklearn Pipeline bundles preprocessing and model into one serialised object that is itself the contents of `digit_classifier_v1.pkl`. `joblib.load` returns the whole Pipeline, preprocessing steps included, so loading the file is loading the preprocessing, so we reduce steps a maintainer might forget; adhering the contract the training script established.

Documentation: scikit-learn, joblib, NumPy.

The model card is where you write the contract down. A model card answers four questions any consumer of the model should know the answer to before they trust it: where did the data come from, what can the model do, what does it fail at, and what is it intended for. The contract that runtime code enforces and the card that humans read are two views of the same thing.

Model cards as a concept were proposed by Mitchell et al. in 2018 and are now standard at Google, HuggingFace, and increasingly across the industry. Model Cards (Google), HuggingFace Model Cards.

For PixelWise the v1 model card lives in the central `pixelwise-model` repository alongside the `.pkl` it documents. The card travels with each tagged release, so anyone who pulls `v1.0` also gets the card describing what `v1.0` can and cannot do:

```
# MODELCARD.md: digit_classifier_v1

## Training Data
MNIST digits 1-9, ~54k train / ~9k test, public domain.
Class 0 withheld intentionally.

## Capabilities
Predict handwritten digits 1-9.
Expected accuracy: ~92% (LogisticRegression baseline).

## Known Failures
6/9 confusion, 3/8 confusion, messy or rotated digits.

## Intended Use
28x28 canvas drawings.
Out of scope: photos, non-digit characters.
```

THE CARD DOES NOT REPLACE THE RUNTIME CHECKS built into the loader. Documentation tells humans what to expect; the assertion at module load tells the program what to expect. Both are needed. Documentation drifts; assertions cannot.

THE CARD IS A CONTRACT, like any other. When v2 is published, the card is updated alongside it; the version number, the training data section, and the capabilities all change in lockstep. Traceability across versions is the entire reason the card is a separate file.

Exercise: Scrutinize the Pipeline against the Card

VERIFY WHAT THE PIPELINE DOES rather than rebuild it. The preprocessing for the model is part of the artefact, not something you re-derive on the caller side; your job here is to confirm what is inside the `.pkl` matches what the canonical model card promises.

GET AN MNIST SAMPLE ONTO YOUR DEV MACHINE. The dataset is available through scikit-learn and downloads on first access. Activate the virtual environment and start an interactive `python3` session from the repository root, then run the snippet below at the »> prompt:

```
source .venv/bin/activate
python3
>>> from sklearn.datasets import fetch_openml
>>> X, y = fetch_openml(
...     "mnist_784", version=1,
...     return_X_y=True, as_frame=False,
...     parser="liac-arff",
... )
>>> sample = X[0].reshape(28, 28).astype("uint8")
>>> label = y[0]
```

You now have one ground-truth example to scrutinise the pipeline against.

WHY `parser="liac-arff"`. Since scikit-learn 1.2, `fetch_openml` defaults to `parser="auto"`, which prefers a pandas-based parser and raises if pandas is not installed. Naming the built-in liac-arff parser keeps the dependency footprint at what `requirements.txt` pins.

OPEN THE PIPELINE AND LOOK AT ITS PARTS. Still inside the same `python3` session, the Pipeline object exposes its named steps as a dictionary, and the classifier exposes the classes it knows:

```
>>> for name, step in model.named_steps.items():
...     print(name, type(step).__name__)
>>> print("classes:", model.classes_)
>>> print("n_features expected:", model.n_features_in_)
```

THE POINT IS VERIFICATION. You are not reproducing the model's preprocessing; you are confirming the artefact behaves the way the model card claims. A pipeline that contains an unexpected step, or is missing one you expected, is a contract violation you want to catch here, not later.

NOW RUN THE MODEL ON THE SAMPLE you fetched and compare against the ground-truth `label`:

```
>>> import numpy as np
>>> arr = (sample > 128).astype(float).reshape(1, -1)
>>> probs = model.predict_proba(arr)
>>> pred = model.classes_[probs.argmax()]
>>> print(f"pred={pred} true={label} conf={probs.max():.2f}")
```

Open the canonical model card at schutera/pixelwise-model and put it side by side with your introspection output. Answer each of these against both sources:

- Which preprocessing steps are inside the pipeline, and which are the caller's responsibility? Does the boundary match what the card claims is in scope?
- Does `model.classes_` match the class roster the card promises?
- Does `n_features_in_` match the input shape the card advertises?
- Does the accuracy you can reproduce on a handful of MNIST samples land near the figure the card cites?

A mismatch between what the pipeline contains and what the canonical card promises is exactly the kind of contract violation the integration layer exists to catch. The next exercise turns this manual check into a smoke test that runs on every commit.

Designing the Inference Interface

Before writing any code, design the contract. Three decisions propagate to every later block, the API, the database schema, the frontend, so they are worth naming up front: the class roster, the function shape, and the loading strategy.

A named class constant makes the valid output set explicit and independent of the model file. `CLASSES` lives in the code as a hard-coded list, not derived from `model.classes_` at import time, so the integration has its own opinion about what labels it speaks. The frontend, the database schema, and the API documentation all reference it. If the model is swapped for one trained on different classes, an assertion described below fires loudly at startup rather than returning silently wrong labels for the rest of the service's lifetime.

Batch-first design. What happens when 50 students submit a drawing in the same second? If the server processes each request individually, the model is called 50 times with a $(1, 784)$ array. If it batches them, the model is called once with a $(50, 784)$ array. sklearn's `predict_proba` is natively vectorised, so the cost is nearly identical. Design around batch; single-user is a special case.

Two functions, one batch-first. The public surface of `app/classifier.py` is:

```
classify_batch(images) -> list[dict]
classify(image)        -> dict
```

`classify_batch` takes an `(N, 28, 28)` `uint8` array and returns a list of `N` dictionaries, each with `prediction`, `confidence`, and `scores`. `classify` is a single-image convenience wrapper that delegates to `classify_batch` with `image[np.newaxis]`, so there is exactly one inference path. When a request queue is added later to amortise cost, the queue assembles a batch and the same function consumes it; no signature change.

Load the pipeline once at module level. Deserialising on every request kills throughput, so `joblib.load(os.getenv("MODEL_PATH"))` runs at import time and the resulting `Pipeline` stays in memory for the lifetime of the process. The path comes from `.env`, not a hard-coded string, so swapping the model in production is an env-var change plus a service restart, the same pattern you will see repeated in later blocks.

The startup assertion is the headline. Right after `joblib.load`, one line: `assert list(_pipeline.classes_) == CLASSES`. The assertion fires at startup if the model's classes disagree with the code constant. A swapped model with the wrong classes is caught the moment the service starts, not the next time a user gets a wrong digit. The next exercise has you trip it on purpose to feel the difference.

Exercise: The Classifier Module

Create the classifier module file on the `dev` VM, inside the PixelWise repository. The module belongs in the `app/` package, so create the directory if it is not there yet and open the file in nano:

```
cd ~/pixelwise
mkdir -p app
nano app/classifier.py
```

Paste the implementation below into the empty file, save with `Ctrl+O`, exit with `Ctrl+X`. It includes the class constant, module-level loading, the startup assertion, and the two functions discussed in the previous section:

```
import os
import joblib
import numpy as np
from dotenv import load_dotenv

load_dotenv()

CLASSES = ["1","2","3","4","5","6","7","8","9"]

_pipeline = joblib.load(os.getenv("MODEL_PATH"))
assert list(_pipeline.classes_) == CLASSES, \
    f"Model/CLASSES mismatch: {_pipeline.classes_}"

def classify_batch(images: np.ndarray) -> list[dict]:
    if images.ndim != 3 or images.shape[1:] != (28, 28):
        raise ValueError(
            f"Expected (N,28,28), got {images.shape}")
    arr = (images > 128).astype(float).reshape(
        len(images), -1)
    probs = _pipeline.predict_proba(arr)
    return [
        {"prediction": CLASSES[p.argmax()],
         "confidence": float(p.max()),
         "scores": dict(zip(CLASSES, p.tolist()))}
        for p in probs
    ]

def classify(image: np.ndarray) -> dict:
    return classify_batch(image[np.newaxis])[0]
```

Optional: pin pandas. `fetch_openml` pulls in `pandas` under the hood, and we have leaned on it implicitly more than once already. If you want it as an explicit, pinned dependency rather than a transitive one, install and refreeze:

```
pip install pandas
pip freeze > requirements.txt
git add requirements.txt
git commit -m "Pin pandas"
```

UPDATE `.env.example` AND `.env` at the repo root so the loader knows where to find the artefact. Open each in nano and append the line below, save, exit:

```
nano .env.example   # commit this; safe to share
nano .env           # local only; never commit
```

```
MODEL_PATH=models/digit_classifier_v1.pkl
```

WRITE `predict.py` AS A SMOKE TEST at the repo root, next to `setup-server.sh`, not inside `app/`: it is a script you run, not a module the application imports. Open it in nano:

```
nano predict.py
```

Paste the smoke test below, save, exit:

```
from app.classifier import classify_batch
from sklearn.datasets import fetch_openml
import numpy as np

X, y = fetch_openml("mnist_784", version=1,
    return_X_y=True, as_frame=False,
    parser="liac-arff")
images = X[:5].reshape(-1, 28, 28).astype(np.uint8)
truth  = y[:5]
results = classify_batch(images)
for r, t in zip(results, truth):
    print(f"Pred: {r['prediction']} "
          f"(conf {r['confidence']:.2f})  True: {t}")
```

RUN THE SMOKE TEST ON DEV from the repo root with the virtual environment activated, then commit:

```
source .venv/bin/activate
python predict.py
git add app/classifier.py predict.py .env.example
git commit -m "Add classifier module pinned to model v1.0"
git push
```

TAG THE RELEASE AS `v0.3`. The end of every block is a tagged, runnable state. Mark it now so future catch-up exercises can reset back here:

```
git tag -a v0.3 -m "End of Block 4: classifier module"
git push origin v0.3
```

`__pycache__` IN YOUR COMMIT? Running `python predict.py` produces compiled bytecode in `app/__pycache__/`. If those `.pyc` files made it into the commit, extend `.gitignore` from Block 3 with the entries below so they stay out of every future commit:

```
__pycache__/
*.pyc
```

Then untrack what was already added and rewrite the previous commit so the bytecode never reaches the remote:

```
git rm -r --cached app/__pycache__
git add .gitignore
git commit --amend --no-edit
git push --force-with-lease
```

Only `--force-with-lease` this commit if you pushed it once and nobody else has pulled yet. Otherwise create a fresh follow-up commit.

The repository is now at the state `v0.3` marks: a working classifier, a dev-side smoke test, and an updated configuration contract pinned to the model release. The `.pkl` stays on each developer's machine, the artefact path is in `.env`, and the rest of the team can reproduce the environment from the code in Git.

Optional: Add the Missing 0

THE DIGIT CLASS "0" WAS INTENTIONALLY EXCLUDED FROM V1. With v1 integrated and dev-verified, you can publish your own v2. Fork the course model repository so you have a place to push:

```
gh repo fork schutera/pixelwise-model --clone --remote
cd pixelwise-model
```

The fork already contains `train.py`, `digit_classifier_v1.pkl`, and the v1 `MODELCARD.md` from upstream. Add `"0"` to the class list, include MNIST's class-0 samples in the training split, re-run `python train.py`, and save the new artefact as `digit_classifier_v2.pkl`.

WHY FORK INSTEAD OF BRANCHING UPSTREAM? Forking is the standard pattern for publishing a derivative of someone else's repository. You get write access to your fork, the upstream stays canonical, and a pull request is the natural way to upstream improvements later.

NOW WRITE A V2 MODEL CARD. This is the first time card authoring is load-bearing in the course: a consumer who pulls v2 will read this file to learn what changed. Rewrite `MODELCARD.md` so it documents v2 specifically:

- training data section now reflects the full ten-class MNIST split,
- capabilities list all ten digits and the new accuracy you measured,
- known failures section reflects what your retrained model actually struggles with (rotated digits, 4/9 confusion, whatever the confusion matrix says),
- intended use is the same as v1 unless you changed it.

Commit artefact, script changes, and card together, and cut a fresh tag:

```
git add digit_classifier_v2.pkl train.py MODELCARD.md
git commit -m "v2: adds class 0"
git tag -a v2.0 -m "Adds class 0"
git push --follow-tags
```

On the PixelWise side, point `.env` at your fork and the new tag by setting `MODEL_REPO=https://github.com/<you>/pixelwise-model.git` and `MODEL_VERSION=v2.0`, then re-run `bash setup-server.sh` to pull the v2 artefact and card into `models/`. The assertion will break the moment v2 is loaded against v1's `CLASSES`; fix it by updating the constant.

This challenge sets up a thread we will revisit later, when feature flags decide which model version is served and analytics compare their performance side by side. No data collection is needed; MNIST has the 0s already.

A NOTE ON WHAT COMES NEXT. `app/classifier.py` works on dev, but it is trapped in a Python script.

Self-Reflection and Recap

Self-Reflection questions to guide your thoughts during and after the exercises:

- What is train/serve skew, and why is it the most common production ML defect?
- Why do we load the model once at module level instead of on every request?
- What does the startup assertion protect against, and what would happen without it?
- Why does PixelWise pin a tagged model release rather than vendoring the `.pkl` into the repo?
- What does the canonical model card promise, and which of its claims did you verify by introspecting the pipeline?
- How would you detect at runtime that the model is missing a class it should predict?
- Why is batch-first design the right default, even when most requests carry a single image?

Recap of key concepts:

- A `.pkl` file is a build artefact, not source code; never commit it to Git.
- Train/serve skew is silent by default; the startup assertion is the loudest defence.
- Batch-first design scales from one user to N concurrent users with no code changes.
- Module-level loading keeps the model in memory for the lifetime of the process.
- The model card travels with the tagged release in `pixelwise-model`; integration code reads it as part of contract verification.
- `predict.py` proves end-to-end correctness before any API exists.

Milestone. `app/classifier.py` exposes a batch-first inference interface that scales from one student to N concurrent users with no code changes. `predict.py` proves it works end to end on real MNIST samples. The canonical `MODELCARD.md` in `pixelwise-model` documents what the model is and is not, and the integration verifies against it.

Security thread. Never load a `.pkl` from an untrusted source; `joblib` and `pickle` deserialisation execute arbitrary code. `MODEL_PATH` comes from `.env`, never hard-coded, so swapping the model in production requires only an env-var change, not a code change. Training data and model artefacts are not committed to Git.

Teaser. The classifier is callable from Python, but who, outside your terminal, can reach it?

We deliberately stop short of `prod`. The classifier works on `dev`, but `prod` has not been updated, on purpose. There is no service on `prod` yet that would call `classify_batch`, no HTTP endpoint, no managed background process, no entry point at all. Pulling the new code there now would just add an unused `app/classifier.py` to the production checkout. Deployment to `prod` happens once there is something to deploy.