Dependencies & Structure

2026-05-09 · cheerful nectarine Karpfen

Works on my machine.

The Why

Block 2 left us with a PixelWise repository on GitHub and a server that has Git and Python installed. Both machines share the same history, both can pull from `origin`, and the workflow that moves code between them is in place. What is still missing is the layer beneath the code: the system packages, the language runtime, the libraries, and the secrets your application reads (installed or provisioned by `setup-server.sh` and recorded in `.env`/`.env.example`).

Kaltstart Capabilities, Your colleague clones the repository, runs the code, and immediately hits an `Error`. You installed a package with a specific version months ago and forgot. They did not. The remainder of the day is spent debugging an artefact of the install history rather than contributing real valeu to the program itself. At the end of this third block we want any fresh VM to go from empty to ready to code with a small, fixed sequence of commands. The two halves of that sequence are the load-bearing idea of the chapter: system packages on one side, Python packages on the other, version controlled in different files.

PixelWise now lives on GitHub and on both machines, but the repository contains only `README.md` and `.gitignore`. Before we can write application code, we need a reproducible way to install everything the project depends on, so that `dev` and `prod` stay in sync without human memory. Use `setup-server.sh` to provision system packages and the runtime, and keep secrets and runtime configuration in `.env.example` (copied to `.env`).

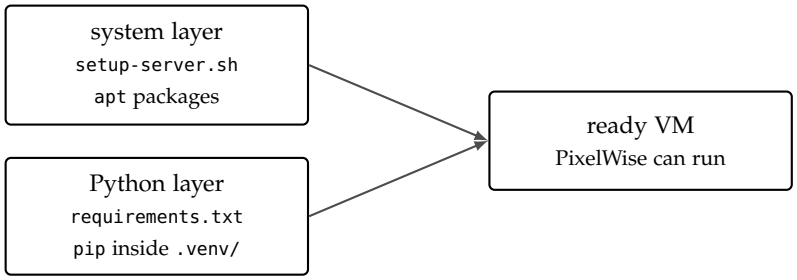

Figure 1: Two layers, two files. `setup-server.sh` provisions the operating system; `requirements.txt` provisions Python inside an isolated virtual environment. Together they reconstruct the environment from scratch on any fresh VM. System packages, the things you install with `apt`, are captured in a shell script called `setup-server.sh`. Python packages, the libraries your application imports, are captured in `requirements.txt` and installed inside an isolated virtual environment. The two files are committed to Git, run on either machine, and produce identical environments without any further intervention.

Hands On Experience

Consider this scenario: you hand your PixelWise repository to a classmate at the end of the day. They clone it, type `python3 train.py` without running the provisioning script `setup-server.sh`, and immediately hit `ModuleNotFoundError: No module named 'sklearn'`. You tell them to install scikit-learn. They do, but they get version 2.0 while you wrote the code against 1.4. The training script crashes with a different error. By the time the original problem is fixed, a few hours are gone and neither of you has touched the actual code nor added any value.

Reproducible projects do not require remembering, asking, or guessing. The exact set of packages, the exact versions, the exact configuration contract, are written down once, committed to Git, and replayed on any machine that has the script and the file.

The two-command promise. A well-managed project lets any new developer go from clone to running code with two commands: create the virtual environment and install from a pinned requirements file. No guessing, no mails or messages asking "which version of numpy do I need?" This block is what makes that promise hold.

This third block introduces dependency management and project structure for PixelWise. By the end you will have

- captured system-level setup in a reproducible shell script,
- created and activated a Python virtual environment,
- installed packages with `pip` and pinned them in `requirements.txt`,
- separated secrets from code using `.env` and `.env.example`,
- scaffolded the PixelWise project directory,
- and audited your dependencies for known security vulnerabilities.

Outside the classroom, the same files are how cloud deployment works in practice. A continuous-integration runner clones the repository, executes the setup script (for example, `setup-server.sh`), installs from `requirements.txt`, and runs the tests. The reproducibility you build for two VMs scales without modification to a hosted server, a colleague's laptop, or a CI runner you have never seen.

Optional Exercise: Catch Up to `v0.1`

CATCHING UP VIA THE REFERENCE TAG. From this block onwards, the GitHub repository is the safety net. The course maintains a public reference repository at `https://github.com/schutera/pixelwise` with a tag at the end of every block. `v0.1` marks the end of Block 2; `v0.2` will mark the end of this block. If you joined late, missed Block 2, or your repository has drifted from a known good state, reset to the previous tag rather than redoing every prior exercise.

TAG MAP. `v0.1` is the end of Block 2, `v0.2` is the end of this block, `v0.3` the end of Block 4, and so on. Every tag is a complete, runnable state of PixelWise. Browse them at `https://github.com/schutera/pixelwise/tags`.

THE MODEL: pull from `schutera/pixelwise`, push to your own `<you>/pixelwise`. The reference repository is read-only for you; the place you commit to is your personal copy on GitHub. The steps below assume nothing from Block 2 is in place, so the exercise is fully self-contained.

CREATE AN EMPTY `pixelwise` REPOSITORY ON GITHUB under your own account. Log in to GitHub, click New repository, name it `pixelwise`, and leave it empty, no README, no `.gitignore`, no licence, since the push step below will populate it. Without this repository the `git remote set-url` URL points at a path GitHub cannot find, and the first push reports `does not appear to be a git repository`.

RESTORE `B1-complete` ON `dev` so you start from the same baseline as everyone else, with Ubuntu installed and the user account in place but nothing on top.

INSTALL GIT, CONFIGURE YOUR IDENTITY, AND GENERATE A SEPARATE SSH KEY FOR GITHUB on the fresh VM. `B1-complete` predates Block 2, so the new VM has no `git` binary and no Git config. It does already have `~/.ssh/id_ed25519` and `~/.ssh/id_ed25519.pub` from Block 1, the key pair you use to log in to `prod`. The `-f` flag below writes the new key to a distinct path, so the existing prod-login key files stay byte-for-byte intact on disk and your password-less SSH into `prod` keeps working:

```
sudo apt update && sudo apt install -y git
git config --global user.name  "Your Name"
git config --global user.email "your@email.com"
ssh-keygen -t ed25519 -C "your@email.com" \
    -f ~/.ssh/id_ed25519_github
ls -l ~/.ssh/id_ed25519*
cat ~/.ssh/id_ed25519_github.pub
```

TELL SSH WHICH KEY TO USE FOR GITHUB.COM. With two keys in `~/.ssh/`, SSH may offer the prod-login key first and GitHub authenticates you as the wrong account, or fails outright. Worse, if `ssh-agent` has already cached the prod-login key, it gets offered before any `IdentityFile` on disk, and the symptom is a misleading `git@github.com:<you>/pixelwise.git does not appear to be a git repository`. Open (or create) `~/.ssh/config` in `nano` and add an entry that pins the GitHub key to github.com and bypasses the agent for that host:

```
nano ~/.ssh/config
```

Paste the following block at the end of the file, save with `Ctrl+O`, then exit with `Ctrl+X`:

```
Host github.com
    HostName github.com
    User git
    IdentityFile ~/.ssh/id_ed25519_github
    IdentitiesOnly yes
    IdentityAgent none
```

LOCK DOWN THE FILE PERMISSIONS. SSH refuses to read `config` if it is world- or group-readable, so tighten it to the owner only:

```
chmod 600 ~/.ssh/config
```

RUN THE VERIFICATION HANDSHAKE. The first SSH connection to GitHub asks you to confirm GitHub's host fingerprint and writes it into `~/.ssh/known_hosts`:

```
ssh -T git@github.com
```

Type `yes` when prompted to trust the fingerprint. GitHub then answers `Hi <you>! You've successfully authenticated, but GitHub does not provide shell access.` The shell-access line sounds like an error but is the intended response: GitHub allows Git operations over SSH, not interactive shells.

CATCH-UP PROCEDURE ON `dev`. With the empty repository in place and the key registered, clone the reference repository, rewind `main` to the `v0.1` tag, point `origin` at your own empty repository, and push:

SANITY CHECK. `ls -l` should now list four files under `~/.ssh/`: the `id_ed25519` pair from Block 1 and the `id_ed25519_github` pair you just generated. If the Block 1 files are missing, you accepted the default path during `ssh-keygen` and overwrote the prod key. Restore `B1-complete` and rerun the step with the `-f` flag.

ADD THE KEY TO GITHUB. Copy the entire `ssh-ed25519 ...` line printed by `cat ~/.ssh/id_ed25519_github.pub`, go to Settings, then SSH and GPG keys, then New SSH key on GitHub, give it a title like `pixelwise-dev-vm`, and paste.

WHY `IdentityAgent none`? `IdentitiesOnly yes` alone tells `ssh` to ignore extra agent keys, but in practice some agent setups still offer them first. `IdentityAgent none` is the belt-and-braces fix: for connections to github.com, skip the agent entirely and read the key from the file you named, period. Other hosts, including `prod`, are unaffected and still use the agent normally, so password-less SSH into `prod` keeps working.

ALREADY POLLUTED THE AGENT? If you ran `ssh-add` earlier in the session and `ssh-add -l` now lists more than one key, drop them all from the agent and reload only the GitHub key:

```
ssh-add -D
ssh-add ~/.ssh/id_ed25519_github
```

`ssh-add -D` only forgets the keys held in agent memory; the files in `~/.ssh/` are not touched. Re-add the prod key later with `ssh-add ~/.ssh/id_ed25519` when you next log in to `prod` if you prefer not to retype the passphrase.

```
cd ~
git clone https://github.com/schutera/pixelwise.git
cd pixelwise
git reset --hard v0.1
git remote set-url origin \
    git@github.com:<you>/pixelwise.git
git push -u origin main
git push --tags
```

Why `reset --hard`? The clone lands you on the reference repository's current `main`, which is ahead of `v0.1`. `git reset --hard v0.1` moves your local `main` back to the tagged commit, discarding the later ones, so the working tree matches the end-of-Block 2 state exactly. There is no work to lose because you have not committed anything yet.

Mirror it on `prod`. By the end of Block 2, PixelWise also lived on the production VM at `/opt/pixelwise`. Restore `B1-complete` on `prod` as well, install `git`, then clone the reference repository to its permanent home and rewind `main` to `v0.1`:

```
sudo apt update && sudo apt install -y git
sudo mkdir -p /opt/pixelwise
sudo chown produser:produser /opt/pixelwise
git clone https://github.com/schutera/pixelwise.git \
    /opt/pixelwise
cd /opt/pixelwise
git reset --hard v0.1
```

You are now at the state where Block 2 ended, with both VMs aligned, ready to start Block 3.

No SSH key on `prod`. `prod` only ever pulls; it never pushes, so an HTTPS clone of a public repository is enough and no SSH key needs to be registered with GitHub. Skip the `ssh-keygen` ceremony you just did on `dev`. If your repository is private, or you later need to push from `prod`, generate a separate key on `prod` and add it to GitHub the same way.

Already have a fork? If your own fork already carries the `v0.1` tag, swap the URL for `git@github.com:<you>/pixelwise.git` on `dev` and the HTTPS equivalent on `prod`, so push and pull continue to target your account rather than the reference repo.

The Setup Script

Provisioning a server is the act of turning a freshly installed operating system into one that can run your application. So far it has been done by typing `sudo apt install` commands at the prompt and reading their output. That works for one-time setups, it does not scale. Software is meant to be scaled.

The setup script is a plain Bash file that captures every `apt install` the project needs. It lives in the repository alongside the code and is run on any fresh VM to bring it up to speed:

```
#!/bin/bash
# setup-server.sh: provision a fresh server VM
set -euo pipefail

sudo apt update
sudo apt install -y git python3 python3-pip \
    python3-venv curl
```

Configuration management is a whole discipline that grew from this small itch. Ansible, Salt, and Puppet declare the desired state of a machine in a structured format and reconcile the live system to match. `setup-server.sh` is the entry-level version: a script you can read end to end. The principle is the same.
Ansible: `https://docs.ansible.com/`

The first line of the body, `set -euo pipefail`, makes the script fail loudly. `-e` aborts on the first error, `-u` aborts on an undefined variable, and `-o pipefail` propagates a failed command in the middle of a pipeline rather than letting a successful command later on mask it. Half-configured machines are a common cause of mysterious bugs; the four-character preamble eliminates most of them.

The shebang line `#!/bin/bash` tells the kernel which interpreter to use when you run the file directly. Without it, the file is read as a sequence of commands by whichever shell happens to invoke it, which on Ubuntu is usually `dash`, not Bash, and the two differ in subtle ways that surface only when something breaks.

Mark it executable and run it:

```
chmod +x setup-server.sh
./setup-server.sh
```

Idempotence is the property that running a script twice yields the same result as running it once. `apt install` is idempotent by default: an already-installed package is a no-op. So re-running is always safe, and the script can be used to fix a machine that has drifted.

See how version control already pays off. The script is committed to Git, so when the VM is rebuilt or a new team member joins, `bash setup-server.sh` reproduces the exact same environment. The commands you typed once are written down where the next person can find them.

Exercise: The Setup Script

WITH THE SETUP SCRIPT MOTIVATED ABOVE, put it on the `dev` VM. At the root of the PixelWise repository, that is `~/pixelwise/` where you ran `git init` in Block 2, create a file called `setup-server.sh` that captures the system packages installed manually in Block 2:

```
#!/bin/bash
set -euo pipefail

sudo apt update
sudo apt install -y git python3 python3-pip \
    python3-venv curl
```

WHY AT THE REPO ROOT? Provisioning is a project-wide concern, not the responsibility of any single subdirectory, so `setup-server.sh` sits next to `README.md` and `.gitignore` where anyone cloning the repository will spot it immediately. Open the file with `nano ~/pixelwise/setup-server.sh` or your editor of choice and paste the contents below.

MARK IT EXECUTABLE AND RUN IT. On a machine that already has the packages, every `apt install` is a no-op; the script reports nothing to do, which is the desired outcome of an idempotent provisioning step:

```
chmod +x setup-server.sh
./setup-server.sh
```

(OPTIONAL) TRY IT ON A FRESH VM. The real test of the script is on a machine that has not seen the commands yet. A second `prod` VM, restored from the `B1-complete` image, clones the repository, runs `bash setup-server.sh`, and provisions end to end without you typing a single `apt install`.

COMMIT IT. This is the first file that makes provisioning reproducible:

```
git add setup-server.sh
git commit -m "Add server provisioning script"
git push
```

Virtual Environments

A VIRTUAL ENVIRONMENT is an isolated Python installation. Each project gets its own set of packages, independent of every other project and of the system Python. The mechanism is local: a directory, by convention called `.venv/`, that holds an interpreter, a copy of `pip`, and every library the project depends on. Activating the environment rewrites your shell's `PATH` so that `python` and `pip` resolve to the local copies.

```
python3 -m venv .venv
source .venv/bin/activate
```

THE DIRECTORY IS LARGE and machine specific, often 50 to 200 MB. Thus it is listed in `.gitignore`: never commit it. A teammate clones the repository, creates their own `.venv`, installs from `requirements.txt`, and ends up with the same packages without ever copying the binary contents.

WHY ISOLATION? Project A needs `numpy==1.24`; project B needs `numpy==2.0`. Without per-project environments, one of them breaks the moment the other is installed. With virtual environments, both coexist on the same machine and make it easy for you to switch between projects seamlessly.

VERIFY THE SWITCH. After activation, `which python` should report a path inside `.venv/bin/`, and `pip list` should show only `pip` and `setuptools`:

```
which python
# /home/user/pixelwise/.venv/bin/python

pip list
# Package    Version
# ---------- -------
# pip        24.0
# setuptools 69.5.1
```

A fresh environment starts empty. Every package you install from this point is tracked and intentional. `deactivate` returns the shell to the system Python.

THE RULE IS SIMPLE: every project gets its own `.venv`. Activate it before you work; deactivate when you are done. The one-time cost is a directory and a command; the ongoing benefit is never debugging a version conflict between two unrelated projects on the same laptop.

THE DATA SCIENCE ECOSYSTEM often uses `conda` instead of `venv`. Conda manages both Python packages and system-level libraries like CUDA or MKL, which makes it popular for GPU workflows. For web projects, `venv` is the lighter and more standard choice. Conda: `https://docs.conda.io/`

Exercise: The Virtual Environment

WITH THE RATIONALE FOR ISOLATION IN MIND, create one on the `dev` VM, inside the PixelWise repository:

```
cd ~/pixelwise
python3 -m venv .venv
source .venv/bin/activate
```

NOTICE THE PROMPT CHANGE. After activation, your shell prompt usually gains a `(.venv)` prefix to remind you which environment is active. If you ever wonder whether you are inside the project's environment, the prompt is the first place to look; `which python` is the second.

VERIFY THE SWITCH. `which python` should print a path inside `.venv/bin/`. `pip list` should show only `pip` and `setuptools`:

```
which python
pip list
```

A fresh environment is empty by design. The next exercise fills it with intent.

ADD THE ENVIRONMENT TO `.gitignore`. The `.venv/` directory must never be committed. Open the `.gitignore` from Block 2 and confirm `.venv/` is listed; if not, add it now. You will notice that currently we ignore `*.py` python files, but in future you do not want to ignore those, so remove that line:

```
.venv/
```

Run `git status`. The only change shown should be the `.gitignore` edit itself, even though `.venv/` sits right next to it on disk with hundreds of files inside. That silence is `.gitignore` doing its job: the absence of `.venv/` in the output is the demonstration, not a missing piece.

READ THE DIFF. `git diff .gitignore` shows what you added compared to the Block 2 version. Reading the diff before committing is the habit that catches accidental changes early; running it once now is muscle memory you will reuse every day for the rest of the course.

COMMIT THE CHANGE. With the diff read and `git status` clean of surprises, record the update so the next exercise starts from a known state:

```
git add .gitignore
git commit -m "Ignore .venv/ and stop ignoring *.py"
git push
```

Managing Packages with pip

PIP IS THE PACKAGE INSTALLER for Python. It downloads packages from PyPI, the Python Package Index at https://pypi.org/, and installs them into the active environment. The core cycle has three commands:

```
pip install scikit-learn joblib python-dotenv
pip freeze > requirements.txt
pip install -r requirements.txt
```

`pip install` downloads each named package and its dependencies and places them in `.venv/`. `pip freeze` prints every installed package with its exact version, including transitive dependencies you never asked for directly. `pip install -r` reads such a file and installs everything in it, the step that turns a captured environment back into a live one.

THE THREE PACKAGES. `scikit-learn` provides the classifier PixelWise will train and serve. `joblib` saves and loads trained models to disk as `.pkl` files. `python-dotenv` reads `.env` files into environment variables so secrets stay out of code.

READ THESE THREE LINES AS A CONTRACT. The first installs into the environment. The second writes a snapshot of it to a text file. The third reconstructs the same environment from the file on any machine. Together they are the reproducibility layer for your programming environment.

Anatomy of `requirements.txt`

ONE PACKAGE PER LINE, pinned to an exact version:

```
scikit-learn==1.4.0
joblib==1.3.2
python-dotenv==1.0.1
```

PINNING MATTERS because packages are a moving target. A package you install today as `numpy>=1.24` may resolve to `1.27` tomorrow, with new features added, new behaviour, new bugs, and possibly new security advisories. Packages are software, and they are subject to change; just the same as your own project. Pinning to `==1.24.3` freezes the resolution and makes switching to newer versions a deliberate act.

VERSION SPECIFIERS. `==` pins exactly, `>=` sets a floor, `~=` allows patch updates: `~=1.4.0` means `>=1.4.0, <1.5.0`. For maximum reproducibility, pin everything with `==`. Looser specifiers are appropriate for libraries that other projects depend on, which is not where most of your work lives.
Semantic versioning: https://semver.org

COUNT WHAT `pip freeze` WRITES. Three direct installs typically produce twenty or more lines, the rest are transitive dependencies pulled in automatically. The flatness is the point: every package that has to be present for the project to work is listed, even the ones you never typed. The cost is that the file is hard to read; the benefit is that it captures exactly what is on disk.

SECURITY AUDITING. Your dependencies have their own dependencies, and any of them can harbour known vulnerabilities. `pip-audit` checks every installed package against the CVE database: `pip install pip-audit && pip-audit`. Run it after every `pip freeze`. In Block 8 we wire it into a cron job so vulnerabilities surface without human labor.
`pip-audit`: https://github.com/pypa/pip-audit
Dependabot: https://docs.github.com/en/code-security/dependabot

Exercise: Pinning Dependencies and `pip-audit`

WITH `pip` AS THE PACKAGE INSTALLER AND `requirements.txt` AS THE CAPTURED ENVIRONMENT, put both to work. With the virtual environment active, install the three direct dependencies PixelWise needs in this block:

```
pip install scikit-learn joblib python-dotenv
```

FREEZE THE ENVIRONMENT into a pinned requirements file:

```
pip freeze > requirements.txt
```

Open `requirements.txt` and inspect. Note the pinned versions and the transitive dependencies that were pulled in automatically. Count how many lines `pip freeze` produced compared to the three packages you installed directly; the gap is the dependency graph working invisibly.

AUDIT THE DEPENDENCY GRAPH. Install `pip-audit` and run it against the active environment:

```
pip install pip-audit
pip-audit
```

Inspect the output. What is a CVE? What would you do if a critical vulnerability were reported in one of your dependencies?

COMMIT THE REQUIREMENTS:

```
git add requirements.txt
git commit -m "Pin core Python dependencies"
```

THE HABIT TO FREEZE AND COMMIT: after installing or upgrading any package, immediately run `pip freeze > requirements.txt` and commit the change. Stale requirements files are the most common cause of "works on my machine" incidents.

(OPTIONAL) READ ONE CVE. Pick any advisory `pip-audit` reports, click through to the CVE record, and read the description, the affected versions, and the fixed version. Practising this once now makes the production response straightforward when an alert fires under time pressure.

Configuration: `.env` and `python-dotenv`

Separating configuration from code is a foundational principle. Configuration is anything that varies between development, staging, and production: API keys, database URLs, feature flags, debug settings. Code is the part that does not change between deployments.

The Twelve-Factor App: https://12factor.net/
python-dotenv: https://github.com/theskumar/python-dotenv

A `.env` file holds key-value pairs that your application reads at startup:

```
SECRET_API_KEY=my-actual-secret-key-here
DEBUG=true
```

In Python, `python-dotenv` loads the file into process environment variables, after which the standard `os.getenv` reads them:

```
from dotenv import load_dotenv
import os

load_dotenv()
api_key = os.getenv("SECRET_API_KEY")
debug = os.getenv("DEBUG", "false").lower() == "true"
```

The `.env.example` Pattern

The real `.env` contains real secrets, so it must never be committed. `.env.example` is the contract: it lists which variables exist, explains what each one is for, and provides a safe placeholder. The example file is committed and public. The filled contracts live on the respective machines and stays out of Git.

A file for convenience. Environment variables can be exported manually, set inline before a command, or supplied by the systemd service file we install in Block 5. A `.env` file is the development-time convenience that mirrors the production-time mechanism: same names, same shape, different source.

```
# API key callers must send in the X-API-Key header
# (a secret, never a real value here)
SECRET_API_KEY=replace-me

# Debug mode: when true, FastAPI shows /docs and full
# error tracebacks. Never true in production.
DEBUG=true
```

On first setup create your actual `.env` by copying `.env.example` and replacing each placeholder with a real value. The contrast is immediate: one is the template, the other is a vault for your secrets.

The pattern grows block by block. Every new service we add, the API key in Block 5, the database URL in Block 6, the model path in Block 4, lands in `.env.example` as a new line with a comment explaining its purpose. A teammate cloning the repository six weeks from now sees the full surface of the application's configuration in one file.

Exercise: Configuration and Secrets

With `.env` and `.env.example` as the contract for separating configuration from code, draft both files. Create `.env.example` at the repository root with two entries:

```
# API key (secret, never a real value here)
SECRET_API_KEY=replace-me

# Debug mode (never true in production)
DEBUG=true
```

Create your real `.env` by copying the example and replacing the placeholder with a value you choose:

```
cp .env.example .env
```

Edit `.env` so `SECRET_API_KEY` holds a value of your own choosing. Make sure to add `.env` to `.gitignore` if it is not already there, then Run `git status` and confirm that `.env` does not appear.

Commit the example only:

```
git add .env.example .gitignore
git commit -m "Add .env.example and update .gitignore"
```

The contrast is immediate. The example is now public, the real values stay on the machine.

The naming convention is uppercase identifiers with underscores. Environment variables are case sensitive on Linux but case insensitive on parts of Windows; convention picks the form that works everywhere.

If `.env` appears in `git status`, the `.gitignore` entry is missing. Add it now and verify again before committing anything. A secret committed once is in the history forever, even if the next commit deletes it.

Exercise: Replicate on the Server

THE REPRODUCIBILITY PROMISE is verifiable only by running the recipe on a different machine. SSH into the `prod` VM, pull the latest code, and recreate the environment from the two files you committed:

```
ssh produser@192.168.56.11
cd /opt/pixelwise
git pull origin main
bash setup-server.sh
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

WHY `origin main`? After the catch-up `reset --hard v0.1`, prod's local `main` may have lost its upstream tracking, in which case bare `git pull` asks you to specify a branch. Naming `origin main` explicitly sidesteps that. To restore the tracking once and let plain `git pull` work afterwards, run `git branch --set-upstream-to=origin/main`.

VERIFY THE AUDIT ON `prod`:

```
pip-audit
```

The same advisories should appear as on `dev`. Identical inputs, identical outputs. This is what reproducibility looks like in practice.

WATCH THE INSTALL OUTPUT. `pip` resolves the dependency graph from the lock-shaped `requirements.txt` and downloads each package at the pinned version. The output you see on `prod` should match the output you saw on `dev` the first time you ran `pip install`; if it does not, something has drifted.

FROM THIS BLOCK ONWARDS, the GitHub repository is the single source of truth. `prod` only ever pulls; it never authors. A `git pull` on `prod` should always be the deployment step, never a place to make edits.

Optional: When `requirements.txt` Starts to Hurt

`pip freeze > requirements.txt` captures everything: your direct dependencies and all their transitive dependencies, flattened into one list. Six months later you want to upgrade one package: which of those forty-seven packages did you deliberately install, and which were pulled into that list automatically? Without that distinction, every upgrade is a guessing game. Solving dependency conflicts becomes a nightmare and quickly feels like brute-force.

`pyproject.toml` solves this by separating "what you depend on", your direct deps loosely pinned, from "what gets installed", a generated lock file with the full resolved graph. Upgrades become surgical: bump one line, regenerate the lock, done.

Tools like `uv` and `poetry` use the project-file model and are fast becoming the standard. Once the `requirements.txt` workflow is second nature, this is the upgrade that makes dependency management sane at scale.
Python Packaging User Guide: `https://packaging.python.org/en/latest/`
uv: `https://docs.astral.sh/uv/`
Poetry: `https://python-poetry.org/`

```
[project]
name = "pixelwise"
version = "0.1.0"
dependencies = [
    "scikit-learn>=1.4",
    "joblib>=1.3",
    "python-dotenv>=1.0",
]
```

Compare this to `requirements.txt`: the project file lists only what you chose to install, with flexible version bounds. The tool resolves the full dependency graph and writes a lock file that pins every transitive dependency exactly. You commit both files; the lock file is your reproducibility guarantee and the project file is your editable surface.

```
# to upgrade a single package and update lock:
poetry update <package>
```

We stay with `requirements.txt` through the rest of the course because it is the format every Python developer recognises and the only one some hosting environments still accept.
The reasoning generalises: when an upgrade is available, learn the workflow beneath it before adopting the tool that abstracts it away.

Optional: The Container Alternative

VIRTUAL ENVIRONMENTS ISOLATE PYTHON PACKAGES. Containers isolate the entire operating system: libraries, system packages, Python version, environment variables, even the filesystem layout. Where `requirements.txt` specifies which Python packages to install, a `Dockerfile` does the whole stack in layers: This Linux image, install these system packages, then install these Python packages.

A MINIMAL DOCKERFILE for a Python service looks much like the two scripts you are about to write, compressed into a single declarative file:

```
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "main.py"]
```

The result is a self-contained unit that runs identically on any machine with Docker installed. The trade is opacity: when something fails inside a container, the layers between your code and the kernel multiply, and debugging across them requires familiarity with the layer above and below.

WE DO NOT USE DOCKER IN THIS COURSE, but you should know it exists and why teams adopt it. The vocabulary, image, container, registry, layer, volume, recurs in any later production work; the next-block decisions in this course translate to it directly.

A NOTE ON WHAT COMES NEXT. The repository now has a scaffold, pinned dependencies, and a provisioned server, but no application code. `app/` is an empty Python package waiting for content. In the next block we drop a trained model into `models/`, write the inference layer in `app/classifier.py`, and verify end to end that PixelWise can classify a digit before any API or frontend exists.

DOCKER and Docker Compose solve the reproducible environment problem at the OS level rather than at the Python level. This course teaches the virtualenv path to keep the OS layer visible and learnable. Later Docker enables you to run Linux containers under Windows as well. Docker is the natural next step after this course; it does not replace any of what you learn here, it builds on it. Docker: `https://www.docker.com/` Docker docs: `https://docs.docker.com/get-started/`

WHY WE HOLD OFF. The OS layer is the layer most students are least exposed to. Containers hide it before students have seen it. By the end of this course you will have provisioned, deployed, and monitored a service on bare Ubuntu; lifting that into a container at that point is mechanical, and you will know exactly what each line of the Dockerfile means.

Self-Reflection and Recap

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

- Why do we use a virtual environment instead of installing packages system-wide?
- What is the difference between `.env` and `.env.example`, and why do we need both?
- What does `pip freeze` capture that you did not explicitly install, and why does it matter?
- If a colleague clones your repository, which two commands do they run to get a working environment?
- What is the risk of committing `.env` to Git, even briefly, and what would the recovery look like?
- Why does the course teach virtual environments before containers, even though many production systems use containers?

Recap of key concepts:

- `setup-server.sh` captures system-level provisioning so any fresh VM can be brought up with a single command.
- Virtual environments isolate Python packages per project; never install globally.
- `requirements.txt` pins every dependency so both machines run identical code.
- `.env.example` documents the configuration contract; `.env` holds the real values and never enters Git.
- `pip-audit` catches known vulnerabilities in the dependency graph and runs anywhere a Python environment lives.

Milestone. PixelWise has `app/`, `data/`, `models/`, a virtual environment, pinned dependencies, a setup script, and a minimal `.env.example`. The `.gitignore` is doing real work. Anyone who clones the repository can recreate the environment in a handful of commands. The next chapter integrates a trained machine learning model so the project can actually classify a digit.

Without writing application code yet, we already have a reproducible system on both machines. Anyone with the repository and the script can reach the same starting line.

Teaser. What does it take to turn a trained model file into something the rest of the application can call?