# *Databases & SQL*

*2026-05-10 · alert currant Gans*

In Comparison, Dumbledore's Pensieve is a joke.

## *The Why*

We have a service that classifies, but it does not remember anything. The moment the response leaves the server, the prediction is gone. This makes it hard to track the model's performance over time, compare versions, or build any type of features and functionality that revolve around past predictions.

We need to wire the API onto a database. This block is the first that adds persistent state. Every later block depends on the database schema you design today.

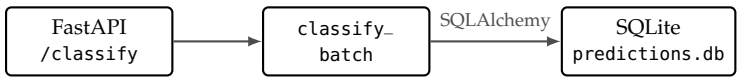


Figure 1: The persistence flow. `POST /classify` computes a prediction with `classify_batch`, the handler maps the result through the `Prediction` SQLAlchemy model, and a row lands in the `predictions` table inside the `predictions.db` SQLite file before the response goes out.

Why persist at all? A response that vanishes the moment it is sent is a response that cannot be counted, compared, or queried. A database makes the prediction permanent: Every drawing becomes a row, the class label becomes a column, the confidence becomes another, and the model version becomes a third.

Beyond this course, the same table is where production teams build retraining datasets, let users correct wrong predictions for labelled data, and detect distribution drift. We do not do any of those here, but it is trivial to see how every one of them starts with having the database.

SQLAlchemy. The Object-Relational Mapper, abbreviated ORM, that bridges Python objects and relational tables. You write Python classes; SQLAlchemy generates the Structured Query Language, abbreviated SQL, behind them, so the handler talks to the database without composing query strings by hand.
https://www.sqlalchemy.org/

*Hands On Experience*

CONSIDER THIS SCENARIO: A teammate asks how confident the model has been on average across the last hundred predictions. You realise nobody can answer that question. The API returned the confidence to every caller, but nothing recorded it. The only way back to those numbers is to ask the users to draw their digits again, which is not an option. In an increasingly data-driven world, data defines your product, your moat, and your competitive advantage. Not having the data is like throwing away your most valuable asset, and until someone comes up with a solution for time-travel, it's a non-recoverable loss.

THE SCHEMA IS A CONTRACT, the same way the API contract is a contract. Get it right and every later question, distribution per digit, average confidence per model version, error rate over time, becomes a query. Get it wrong and you find out later, with a hundred thousand rows already written, that the column you need does not exist. We keep the schema small and deliberate from the start, so the columns worth having are there before any rows are.

DESIGNING SCHEMAS is its own discipline. We start with a small, deliberate schema, four columns, that covers everything later blocks need to read back. The point is to feel the cycle of design and deploy on a real database, not to ship a kitchen-sink table on day one.

THIS SIXTH BLOCK introduces persistence for PixelWise. By the end you will have

- written your first SQL queries against a SQLite database,
- modelled the `predictions` table as a SQLAlchemy class,
- wired `POST /classify` so every prediction becomes a row,
- replaced the `GET /results` stub with a real query,
- pointed the application at the database through `.env`,
- and, optionally, run your first schema migration with Alembic.

## *Optional Exercise: Catch Up to `v0.4`*

CATCHING UP VIA TAG. This catch-up extends the one in Block 5. If you have not done that one, do it first: It lands you at `v0.3`, the end-of-Block 4 state. The exercise here picks up from there and rewinds to `v0.4`, the end-of-Block 5 state, with the FastAPI service exposing `POST /classify` and `GET /health`, API key authentication, basic rate limiting, and a `systemd`-supervised `uvicorn` running on `prod`.

BOTH VMS THIS TIME. Block 5 ended with the service actually running on `prod`, so unlike the previous catch-up, this one brings both machines up to `v0.4`. The Git-tracked changes since `v0.3` include `app/main.py`, `deploy/pixelwise.service`, an extended `setup-server.sh` that installs the systemd unit on `prod`, an updated `requirements.txt` adding `fastapi`, `uvicorn`, `slowapi`, and `pydantic`, and a refreshed `.env.example` with `SECRET_API_KEY`. The local `.env` on `prod` needs the new key written by hand; `setup-server.sh` then pulls the model, copies the unit, enables, and restarts the service in one shot.

RUN THE CATCH-UP ON `dev`. Rewind the repository to `v0.4`, refresh `.env` from the updated example and edit it to set `SECRET_API_KEY`, reinstall pinned dependencies if your virtual environment was wiped, then run `setup-server.sh` to pull the model artefact. The exact commands are in the margin.

RUN THE CATCH-UP ON `prod`. Same sequence on prod, where `setup-server.sh` also copies the systemd unit into place and restarts the service. After the script finishes, verify the API is reachable from `dev`.

YOU ARE NOW at the state where Block 5 ended, with both machines aligned, ready to start Block 6.

TAG MAP. `v0.4` marks the end of Block 5, `v0.5` will mark the end of this block. Browse the full set at `https://github.com/schutera/pixelwise/tags`.

SEQUENCE ON `dev`.

```
cd ~/pixelwise
git fetch origin --tags
git reset --hard v0.4
cp .env.example .env
nano .env
source .venv/bin/activate
pip install -r requirements.txt
bash setup-server.sh
```

Edit `.env` to set `SECRET_API_KEY` to any string of your choosing; `openssl rand -hex 32` prints a clean random value if you want one. If you already have a `.env` from earlier work, skip the `cp` and just edit the file in place. If `.venv/` is missing, run `python3 -m venv .venv` before the `source` step.

SEQUENCE ON `prod`.

```
ssh produser@192.168.56.11
cd /opt/pixelwise
git fetch origin --tags
git reset --hard v0.4
cp .env.example .env
nano .env
source .venv/bin/activate
pip install -r requirements.txt
bash setup-server.sh
```

Use the same `SECRET_API_KEY` value as on `dev`. If you already have a `.env` on `prod`, skip the `cp` and edit the file in place. Then from `dev`: `curl http://192.168.56.11:8000/health` should return `{"status":"ok"...}`.

## *Relational Databases and SQL*

A RELATIONAL DATABASE STORES DATA IN TABLES. Each table has a fixed schema: a list of named columns, each with a type. Each row is a single record. The schema is the contract that every row obeys; the database refuses inserts that violate it.

VOCABULARY TO KEEP STRAIGHT. A *table* is a named set of rows; a *row* is one record; a *column* is a named field with a type; a *primary key* is the column (or set of columns) that uniquely identifies each row. Most tables use an auto-incrementing integer or a UUID as their key.

SQL IS THE LANGUAGE you use to talk to a relational database. Four operations cover almost everything:

```
-- Insert a row
INSERT INTO predictions (prediction, confidence,
                         model_version)
VALUES ('7', 0.94, 'v1');

-- Read rows
SELECT prediction, confidence FROM predictions
    WHERE model_version = 'v1'
    ORDER BY created_at DESC
    LIMIT 10;

-- Update a row
UPDATE predictions SET confidence = 0.95
    WHERE id = 42;

-- Delete a row
DELETE FROM predictions WHERE id = 42;
```

THE FOUR OPERATIONS ARE WHAT EVERY WEB APPLICATION DOES TO ITS DATABASE ALL DAY LONG: write a row when something happens, read rows back when somebody asks, update a row when state changes, delete one when it should be gone. Every higher-level pattern, an ORM call, a migration, an analytics query, reduces to a sequence of these.

WHAT YOU WILL MEET LATER. `JOIN` combines rows from multiple tables; `GROUP BY`, `COUNT`, and `AVG` aggregate; `ORDER BY` and `LIMIT` shape the output; `CREATE INDEX` speeds up queries that scan large tables. The full treatment belongs in a dedicated database course; the four operations above are enough to get PixelWise running.

## *SQLite for this Course*

SQLITE IS A DATABASE IN A SINGLE FILE. There is no server to install, no users, no passwords, no network configuration. The application opens the file, writes rows, closes the file. For a course where one student runs one service on one VM, this is exactly the right shape.

WHAT YOU GIVE UP. SQLite has no user authentication and limited concurrency: a single writer blocks all readers. For a multi-user production web service hit by hundreds of clients per second, the file-lock model becomes the bottleneck, and a separate database process becomes the right call. For PixelWise with one user clicking through Swagger, the file is more than enough.

PostgreSQL is the production next step. It runs as a separate process, accepts network connections, has real users with real permissions, and supports concurrent reads and writes without serialising the whole file. The shift from SQLite to PostgreSQL is mostly one line of `DATABASE_URL` plus the operational work of installing the server, creating a role, and managing the password. We stop one step short of that here; everything else in the chapter, the SQLAlchemy model, the migrations, the queries, looks the same against either backend.

SQLite: https://www.sqlite.org/
SQLite CLI docs: https://www.sqlite.org/cli.html
PostgreSQL: https://www.postgresql.org/

Setting up the database is a one-liner: pick a file path, and SQLAlchemy creates the file on first connect. The application user is the OS user that owns the file; the privilege model collapses to file-system permissions. `sqlite3` is the operator's interface, the same shape of tool as `psql` for PostgreSQL or `systemctl` from Block 5: a small REPL you drop into when you want to look at the data behind the application.

Where the file lives. On `prod` the database file sits inside `/opt/pixelwise/`, owned by `produser`. The same OS-level least-privilege story from Block 1 applies: the application user owns the data file, no one else needs to read it, and a backup is one `cp` of a single file.

## *Exercise: SQLite on the Server*

EXERCISES are where the theory becomes muscle memory. SQLite needs almost no setup, so this first exercise is a five-minute warm-up: pick a file path, make sure the directory exists, verify with the `sqlite3` CLI. The substance comes in the next two exercises where SQLAlchemy and the API land on top.

INSTALL THE `sqlite3` CLI ON `prod`. The SQLite database engine itself ships as a Python C extension and needs no separate install. The `sqlite3` command-line tool is a small REPL useful for poking at the file by hand:

```
ssh produser@192.168.56.11
sudo apt update
sudo apt install -y sqlite3
sqlite3 --version
```

PICK A PATH FOR THE DATABASE FILE. The application user owns it, so it lives inside the application directory. On `prod` that is `/opt/pixelwise/`; the file will be created on first connect:

```
# on prod
ls -la /opt/pixelwise/
# /opt/pixelwise/pixelwise.db will appear here
# the first time the API or init_db.py runs
```

THE CONNECTION STRING is a single URL: `sqlite:////absolute/path/to/file.db`. Note the four slashes after the colon: three for the URL scheme, one for the absolute path. A relative path uses three slashes (`sqlite:///./pixelwise.db`). The application reads this from `.env`, so the same code works on `dev` and `prod` with different paths.

ADD `DATABASE_URL` to both VMs' `.env` and to the committed `.env.example`. The values differ per host, the key does not:

```
# /opt/pixelwise/.env on prod
DATABASE_URL=sqlite:////opt/pixelwise/pixelwise.db

# ~/pixelwise/.env on dev
DATABASE_URL=sqlite:///./pixelwise.db

# .env.example, committed on dev
DATABASE_URL=sqlite:///./pixelwise.db
```

VERIFY THE CLI WORKS against an empty database. The file does not exist yet, so `sqlite3` creates it on the spot and drops you into a prompt; `.tables` lists the empty schema and `.quit` exits:

```
sqlite3 /opt/pixelwise/pixelwise.db
sqlite> .tables
sqlite> .quit
```

The file now exists on disk with zero tables; the next exercise has SQLAlchemy create the `predictions` table inside it.

## *SQLAlchemy: Tables as Python Classes*

SQLAlchemy is a Python library that maps database tables to Python classes. You define a `Prediction` class with fields like `prediction`, `model_version`, and `created_at`; SQLAlchemy generates the SQL to create the table, insert rows, and query data. This is called an *ORM*, an object-relational mapper, and is standard in Python web development.

SQLAlchemy: `https://www.sqlalchemy.org/`
SQLAlchemy Tutorial: `https://docs.sqlalchemy.org/en/20/tutorial/`

The model class declares the columns and their types in plain Python. A `Prediction` class inherits from a SQLAlchemy declarative base, sets `__tablename__` to `predictions`, and lists each column as a class attribute: An auto-incrementing integer primary key, a non-null string for the predicted class, a non-null string for the model version, and a timestamp that defaults to the current UTC time. The exercise below has the full `app/models.py` the API will import.

An engine and a session are the runtime objects that connect the model to the database. `create_engine` reads the `DATABASE_URL` from the environment and opens a pool of connections; `sessionmaker` returns a `SessionLocal` factory the handlers will call once per request; `Base.metadata.create_all(engine)` creates the table on first run. The exercise wires these three lines into a small initialisation script the team runs on `prod` before the first deploy.

Why an ORM? Two reasons. The first is ergonomic: writing Python is faster than writing string SQL, and the editor knows the field names. The second is safety: SQLAlchemy uses parameterised queries by default, which means user input never ends up concatenated into a SQL statement. SQL injection becomes a footgun the framework refuses to hand you.

Inserting a row from inside the API handler is now a few lines: Open a session from `SessionLocal`, construct a `Prediction` with the fields from the classifier result, add it to the session, and commit. The handler still calls `classify_batch` from Block 4 and returns `ClassifyResponse` from Block 5; the only addition is one round-trip to the database before the response leaves the server. The exercise below wires it up inside `POST /classify`.

The connection string is a single URL the engine parses: `sqlite:////opt/pixelwise/pixelwise.db` on `prod`, the matching relative form on `dev`. It encodes everything the driver needs to reach the database, and it lives in `.env` so the same code works against different files on each host without code changes.

## *Exercise: The `Prediction` Model*

TURN THE THEORY ABOVE INTO CODE. On `dev`, install SQLAlchemy into the virtual environment from the repository root. SQLite ships with the Python standard library, so no separate driver is needed:

```
cd ~/pixelwise
source .venv/bin/activate
pip install sqlalchemy
pip freeze > requirements.txt
```

WRITE `app/models.py` with the schema we will want to read back later: the class label, the confidence, and the model version, all stamped with a timestamp. From the repository root on `dev`:

```
cd ~/pixelwise
nano app/models.py
```

Paste the following, save with `Ctrl+O`, exit with `Ctrl+X`:

```
from sqlalchemy import (Column, Integer, String,
                        Float, DateTime)
from sqlalchemy.orm import declarative_base
from datetime import datetime

Base = declarative_base()

class Prediction(Base):
    __tablename__ = "predictions"
    id = Column(Integer, primary_key=True)
    prediction = Column(String, nullable=False)
    confidence = Column(Float, nullable=False)
    model_version = Column(String, nullable=False)
    created_at = Column(DateTime,
                        default=datetime.utcnow)
```

CREATE THE TABLE ONCE from a small initialisation script saved as `init_db.py` at the repository root on `dev`. The script reads `DATABASE_URL` you set in the previous exercise and calls `Base.metadata.create_all` to materialise every model class into a table:

```
cd ~/pixelwise
nano init_db.py
```

Paste the following:

WHY THESE FOUR COLUMNS. The class label and confidence are what the model returns; the model version lets you compare `v1` against future retrains; the timestamp orders rows by recency. Any later monitoring or analytics reads all four directly from this table, so a complete schema today saves a migration later.

```
from sqlalchemy import create_engine
from app.models import Base
import os
from dotenv import load_dotenv

load_dotenv()
engine = create_engine(os.getenv("DATABASE_URL"))
Base.metadata.create_all(engine)
```

Commit and push from `dev`, then run the script on `prod` where the database lives:

```
# on prod, after git pull
cd /opt/pixelwise
source .venv/bin/activate
python init_db.py
```

Confirm with `sqlite3 /opt/pixelwise/pixelwise.db` that the `.tables` command now lists `predictions`:

```
sqlite3 /opt/pixelwise/pixelwise.db
sqlite> .tables
predictions
sqlite> .schema predictions
sqlite> .quit
```

## *Exercise: Wiring the API to the Database*

UPDATE `app/main.py` so `POST /classify` writes a row before returning the response. Reopen the file from the repository root on `dev`:

```
cd ~/pixelwise
nano app/main.py
```

The handler from Block 5 stays the same shape; one new dependency creates a database session per request:

```
from app.models import Prediction, SessionLocal

@app.post("/classify",
          response_model=ClassifyResponse,
          dependencies=[Depends(verify_api_key)])
def classify(req: ClassifyRequest):
    arr = np.array(req.pixels,
                   dtype=np.uint8)[np.newaxis]
    result = classify_batch(arr)[0]

    db = SessionLocal()
    db.add(Prediction(
        prediction=result["prediction"],
        confidence=result["confidence"],
        model_version="v1"))
    db.commit()
    db.close()

    return result
```

ONE SESSION PER REQUEST. A session is the unit of database work. Open it at the start of the handler, close it at the end. The framework patterns FastAPI documents (`Depends` for sessions) do this more elegantly; the manual version above is the simplest one that shows what is happening.

REPLACE THE `GET /results` STUB in the same `app/main.py` with a real query:

```
@app.get("/results")
def results():
    db = SessionLocal()
    rows = (db.query(Prediction)
              .order_by(Prediction.created_at.desc())
              .limit(20).all())
    db.close()
    return {"results": [
        {"id": r.id,
```

```
         "prediction": r.prediction,
         "confidence": r.confidence,
         "model_version": r.model_version,
         "created_at": r.created_at.isoformat()}
        for r in rows]}
```

Restart the service and feed it data. There is no browser canvas yet, so trigger predictions the same way as in Block 5: through Swagger at `http://192.168.56.11:8000/docs`, with the API key in the `x_api_key` field and a zero-filled 28-by-28 body in the request body. After a handful of calls, `curl http://192.168.56.11:8000/results` from `dev` returns the rows you just wrote, each with its class, confidence, model version, and timestamp. The stub from Block 5 is now complete; the system stores what it predicts.

Commit and tag as `v0.5`. The repository now matches the end-of-Block 6 state that later catch-up exercises rewind to. From the repository root on `dev`:

```
cd ~/pixelwise
git add app/main.py app/models.py init_db.py \
    requirements.txt .env.example
git commit -m "Add SQLite persistence via SQLAlchemy"
git push
```

Why stop here. The schema covers everything we will want to read back later, and the service stores what it predicts. Anything that comes after can sit on top of this state without needing further database work. The optional Alembic section that follows shows how schemas evolve once real rows already exist; nothing in the core path depends on it.

## *Optional: Schema Migrations with Alembic*

Schemas always change. You design the table, deploy it, store 500 predictions. Then you realise you forgot a column, or want to capture a new fact about each row, like a human annotator's notes on whether the prediction was correct. You cannot drop the table and recreate it; those 500 rows are real data. You need a tool that alters the live schema without losing them.

Alembic is the migration tool for SQLAlchemy. It generates migration scripts from changes you made to the model, applies them in order, and tracks which migrations have run on which database. The cycle is three commands: `alembic init` once per repository to lay down the migrations folder, `alembic revision --autogenerate` after every model edit to draft the migration, and `alembic upgrade head` to apply it. The exercise below runs the full cycle to add a `notes` column for human annotations on individual predictions.

In a real system with millions of rows and live traffic, you cannot just drop and recreate. Migrations are how schemas evolve safely, in production, with the application running. This topic is covered in much more depth in dedicated database and backend courses; here we run one migration end to end so the cycle of edit-model, generate-migration, apply-migration is concrete.

Optional. Nothing in the core path depends on this section. The schema you shipped at `v0.5` already carries every column later parts of the course will want to read back. Run this section if you want to feel the migration cycle on a live database; skip it otherwise.

Alembic: `https://alembic.sqlalchemy.org/`

The metaphor. A migration is a recipe that describes a change to the schema and how to roll it back. The migration tool keeps an internal log of which migrations have run, applies new ones in order, and refuses to run the same one twice. The result is that the schema in the database always matches the migrations folder in Git.

The autogenerate is a draft, not a final answer. Alembic compares the model in code against the schema in the database and guesses what changed. For straightforward additions it gets it right; for renames or splits it needs a human to read the generated script and adjust it. Always read the migration before you run it.

Sidenote: Redis. Redis is an in-memory key-value store with sub-millisecond reads. It is the right tool for caching, session storage, and rate-limit counters. For PixelWise, every drawing is unique pixel data, so the cache hit rate would be effectively zero; we mention it so you recognise it later. Redis: `https://redis.io/`

## *Optional Exercise: First Migration with Alembic*

INITIALISE ALEMBIC IN THE REPO. With the migration cycle from the theory above in mind, configure the tool to use the same engine as the application. `alembic init` writes its scaffold into the current directory, so run everything from the repository root on `dev`:

```
cd ~/pixelwise
source .venv/bin/activate
pip install alembic
pip freeze > requirements.txt
alembic init alembic
```

Edit `alembic.ini` so `sqlalchemy.url` reads from `.env`, and edit `alembic/env.py` to import `Base` from `app.models` so autogenerate sees your tables.

READ THE GENERATED CONFIG. `alembic init alembic` writes a folder of starter files. Skim them once before you change anything; the structure is small and the mental model becomes obvious by reading the templates.

ADD A NEW COLUMN to the model. Reopen `app/models.py` from `~/pixelwise` on `dev` and add a nullable `notes` field for free-form human annotation of individual predictions:

```
cd ~/pixelwise
nano app/models.py
```

```
class Prediction(Base):
    __tablename__ = "predictions"
    id = Column(Integer, primary_key=True)
    prediction = Column(String, nullable=False)
    confidence = Column(Float, nullable=False)
    model_version = Column(String, nullable=False)
    notes = Column(String)                # NEW
    created_at = Column(DateTime,
                        default=datetime.utcnow)
```

GENERATE THE MIGRATION from the repository root on `dev`, commit the new file under `alembic/versions/`, and push:

```
cd ~/pixelwise
alembic revision --autogenerate \
    -m "add notes column"
```

Apply the migration on `prod`, where the database lives:

```
# on prod, after git pull
cd /opt/pixelwise
source .venv/bin/activate
alembic upgrade head
```

READ THE MIGRATION BEFORE YOU RUN IT. The generated script lives in `alembic/versions/`. Open it, confirm the only operation is `add_column("notes")`, and only then run `upgrade head`. A wrong migration on a populated table is much harder to undo than to prevent.

Verify on `prod` with `sqlite3`:

```
sqlite3 /opt/pixelwise/pixelwise.db \
  "SELECT prediction, confidence, notes
   FROM predictions
   ORDER BY created_at DESC LIMIT 5;"
```

Every row shows `NULL` for `notes`; the column exists but no code writes to it yet. That gap is the visible signature of a migration: the schema changes everywhere at once, but values only appear once the application code is extended to fill them in.

Commit the Alembic artefacts on the side without disturbing the `v0.5` tag from the previous exercise:

```
cd ~/pixelwise
git add app/models.py alembic/ alembic.ini \
    requirements.txt
git commit -m "Add optional notes column via Alembic"
git push
```

A note on what comes next. The system is reachable on the network and remembers what it does, but nobody outside the SSH tunnel can use it. Making PixelWise accessible from a browser at a real URL is a thread we will revisit later in the course.

## *Self-Reflection and Recap*

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

- Why do we store predictions in a database rather than just returning them in the API response?
- What does SQLite buy you for a course like this, and where would the same choice break down in production?
- How does the ORM make SQL injection harder, and where does the protection break down?
- Why does it pay to think the schema through up front rather than adding columns later?
- If you did the optional Alembic section: what is the difference between dropping and recreating a table versus running a migration, and why did the new column show `NULL` on every existing row?

Recap of key concepts:

- Relational databases store data in typed tables; SQL is the language that queries and changes them.
- SQLite is the database for this course: a single file, no server, no users, no passwords. PostgreSQL is the production next step we stop short of here.
- SQLAlchemy maps tables to Python classes and uses parameterised queries by default, so the model code looks the same against either backend.
- `DATABASE_URL` lives in `.env` so each host can point at a different file without code changes.
- Optionally, Alembic generates and applies migrations so schemas can evolve without losing rows.

Bridge. So far, PixelWise has a memory. Every prediction is written to the SQLite database with its confidence and model version, `GET /results` reads them back, and the repository is tagged `v0.5`. But the only client that has ever touched this service is `curl` on the command line, holding an API key. The next chapter puts a browser in front of all of this: Nginx as a reverse proxy on ports 80 and 443, a small frontend that lets a user draw a digit, and HTTPS so the API key no longer travels in plaintext. What we built behind a terminal we now point at the open web.

Security thread. `DATABASE_URL` lives in `.env`, never in code, so a leaked repository does not leak the file path the application uses. The SQLite file itself is protected by ordinary file-system permissions: `produser` owns it, nobody else reads it. SQLAlchemy's parameterised queries prevent SQL injection by default; never build queries with string formatting.

Teaser. The service remembers, but only `curl` with the API key can talk to it. What does it take to put this in front of a real browser?