PROF. DR.-ING. MARK SCHUTERA

FULL STACK HANDWERK

UNFINISHED LECTURE NOTES

PUBLISHED BY UNFINISHED LECTURE NOTES

These notes are, by their very nature, unfinished, and they improve with every reader. If you spot an error, disagree with a framing, or want to add a source, a question, or a position, open a pull request at https://github.com/Quillstacks/LectureMaterial/tree/main/lecturenotes/notes_missingsemester. Every contribution is welcome.

2026-05-10 · happy pear Nachtigall

“SEINE WORT’ UND WERKE MERKT ICH UND DEN BRAUCH,

UND MIT GEISTESSTÄRKE TU ICH WUNDER AUCH.”

DER ZAUBERLEHRLING

Contents

Introduction

This is an adaptation of the MIT Missing Semester course material. And oh boy, we need it here in Germany as well! In the Länd of the "Ingenieure", "Handwerk", and "Lehrlings" - we forgot the importance of practice and mastering our tooling skills.

Original course URL https://missing.csail.mit.edu/

As a mechanical engineer by training, I have always had this nagging feeling that something important was missing from my education. The more I transitioned into software engineering roles, the more this feeling intensified. Looking at your curriculum, there is no shortage of advanced concepts and theories to learn. This one fills the gap that many computer science programs overlook: practical proficiency with essential tools and workflows that every software engineer relies on. And truth be told, Professors rely on students having this basic proficiency already acquired, or to figure this stuff out on their own - which is fair in academia. For me, software engineering is also a practical discipline, and ought to be taught as such. I need you to know this stuff, so it is also fair to teach it.

This course will teach you everything you need to know to become proficient with the command line, version control systems like Git, text editors, shell scripting, and other essential tools that are the backbone of modern software development - in the end you will be able to develop and deploy your software solutions. This will make you a better software engineer, and it will make your life easier latest when you need to go full ML Lifecycle.

Projektbericht

2026-05-09 · quiet orange Antilope

Real artists ship. - Steve Jobs

The Why

A lecture that teaches tools and workflows must also assess tools and workflows. Written exams test recall; a project report tests whether you can actually build and deploy something. Enter the Projektbericht.

Pick any block from the course and build a meaningful extension onto it. The extension should add functionality, improve reliability, or strengthen security in a way that goes beyond what the lecture covered. The choice is yours, but each extension must be unique across the class: submit your proposal for approval on a first come, first served basis with me. If your proposal strikes home with me, we can also discuss small full stack projects, which live outside the scope of the lecture material.

Submit a written report as PDF and a link to your snapshot.
The report must cover:

- Which block you chose and what it covers in the course.
- What you built on top of it, which methods, tools and techstack you used, and why it adds value.
- Open questions and problems you encountered and how you resolved them.
- What works, what remains incomplete, and what would you do next.

The Projektbericht as in the Prüfungsordnung. The official frame for the Projektbericht is set by the DHBW Studien- und Prüfungsordnung Wirtschaft, available here:

"Die zu prüfende Person soll zeigen, dass sie in der Lage ist, Projekte oder Studien mit wissenschaftlicher oder praktischer Problemstellung selbstständig zu bearbeiten sowie deren Ergebnisse schriftlich zu dokumentieren. Der Projektbericht soll je zu prüfender Person in Modulen mit fünf bis sechs ECTS-LP zehn bis zwölf Seiten, in Modulen mit sieben beziehungsweise acht ECTS-LP 14 bis 16 Seiten und in Modulen mit neun beziehungsweise zehn ECTS-LP 18 bis 20 Seiten umfassen."

How to write the report. A practical guide for writing reports and theses lives here.

For deeper druidic knowledge on writing a thesis, consult here.

Evaluation considers three dimensions: *depth* of the extension beyond the lecture material, *understanding* demonstrated through explanation and a VM snapshot, and *quality* of the writing.

Looking for inspiration? Explore what tools modern teams actually use here or browse the ThoughtWorks Technology Radar here. Pick something that excites you; the best reports come from genuine curiosity.

Development and Production

2026-05-10 · happy pear Nachtigall

Divide and conquer.

The Why

Most software engineering curricula focus on algorithms, data structures, and the theory of computation. Writing a Python script that runs once on a single laptop is a skill; making that script available as a service, keeping it running, and recovering when it fails is a different one. This course addresses the second set of skills, the tools and workflows that sit between working code and the full stack of setting up and maintaining a running system.

Across ten lecture blocks we build PixelWise, a handwritten digit recognition web application. The starting point is two empty virtual machines; the end point is a deployed service including a frontend, and a backend with a trained model, an API, and a database, all running on Linux servers and connected over a network, with version control, automated deployment, persistent storage, and monitoring. Each tool we introduce is motivated by a specific need that arises in the construction of a fullstack application: Pixelwise.

PixelWise is the narrative thread of the entire course. Users draw a digit on a pixelated 28×28 canvas in the browser, the backend runs an image classifier trained on MNIST, the model predicts class and confidence, and results are stored in a database.

This enables you to deploy your own prooducts and services in the future.

Development and Production.

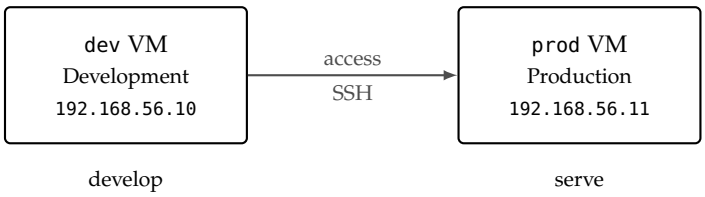

Figure 1: The two-machine setup. The dev VM is where code is written and tested; the prod VM is where the deployed application runs. Code travels from dev to prod through a deliberate deployment step over SSH.

The figure shows the starting arrangement: two virtual machines that play distinct roles throughout the course. The `dev` VM is the

development environment in which code is written, modified, and tested. The `prod` VM is the production environment in which the deployed application runs. Later blocks add the backend stack, the frontend, persistence, and automation between the two, but the separation between development and production stays intact from this block onward.

Hands On Experience

Consider a small web application running on your laptop. A classmate two time zones away wants to try it; you send them the address. It works, until you close the lid to catch a train. That evening the Wi-Fi has changed, the address is different, and the process you started in the terminal is gone.

Count the hours your laptop is actually reachable at a stable address. Subtract the time it is closed, asleep, on battery, behind a café router, or rebooting after an update. What remains is the service level a laptop alone can offer.

A server is a machine whose job is to be available when no one is watching. It stays powered on, holds a stable address, and answers requests around the clock. Your laptop is where code is written and changed; the server is where code meets users. Keeping these two roles on separate machines is the first design decision of the course.

Two machines, one mental model. `dev` at 192.168.56.10 stands in for your laptop. `prod` at 192.168.56.11 stands in for the always-on host. Later lectures fill in the bridge.

This first block focuses on machines and environments rather than on code. Before introducing a programming language, a version control system, or a framework, we need a place which is always up, to run programs, isolated from experiments, and a place for development. By the end of the block you will have provisioned both virtual machines,

- set up a server and a development machine as virtual machines on your laptop,
- learned the subset of Linux needed to navigate a server from the command line,
- and configured remote access over SSH using key-based authentication.

Virtual Machines

A VIRTUAL MACHINE IS A SIMULATED COMPUTER running inside your real computer. The software that creates and runs virtual machines is called a hypervisor. The hypervisor partitions the host's CPU, memory, and disk, and presents each virtual machine with what appears to be its own hardware. From the guest operating system's perspective, it is running on a physical machine.

HYPERVISORS are often categorised as type 1 or type 2. A type 1 hypervisor runs directly on the hardware, with the host operating system built into the hypervisor itself. Examples include VMWare ESXi and Microsoft Hyper-V in its server role; cloud providers use type 1 hypervisors to host the virtual machines you rent. A type 2 hypervisor runs as an application on top of a regular operating system. Oracle VirtualBox Manager is a type 2 hypervisor, which is why we can install it as a normal program on Windows, macOS, or Linux. The conceptual model is the same in both cases; the differences are in performance and deployment context.

ORACLE VIRTUALBOX MANAGER at `https://www.oracle.com/virtualization/virtualbox/` is the hypervisor we install on the host. Virtmanager and VMWare serve the same role with different trade-offs, and we could run this course on any of them.

ISOLATION is the primary reason to use a virtual machine. A misconfiguration, an uninstalled package, or a destructive command inside the guest does not affect the host operating system. Because the hypervisor abstracts away the hardware, the same VM runs identically on Windows, macOS, and Linux hosts, which is the basis of the reproducibility argument.

A SNAPSHOT captures the entire machine state at a point in time, including disk contents, memory, and running processes. Snapshots are inexpensive to create and well-suited to exploratory work; making one before a risky change and restoring it afterwards is a standard pattern.

THERE ARE COSTS. A virtual machine executes a full operating system on top of the host, which carries a performance overhead that is typically small for I/O-bound work and larger for CPU-bound work. Each VM reserves some amount of RAM, disk space, and CPU time from the host, so the number of concurrent VMs is limited by available resources. Boot time is longer than launching an application but shorter than booting physical hardware. These costs are the price paid for isolation, reproducibility, and rollback.

OUTSIDE THE CLASSROOM, a production host is usually a rented or purchased server in a data centre, and the development machine is the laptop or workstation in front of you. Running both as VMs inside a single laptop is a teaching shortcut: it preserves the two-machine separation without asking you to rent hardware, and everything we learn here transfers directly when `prod` is later replaced by a cloud instance reached over the public internet.

HOST-ONLY NETWORKING puts `dev` and `prod` on a private subnet, typically 192.168.56.0/24, that only the host and the guests can reach. The guests get internet through a second, NAT-mode adapter, so they can install packages without being exposed to the outside world. We assign static IPs, 192.168.56.10 for `dev` and 192.168.56.11 for `prod`, so that every command in the course refers to the same addresses.

Exercise: Provisioning Virtual Machines

Exercises are for practice and reinforcing concepts. Work through them yourself first, break things, discuss, this is not a time trial. If your VM ends up in a state you cannot recover, restore the `B1-fresh` snapshot and start over. At the end of the session, take a snapshot named `B1-complete`, your safety net for the next block.

Install the Oracle VirtualBox Manager for your host OS with default settings, and download two Ubuntu LTS ISOs (go for version 24.x.x if you find one): the Desktop edition for `dev` and the Server edition for `prod`. The asymmetry is deliberate, `dev` carries a graphical environment so you can run an editor and a browser locally, while `prod` stays headless because a production server has no business running a desktop. You can also decide to run dev on your machine directly, but this is not further supported in the excercises.

Where to get them. VirtualBox: `https://www.oracle.com/virtualization/virtualbox/`. Ubuntu Desktop LTS ISO: `https://ubuntu.com/download/desktop`. Ubuntu Server LTS ISO: `https://ubuntu.com/download/server`. Pick the LTS release in both cases, not the interim one, so support extends across the semester. On an Ubuntu host you can use GNOME Boxes instead: `https://apps.gnome.org/Boxes/`. O n macOS, UTM is the closest equivalent: `https://mac.getutm.app/`.

Create the `dev` VM in the Oracle VirtualBox Manager. Click `New` and configure:

- Name `dev`, Type `Linux`, Version `Ubuntu (64-bit)`
- Memory at least 4096 MB, CPUs at least 2
- Hard disk at least 25 GB, VDI, dynamically allocated

Create the `prod` VM with the same procedure but tighter resources, since it runs headless:

- Name `prod`, Type `Linux`, Version `Ubuntu (64-bit)`
- Memory at least 2048 MB, CPUs at least 2
- Hard disk at least 10 GB, VDI, dynamically allocated

Installer choices that matter: use distinct credentials per VM, `user/password` on `dev` and `produser/prodpassword` on `prod`. The asymmetry is deliberate, you should feel that logging into `prod` is a different act from working on `dev`.
Why the spread. Ubuntu Desktop with GNOME wants roughly 4 GB of RAM and 25 GB of disk to feel responsive. Ubuntu Server has no GUI and runs comfortably in a quarter of that. Sized this way the pair fits inside a 6 GB / 35 GB envelope on the host.

Attach the matching ISO to each VM. VirtualBox offers an unattended installation checkbox during creation, don't select it, you want the interactive installer. Once a VM exists, select it in the manager and open `Settings`, `Storage`. Under `Storage Devices` you will see a controller, `IDE` or `SATA`, with an `Empty` optical drive beneath it. Click that `Empty` entry, then on the right under `Attributes` click the small disc icon next to `Optical Drive`, choose `Choose a disk file`, and pick the Desktop ISO for `dev` or the Server ISO for `prod`. Confirm with `OK`, start the VM, follow the installer, when in doubt go with the defaults and just forward-enter.

After the first boot of each VM (both dev and prod), install the OpenSSH server to enable remote access:

```
sudo apt update
sudo apt install openssh-server
```

This step is required for both Desktop and Server editions. Reboot after installation.

Host-only network. In `File`, `Host Network Manager` (or `File`, `Tools`, `Network Manager` on version 7+), create a host-only network if none exists, typically 192.168.56.1/24. **Disable DHCP** for this network so you can assign static IPs manually.

CONFIGURE THE NETWORK ADAPTERS FOR EACH VM: Select the VM in VirtualBox Manager and open `Settings`, then go to the `Network` section. For Adapter 1, check "Enable Network Adapter" and set it to `NAT` (for internet access). For Adapter 2, select the Adapter 2 tab, check "Enable Network Adapter," set it to `Host-only Adapter`, and choose the host-only network you created.

MISSING ADAPTER 2? Power-off the VM, make sure you are in Expert-mode, go to `Settings`, `Network`, select Adapter 2, check "Enable Network Adapter," and set it to `Host-only Adapter`.

ASSIGN STATIC IPS INSIDE EACH VM:

1. Boot the VM and log in.
2. Run `ip a` to list network interfaces. The host-only adapter is usually `enp0s8`, but confirm whether the name shows up in the output.
3. Edit the Netplan config with `sudo nano /etc/netplan/00-installer-config.yaml` Save and exit nano with (Ctrl+X, Y, Enter).
4. For `dev`, set the config as:

```
network:
  version: 2
  ethernets:
    enp0s8:
      addresses:
        - 192.168.56.10/24
```

OPEN A TERMINAL (DESKTOP). On Ubuntu Desktop press `Ctrl+Alt+T`, or open `Activities` (top-left) and type "terminal"; press Enter. You can also right-click the desktop and select "Open Terminal" if the menu is present.

For `prod`, use `192.168.56.11/24` instead.

5. Apply the config with `sudo netplan apply`
6. Check the new address with `ip a` again; you should see the static IP assigned to `enp0s8`.

NETPLAN PERMISSION WARNING. Netplan will warn if a configuration file is writable by group/others or not owned by `root`. This is a security check to avoid loading untrusted configuration files; expected ownership is `root:root` and modes such as `644` for the YAML files.

VERIFY CONNECTIVITY: On `dev`, run `ping 192.168.56.11` to test if `prod` responds. If you get no reply, double-check the Netplan configuration (correct interface and address), ensure Adapter 2 is set to Host-only in both VMs, and confirm that both VMs are running and attached to the same host-only network.

TROUBLESHOOTING. If the interface name is not `enp0s8`, use the correct one from `ip a`.

```
user@dev:~$ ping 192.168.56.11
PING 192.168.56.11 (192.168.56.11) 56(84) bytes of data.
64 bytes from 192.168.56.11: icmp_seq=1 ttl=64 time=1.63 ms
```

Figure 2: ping

Linux on the Server

Linux is a family of Unix-like operating systems built around a common kernel, first released by Linus Torvalds in 1991. A distribution combines the kernel with a package manager, system libraries, and user-space tools into a coherent product. Ubuntu, Debian, Fedora, and Arch are distributions that differ in release cadence and defaults but share most of what matters at the command line.

A server in this section is a computer configured to run services and accept connections over a network, typically without a graphical session. You interact with it through a shell, a program that reads commands and prints their output; the shell we use is Bash, the default on Ubuntu.

Five commands are sufficient for the first session. `ls` lists a directory, with `-la` showing permissions, ownership, and hidden files. `cd` changes directory; `cd ..` moves up one level and `cd ~` returns home. `pwd` prints the current path, `mkdir` creates a directory, and `cat` prints a file to the terminal.

Ubuntu at `https://ubuntu.com/` is the distribution we use on both VMs. Long-term support releases receive five years of security updates, which is why we pick the LTS ISO rather than the latest release.

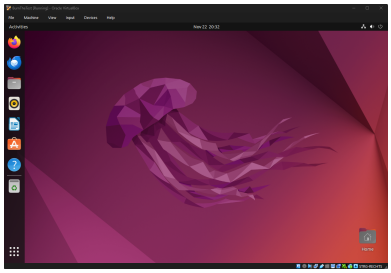

Figure 3: A running Ubuntu guest inside the Oracle VirtualBox Manager. The guest believes it owns a physical machine; the hypervisor is what makes that illusion hold.

Linux is a multi-user operating system. Every file belongs to a user and a group and carries three permission sets, for owner, group, and everyone else. Each set independently grants read, write, or execute access. `whoami` reports the current user, and `ls -la` shows ownership and permissions:

```
-rw-r--r-- 1 user user 1234 Jan 15 10:00 config.txt
drwxr-xr-x 2 user user 4096 Jan 15 10:00 mydir/
```

The Unix philosophy shapes the design of these tools. Each program does one thing well, programs pass streams of text to each other, and configuration lives in plain files. The result is a small vocabulary of composable commands that combine into pipelines no single command was designed for.

Two commands modify these. `chmod` changes permissions and `chown` changes ownership:

```
chmod 600 ~/.ssh/id_ed25519          # owner-only read and write
sudo chown user:user file.txt        # assign user and group
```

Reading the mode string from left to right, the first character tells you whether it is a file or directory, then three groups of `rwx` for owner, group, others. A dash means the permission is not granted. The leading `d` marks a directory.

The principle of least privilege applies throughout the course: grant each user, process, or service only the access it needs, so that narrow defaults limit the blast radius of a compromise or a mistake.

Exercise: Navigating Linux

This exercise turns the five commands and the permission model into something your hands remember. You walk the directories common to every Linux system, identify yourself to the machine, and read and change a file's mode string. By the end, the output of `ls -la` should read as a description you can speak aloud rather than a blob of symbols.

Ten-finger typing is part of the toolchain once the terminal is your main tool. A server has no menus, so your throughput is bounded by how quickly you can speak to the shell without looking at the keys; every hunt for a slash, tilde, or brace is attention spent on the keyboard instead of the problem. Ten minutes a day at `https://www.keybr.com/`, `https://typing.io/`, or `https://monkeytype.com/` is enough to remove that tax over a few weeks.

Explore the filesystem by visiting `/home`, `/etc`, and `/opt` in turn with `cd` and `ls -la`, and read two files with `cat`:

```
cat /etc/hostname
cat /etc/os-release
```

The filesystem as map. `/home` is where users live, `/etc` holds configuration, and `/opt` is where PixelWise will be installed in a later block. These locations reflect decades of convention you will meet on every Linux machine; knowing them lets you locate things by instinct.

Create a workspace and check that you are where you think you are:

```
mkdir ~/workshop && cd ~/workshop
pwd
touch notes.txt
rm notes.txt
```

`pwd` is a reassurance command. When something does not behave as you expect, asking the shell where it thinks it is usually explains the problem faster than re-reading the error.

Understand who you are and look at the permissions on your new file:

```
whoami
id
ls -la notes.txt
```

Decode the first column of `ls -la` against the margin note on mode strings from the theory above. Name aloud what each of the ten characters means; if you can do that without looking, permissions have stopped being mysterious.

Change the mode and watch it:

```
chmod 600 notes.txt
ls -la notes.txt
```

The next exercise will require `600` on your SSH private key, or OpenSSH refuses to use it; you have now seen what `600` looks like and why ownership matters.

A thought experiment on least privilege. Your `user` account can read `/etc/os-release` but cannot edit `/etc/ssh/sshd_config` without `sudo`. Why does that separation exist, and what would change if the web service we deploy in a later block ran as `root`? Name one file an attacker who compromised such a service could reach that a non-root service could not; the mental habit matters more than the exact file.

SSH, the Only Door

SSH, the Secure Shell, is a protocol for operating a remote machine from a terminal over an encrypted channel. In the setup used in this course, SSH is the only means of access to the `prod` VM: there is no graphical login and no console attached. Misconfiguring SSH can lock a user out of the machine, which is why key material and the daemon's configuration are treated carefully in the exercises.

The basic command takes the form `ssh <username>@<host>`; for example, `ssh produser@192.168.56.11` opens an encrypted connection to the server as the `produser` account. All input and output travel through this channel until the session is closed with `exit` or `Ctrl+D`.

Password-based authentication is vulnerable to guessing, brute-force attacks, and phishing. SSH supports an alternative in which the user presents a cryptographic key pair. The private key is kept on the client and never transmitted; the public key is placed on the server. During authentication the server issues a challenge that only the holder of the private key can answer, proving identity without the key itself ever leaving the client.

Generate an Ed25519 key pair on `dev`:

```
ssh-keygen -t ed25519 -C "you@example.com"
```

The command writes the private key `id_ed25519` and the public key `id_ed25519.pub` to `~/.ssh/`. The private key must have permissions `600`, owner-only read and write, or SSH refuses to use it.

Copy the public key to the server:

```
ssh-copy-id produser@192.168.56.11
```

This appends the key to `~/.ssh/authorized_keys`. Subsequent connections authenticate using the key pair and do not prompt for a password.

Once key authentication is verified, password authentication can be disabled on the server. Edit `/etc/ssh/sshd_config` to set `PasswordAuthentication no`, then restart the daemon:

```
sudo systemctl restart ssh
```

OpenSSH at `https://www.openssh.com/manual.html` is the implementation shipped with Ubuntu and most other Linux distributions. The client on `dev` and the daemon on `prod` are both part of this project.

PuTTY at `https://www.putty.org/` is a legacy SSH client for Windows, from the years before OpenSSH shipped with the operating system. Windows 10 and later include `ssh.exe` natively, so PuTTY is rarely necessary; you may still encounter it in older tutorials and on locked-down corporate machines.

VS Code Remote-SSH at `https://code.visualstudio.com/docs/remote/ssh` runs the editor locally while the language server, terminal, and file system live on the remote host. It reads `~/.ssh/config` and reuses the same keys as the `ssh` command, so any host you can reach with `ssh user@host` is one click away inside the editor.

Ed25519 keys use elliptic-curve cryptography and produce shorter keys than the older RSA algorithm while providing comparable or stronger security. They are the default choice on modern SSH clients.

Managing private keys. Private keys live in `~/.ssh/` on the machine that initiates connections, protected by filesystem permissions and, when generated with a passphrase, by symmetric encryption. `ssh-agent` holds the decrypted key in memory for the session so the passphrase is typed once per login. Password managers such as KeePass at `https://keepass.info/` or 1Password can store the passphrase and, with their SSH-agent integrations, supply the key to `ssh` without writing it to disk in plaintext. Keys are never synced to cloud storage in the clear, and a separate key pair per machine keeps the blast radius of a compromised laptop local.

After this change, password-based login attempts are rejected before credentials are even evaluated, which eliminates brute-force and credential-stuffing as viable attacks against the machine. The exercise introduces `nano` as a terminal editor and `systemctl` as the interface to the `systemd` service manager, both of which return throughout the course.

Exercise: SSH and Key-Based Access

This exercise walks through three states of remote access in order: password login, key-based login, and password login disabled. The point is not the commands but to recognise which state you are in at any moment and why each is stronger than the last.

First, log in with a password. From `dev`, open a session to `prod`:

```
ssh produser@192.168.56.11
```

Type your password, look around briefly, and log out with `exit`. This is the baseline. Every later step is interpretable as an upgrade over this state.

Generate a key pair on `dev` and install it on `prod`:

```
ssh-keygen -t ed25519 -C "you@example.com"
ssh-copy-id produser@192.168.56.11
```

Reconnect with `ssh produser@192.168.56.11` and notice the password prompt is gone. Log in and inspect what `ssh-copy-id` wrote for you:

```
cat ~/.ssh/authorized_keys
```

What just happened. `ssh-copy-id` appended the contents of `id_ed25519.pub` on `dev` to `~/.ssh/authorized_keys` on `prod`. At the next connection the daemon issued a challenge that only the private key on `dev` could answer; no secret material ever left the client. Seeing your public key in a readable file anchors the whole exchange to something concrete.

Disable password authentication. Edit `/etc/ssh/sshd_config` with `sudo nano`, set `PasswordAuthentication no`, and restart the daemon:

```
sudo systemctl restart ssh
sudo systemctl status ssh
```

The `status` output should show `active (running)`. If it does not, keep your current session open and fix the configuration before logging out; a locked-out VM is rescued by restoring the snapshot, not by wishful thinking. From a machine without the key, the daemon now rejects the attempt without even asking for a password.

Nuke the machine. From `prod`, run `sudo rm -rf /` to delete all files on the machine. Restore from the snapshot.

Run prod without GUI. In future run prod headless without GUI from the VirtualBox Manager. You can still log in with SSH and run commands, but there is no desktop to interact with. This is how a real production server works.

Take a snapshot of both VMs once everything works, naming them `B1`; this is your known good state for the rest of the course. Then deliberately break something inside a VM, watch the fallout, and restore the snapshot. You now have physical evidence that the safety net holds.

A note on what comes next. The commands you ran in this session are reproducible but not yet reproduced. You ran them in a terminal, they worked, and they are gone. In the next block we place them under version control and turn the sequence into a script that any future VM can run end to end, without you remembering anything.

Self-Reflection and Recap

SELF-REFLECTION Questions which can guide your thoughts during and after the exercises:

- Why do we use two separate machines instead of running everything on one?
- Why is disabling password authentication a stronger protection than picking a longer password?
- What does the principle of least privilege mean in the context of file ownership, and where did it show up in today's workshop?
- If your laptop died tomorrow, how much of today's setup could you reproduce, and what would you lose?
- Which of the five Linux commands do you still have to look up, and which one surprised you?
- How does the two-machine model map to a real cloud deployment with staging and production? How is our development machine different from staging?

RECAP of Key Concepts:

- Virtual machines isolate experiments from the host and make snapshots a first-class safety net.
- The two-machine architecture separates development from production at the hardware level.
- Files on Linux carry owners, groups, and three sets of permissions, and `chmod` and `chown` manage both.
- SSH is the only door into the server, and key-based authentication replaces passwords with cryptography that cannot be guessed.

MILESTONE. So far, we own the machine. The next step is to own the way we build software on it. In the next chapter we introduce Git and GitHub: not just a version history, but the collaboration layer that lets a team branch, review, and merge work in parallel. It is the tool that turns two developers editing the same file from a disaster into a workflow. ,

WITHOUT SHIPPING ANYTHING YET, we already have two machines that talk to each other, a secure door, and a known good state to fall back on.

TEASER. You and a classmate both want to improve PixelWise at the same time. How do you work on the same codebase without overwriting each other's changes, review what the other person did, and merge the results safely?

Version Everything: Git & GitHub

2026-05-09 · vibrant kiwi Reiher

Git a Life.

The Why

Block 1 gave us two machines and a way to reach one from the other. We can write code on `dev` and, eventually, deploy it to `prod`. But nothing yet records what changed between deployments, who changed it, or how to undo a mistake. The moment two people touch the same file, the same function, or one experiment needs to coexist with another, the workflow breaks down entirely.

PixelWise needs to live on two machines. Right now there is no reliable way to move code from `dev` to `prod` and keep both in sync. Git gives us exactly that: a shared history that either machine can pull from, so deployment becomes a deliberate, testable, repeatable step instead of manual file copying.

Version control solves an entire family of problems at once. It records every change with a message, a timestamp, and an author, so the full history of a project is always available. It lets you return to any previous state in seconds, turning risky experiments into cheap, reversible ones.

Parallel development becomes natural. Multiple people work on the same codebase without overwriting each other's changes. A new feature, a bug fix, and a refactor can all proceed simultaneously on isolated branches and be integrated later with explicit, reviewable merge steps.

Downtime shrinks as a direct consequence. Developers merge through a guided review step that catches conflicts and errors before they reach `main`, deploying only code that has already been integrated and verified. When something still slips through, you can roll back to the last known-good state in a single command instead of debugging under pressure while users wait.

Combined with a hosting platform like GitHub, version control becomes the backbone of code review, continuous integration,

GitHub: `https://github.com`

issue tracking, and project management, built around the software development process.

Hands On Experience

CONSIDER THIS SCENARIO: you and a classmate are both improving the PixelWise classifier. You rewrite the data loader; your classmate changes the model architecture. You each work on your own laptop for a few hours, then try to combine the results. Without a system for tracking changes, combining means copying files back and forth, comparing line by line, and hoping nothing was lost. Worse still, if both of you would have worked on the same machine, overwriting each others files as you go.

GIT IS THAT SYSTEM. It tracks exactly what changed in every file, when, and by whom. It lets you branch off to try an idea, or build a feature without disturbing stable code, merge branches back together with full visibility into conflicts, and review each other's contributions before they reach the shared codebase. The design principle is simple: never lose work, and never wonder what happened.

EVERY TEAM that ships software uses version control. It is not an optional workflow preference; it is the infrastructure that makes collaborative, iterative development possible at any scale. As projects grows, interface contracts, clear agreements on inputs, outputs, and responsibilities between components, will become increasingly important. Version control lays the foundation by making every change visible and reviewable, so contracts can be enforced rather than assumed.

THIS SECOND BLOCK introduces Git and GitHub as the version control layer for PixelWise. By the end you will have

- understood the Git data model and why it makes history tamper-proof,
- practised the staging workflow of editing, staging, and committing changes,
- created branches, merged them, and resolved conflicts,
- pushed code to a remote repository on GitHub and collaborated through pull requests,
- and configured SSH-based authentication so Git operations are seamless from both VMs.

Optional Exercise: Catch Up to Block 1's End State

This block builds on the state at the end of Block 1: Two virtual machines, dev and prod, with key-based SSH from dev to prod and password authentication disabled. If you joined late or your VMs are in an unknown state, restore the snapshots rather than redoing every prior exercise.

SNAPSHOT MAP. `B1-complete` is the snapshot taken at the end of Block 1 on both VMs. From Block 2 onwards, repository state is also versioned with Git tags, the first being `v0.1` at the end of this block.

RESTORE THE SNAPSHOT. Open VirtualBox Manager, select the `dev` VM, and restore the `B1-complete` snapshot. Then select the `prod` VM and restore its `B1-complete` snapshot as well. Both machines now sit at the exact state in which Block 1 ended.

VERIFY THE NETWORK AND SSH. From a terminal on `dev`, confirm that `prod` is reachable and that the key-based login still works:

```
ping -c 3 192.168.56.11
ssh produser@192.168.56.11
```

The ping should succeed and the SSH login should complete without a password prompt. If either fails, replay Block 1's networking and SSH exercises before continuing.

You are now at the state where Block 1 ended, with both VMs aligned, ready to start Block 2.

WITHOUT A SNAPSHOT. If you never took `B1-complete`, replay the Block 1 exercises end to end: Provisioning the two VMs, configuring host-only networking with static IPs, generating an Ed25519 key on `dev`, copying it to `prod` with `ssh-copy-id`, and disabling password authentication on `prod`'s `sshd`.

The Git Data Model

GIT IS NOT A BACKUP TOOL. It is a content-addressable filesystem with a version history layered on top. Understanding the data model makes every command intuitive.

Git was created by Linus Torvalds in 2005 to manage the Linux kernel source code. It is now the de facto standard for version control in software development.
Git: `https://git-scm.com/`
Pro Git book (free): `https://git-scm.com/book/en/v2`

Blob A file's contents, stored by its SHA-1 hash. The name is irrelevant; only the content matters.

Tree A directory listing: maps filenames to blobs (files) or other trees (subdirectories).

Commit A snapshot of the entire project at a point in time. Contains: a pointer to the root tree, the author, a timestamp, a message, and a pointer to the parent commit(s).

Branch A lightweight, movable pointer to a commit. `main` is a branch. Creating a branch is instantaneous; it just creates a new pointer.

HEAD A pointer to the branch you are currently on. When you commit, `HEAD` moves forward.

EVERY COMMIT is identified by a SHA-1 hash, a 40-character hexadecimal string like `a3f2b7c...`. This hash is computed from the commit's contents. If anything changes, even a single character in a single file, the hash changes. This makes Git history tamper-proof.

DATA HYGIENE follows directly from the data model. Because Git stores every version of every blob permanently, committing a large dataset means carrying it in the repository history forever, even if you delete the file later. A 500 MB training set committed once, modified twice, and then removed still occupies 1.5 GB of history. Repositories should contain code, configuration, and small metadata files. Large or frequently changing binary assets, datasets, and model weights do not belong in Git.

The `.gitignore` file, covered in its own subsection below, is the practical mechanism for keeping these artefacts out of the repository.

DEDICATED TOOLS fill the gap. Git LFS replaces large files with lightweight pointers. Hugging Face Hub hosts datasets and model weights with built-in Git LFS. DVC tracks data pipelines and links datasets to experiments. Weights & Biases versions datasets alongside experiment logs.
Git LFS: `https://git-lfs.com`
Hugging Face Hub: `https://huggingface.co`
DVC: `https://dvc.org`
W&B: `https://wandb.ai`

Exercise: Setting Up Git and the Repository

CREATE A GITHUB ACCOUNT if you do not have one. Then open a terminal on the `dev` VM.

INSTALL GIT. Ubuntu does not ship with Git by default, so install it through the package manager. It does not matter which directory you run this in, `apt` installs system-wide:

```
sudo apt update
sudo apt install -y git
git --version              # confirm the install succeeded
```

CONFIGURE YOUR GIT IDENTITY:

```
git config --global user.name "Your Name"
git config --global user.email "your@email.com"
```

GENERATE AN SSH KEY. SSH keys authenticate you to GitHub from the command line, replacing the password prompt on every push and pull. Before generating, check whether a key already exists so you do not overwrite one you rely on for another server:

```
ls ~/.ssh/
```

If you see a file called `id_ed25519`, that path is already in use. Either reuse it for GitHub as well, or generate a new key under a distinct filename so both keep working. Otherwise, generate the default key pair on the `dev` VM:

```
ssh-keygen -t ed25519 -C "your@email.com"
```

The tool prompts for a file location. If `~/.ssh/id_ed25519` does not yet exist, press Enter to accept the default; if it does, type a distinct path such as `~/.ssh/id_ed25519_github` so the existing key is preserved. It then prompts for a passphrase, optional but recommended, it encrypts the private key on disk so a stolen key file is not immediately usable.

ADD THE PUBLIC KEY TO GITHUB. Print it to the terminal and select the entire output, beginning with `ssh-ed25519` and ending with the comment:

```
cat ~/.ssh/id_ed25519.pub
```

AT THE END OF THE SESSIONS, take a snapshot named `B2-complete`, your safety net for the next block.

OPENING A TERMINAL ON UBUNTU. The fastest way is the keyboard shortcut `Ctrl+Alt+T`, which launches the default terminal emulator. Alternatives: press the `Super` key (the Windows key) and type "terminal", or right-click on the desktop and choose "Open Terminal". Inside a terminal you can open another tab with `Ctrl+Shift+T` and a fresh window with `Ctrl+Alt+T` again.

`sudo` runs the command with administrator privileges, which `apt` needs to write to system directories. You will be asked for your password the first time. `apt update` refreshes the package index so you get the current version; `apt install -y` installs without prompting for confirmation.

RINGS A BELL? You generated an `id_ed25519` key in Block 1 to log in to the `prod` VM without a password. That same key is sitting in `~/.ssh/` now, which is exactly why `ls` matters before the next step: accepting the default path would overwrite the production key and lock you out of the server.

MULTIPLE SSH KEYS? If you keep one key for production servers and another for GitHub, tell SSH which to use for each host with an `~/.ssh/config` entry:

```
Host github.com
  HostName github.com
  User git
  IdentityFile
~/.ssh/id_ed25519_github
  IdentitiesOnly yes
```

Then `chmod 600 ~/.ssh/config`. Without `IdentitiesOnly`, SSH may offer the wrong key first and GitHub will reject the connection after a handful of auth failures.

In GitHub, navigate to Settings, then SSH and GPG keys, then New SSH key. Give it a recognisable title like `pixelwise-dev-vm` so you know which machine it belongs to, paste the public key into the Key field, and save.

What did just happened?
`ssh-keygen` created two files in `~/.ssh/`. `id_ed25519` is the private key: never share it, never commit it, never copy it to another machine without good reason. `id_ed25519.pub` is the public key, safe to share, this is the file you paste into GitHub. The `-t ed25519` flag picks a modern signature algorithm; the `-C` comment labels the key so you can recognise it later when you have several.

Test the connection:

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

The first time, SSH asks whether to trust GitHub's host fingerprint. Type `yes`. If the key is set up correctly, GitHub responds with `Hi <username>! You've successfully authenticated, but GitHub does not provide shell access.` The message about shell access sounds like an error but is the intended response: GitHub only allows Git operations over SSH, not interactive shells.

Initialise the PixelWise repo on the `dev` VM. Pick a stable working directory, your home folder is the natural place, so the project does not get lost in `/tmp` or buried under `Downloads`:

`cd ~` jumps to your home directory from anywhere. `pwd` prints the current path so you can confirm where you are before running `git init`. Avoid initialising a repo inside `/tmp`, on a mounted USB drive, or inside another existing Git repository.

```
cd ~                          # start from your home directory
mkdir pixelwise && cd pixelwise
pwd                           # confirm: /home/<user>/pixelwise
git init
```

Write a `README.md` with a one-line project description.

The Staging Workflow

GIT HAS THREE AREAS:

- *Working directory* The files on disk. What you edit.
- *Staging area* What you have marked for the next commit. A deliberate selection.
- *Repository* The committed history. Permanent, immutable snapshots.

THE NAPKIN CYCLE as easy as it gets:

```
git pull
git add <files>
git commit -m "..."
git push
```

Pull the latest changes, stage your work, commit it, push it. Every other command is a variation on this loop.

THE CORE COMMANDS:

```
git status              # what changed? what is staged?
git add file.py         # stage a file for the next commit
git add .               # stage everything (use with care)
git commit -m "message" # commit the staged changes
git diff                # unstaged changes vs last commit
git diff --staged       # staged changes vs last commit
git log                 # show commit history
git log --oneline       # compact view: one line per commit
```

Write commit messages in the imperative mood: "Add classifier module" not "Added classifier module." The message completes the sentence "This commit will ...". Keep the first line under 72 characters. For a gallery of how not to do it, visit https://whatthecommit.com.

WHEN TO COMMIT? Commit when you have completed one logical change: a bug fix, a new function, a config update. If you struggle to write a short commit message, you probably changed too many things at once. If you go hours without committing, you are accumulating risk. Small, frequent commits make history useful; large, rare commits make it a wall of text.

```
(\ /)
(O.o)
(> <)
```

Bunny approves these changes.

The `.gitignore` File

NOT EVERYTHING BELONGS IN VERSION CONTROL. The `.gitignore` file tells Git which files and directories to exclude from tracking:

```
# Python
__pycache__/
*.pyc
.venv/

# Secrets
.env

# Data and models (too large, reproducible)
data/
models/*.pkl

# Editor files
.vscode/
*.swp
```

This includes all credentials: API keys, database passwords, webhook secrets, tokens for AI services. These live on the server in environment variables or protected configuration files, never in the repository. A leaked production database password or API key can be exploited within minutes by automated scanners.

Git history is permanent. A secret committed once is leaked forever, even if you delete the file in a later commit. The old commit still contains it. `.gitignore` is your first line of defence.

Exercise: Staging, Committing, and .gitignore

WRITE A `.gitignore`. Create a `.gitignore` file that excludes Python caches, virtual environments, secret files, dataset directories, model checkpoints, and editor files. Verify it works: create a file called `scratch.pyc`, run `git status`, and confirm Git does not list it. Then remove the test file.

MAKE YOUR FIRST COMMIT. Stage and commit `README.md` and `.gitignore`:

```
git add README.md .gitignore
git commit -m "Add project README and .gitignore"
```

Run `git log` to see your first commit. Run `git status` to confirm the working directory is clean.

THE HABIT TO BUILD: after every commit, run `git status` and `git log --oneline` to confirm what just happened. It takes two seconds and prevents surprises.

Branching and Merging

BRANCHES LET YOU WORK ON IDEAS IN ISOLATION. The `main` branch is the stable baseline. Feature branches diverge, evolve, and merge back when ready.

```
git branch                  # list branches
git branch feature-x        # create a new branch
git checkout feature-x      # switch to it
git checkout -b feature-y   # create and switch in one step
git branch -d feature-x     # delete after merge
```

Delete branches after merging. Stale branches accumulate fast and make `git branch` unreadable. If the branch is merged, it is safe to delete; the history lives on in `main`.

WHEN THE FEATURE IS READY, merge it back and clean up:

```
git checkout main
git merge feature-x
git branch -d feature-x
```

Merge Conflicts

A MERGE CONFLICT occurs when two branches changed the same line in the same file. Git cannot decide which version to keep, so you must resolve it manually.

Edit the file to keep the correct version and remove the markers. Then stage the resolved file with `git add file.py` and commit with `git commit -m "Resolve merge conflict in file.py"`.

GIT MARKS THE CONFLICT in the file. The block between `<<<<<< HEAD` and `=======` is what the current branch has; `HEAD` always means "the commit you are on right now." The block between `=======` and `>>>>>> feature-x` is what the incoming branch has.

```
<<<<<<< HEAD
result = model.predict(x)
=======
result = classifier.classify(x)
>>>>>>> feature-x
```

Exercise: Branching, Merging, and Conflicts

CREATE A FEATURE BRANCH and make a change to `README.md`:

```
git checkout -b feature-readme
nano README.md          # add a project description line
git add README.md
git commit -m "Add project description to README"
```

SWITCH BACK TO `main` and edit the same line differently:

```
git checkout main
nano README.md          # edit the same line, but differently
git add README.md
git commit -m "Add alternative description to README"
```

MERGE AND RESOLVE THE CONFLICT. Now merge the feature branch and observe the conflict:

```
git merge feature-readme
```

Git reports a conflict and pauses the merge. Open `README.md` with `nano README.md`. You will see a block that looks like this:

```
<<<<<<< HEAD
... your text on main
=======
... text from feature-readme
>>>>>>> feature-readme
```

`HEAD` marks the version sitting on the branch you are currently on, in this case `main`. The lines after `=======` are what `feature-readme` is bringing in. Choosing the correct version means deciding what the file should look like once both changes are reconciled. You have three options: keep only the `HEAD` side, keep only the incoming side, or write a merged version that combines parts of both. Then delete all three marker lines, `<<<<< HEAD`, `=======`, and `>>>>> feature-readme`, so the file contains only the resolved content. Save with `Ctrl+O` and exit with `Ctrl+X`, then complete the merge:

```
git add README.md
git commit -m "Resolve merge conflict in README"
```

Verify with `git log --oneline --graph` that both branches are now integrated.

EDITING FILES IN THE TERMINAL. `nano README.md` opens the file in a simple terminal editor. The bar at the bottom of the screen lists the shortcuts, where ^ stands for the `Ctrl` key: `Ctrl+O` writes the file to disk, `Ctrl+X` exits, and `Ctrl+G` shows full help. Vim and VS Code over Remote SSH work just as well if you prefer them.

`main` OR `master`? Older Git versions called the default branch `master`. Since 2020, the convention has shifted to `main`, and GitHub creates new repositories with `main` by default. If `git checkout main` reports that the branch does not exist, run `git branch` to see what your repo actually calls it. To bring an existing repository in line with the new convention, rename in place with `git branch -m master main`, then set the default for future `git init` runs with `git config --global init.defaultBranch main`.

`git status` DURING A MERGE. While the merge is paused, `git status` reports "You have unmerged paths" and lists every file with conflict markers still in it. Run it any time to remind yourself which files still need to be resolved before you can finish the merge.

Tags

A TAG MARKS A POINT IN HISTORY THAT MATTERS: a release, a milestone, a known-good state.

```
git tag v0.1                          # lightweight tag
git tag -a v0.1 -m "initial setup"    # annotated tag (preferred)
git push --tags                       # push tags to remote
```

Lightweight tags are just a name pointing to a commit. Annotated tags store the tagger, date, and message; use them for releases. We tag `v0.1` today; the tag becomes meaningful when you look back at it in Block 8.

SEMANTIC VERSIONING gives tag names a shared meaning. A version number follows the pattern `MAJOR.MINOR.PATCH`. Increment `PATCH` for bug fixes that do not change behaviour, `MINOR` for new features that remain backwards-compatible, and `MAJOR` when you introduce breaking changes. A jump from `v1.2.3` to `v1.3.0` tells users that something was added but nothing they depend on was removed. A jump to `v2.0.0` warns them to check for incompatibilities.

`tig` is a terminal-based Git browser that makes it easy to navigate commits, branches, and tags on a headless Ubuntu machine. Install with `sudo apt install tig`.
https://jonas.github.io/tig/

Semantic Versioning: https://semver.org

Exercise: Tags and History Browsing

TAG THE RELEASE. Mark this point in history:

```
git tag -a v0.1 -m "initial server setup"
```

Verify the tag exists with `git tag -l` and inspect it with `git show v0.1`.

WHY NO `git push` YET? Tags live locally until you push them. We have not configured a remote yet, that happens in the next exercise. Once the remote exists, `git push --tags` ships every local tag to GitHub, where it appears under the repository's "Releases" tab.

INSTALL `tig`, a terminal-based Git browser:

```
sudo apt install -y tig
tig
```

Navigate the commit history and find your tag. Press `q` to quit.

VS CODE can also browse Git history visually. Open the PixelWise folder via `code .` or Remote SSH, then use the Source Control panel and the GitLens extension to explore commits, branches, and tags with a graphical interface.

Remotes and GitHub

```
git clone <url>              # copy a remote repo locally
# fork: copy repo to your GitHub account (GitHub UI, not a git command)
git remote add origin git@github.com:user/pixelwise.git
```

A REMOTE is a copy of your repository on another machine, usually GitHub.

CLONE AND FORK are two ways to start from an existing repository.

A `fork` is a GitHub concept, not a Git command. It creates a full copy of someone else's repository under your own GitHub account. You clone your fork locally, work on it freely, and propose changes back to the original via a pull request. Forking is the standard workflow for contributing to open-source projects where you do not have write access to the original repository.

`git clone` copies a remote repository to your local machine, keeping the original as the `origin` remote. You can pull updates and, if you have permission, push changes back.

GitHub is not Git. Git is the version control system; GitHub is a hosting platform that adds pull requests, issues, CI/CD, and collaboration features on top of Git. Alternatives include GitLab and Bitbucket.
GitHub: https://github.com/
GitHub Docs: https://docs.github.com/

Pull Requests

PUSH sends your local commits to the remote. Until you push, your work exists only on your machine.

FETCH does the opposite: it downloads new commits from the remote but leaves your working directory untouched, so you can inspect what changed before merging.

PULL combines fetch and merge in one step, updating your local branch to include the remote's latest commits.

```
git push -u origin main     # push and set upstream
git pull                     # fetch + merge from remote
git fetch                    # fetch without merging
```

THE `-u` FLAG links your local branch to the remote branch. After running `git push -u origin main` once, Git remembers the connection and future `git push` and `git pull` work without specifying the remote or branch.

A PULL REQUEST is a proposal to merge a branch into another branch, typically a feature branch into `main`. It is the standard mech-

anism for code review: your changes are visible, reviewable, and discussable before they become part of the main codebase.

Pull requests and CI. Because a PR represents a well-defined set of changes that has not yet reached `main`, it is the ideal moment to run a test suite, a linter, or any other check automatically. If the tests fail, the merge is blocked and `main` stays healthy. We will wire up this automation in a later block; for now, recognise that the PR is where human review and machine verification meet.

Exercise: Remotes, Push, and Clone

CONNECT TO GITHUB. Create a repository on GitHub, add it as a remote, push the commits, then push the tag you created in the previous exercise:

```
git remote add origin git@github.com:user/pixelwise.git
git push -u origin main
git push --tags
```

Verify on GitHub that your commits, `.gitignore`, and the `v0.1` tag under the "Releases" tab are all visible.

CLONE TO THE SERVER. SSH into the server and clone PixelWise to its permanent home:

```
sudo mkdir -p /opt/pixelwise
sudo chown produser:produser /opt/pixelwise
git clone https://github.com/user/pixelwise.git /opt/pixelwise
```

PixelWise now lives on both machines. Run `git log --oneline` on the server to confirm the full history arrived.

REPLACE `user` WITH YOUR GITHUB USERNAME. The `user` placeholder in the remote URL is just a stand-in. Substitute your own GitHub handle, the one shown in your profile URL at `github.com/<your-handle>`, so the URL points at the repository under your account. The same goes for the `git clone` URL on the server below. Pushing to a `user/pixelwise` that you do not own will fail.

SSH INTO `prod` AGAIN. From the `dev` VM, open a session to `prod` the same way you did in Block 1:

```
ssh produser@192.168.56.11
```

The key-based login from Block 1 is still in place, no password should be required. Once you are on `prod`, the prompt changes to `produser@prod`, and the commands below run on the server. Type `exit` when you are done to return to `dev`.

WHY HTTPS ON `prod`, SSH ON `dev`? `dev` pushes commits back to GitHub, so it needs the SSH key you registered earlier. `prod` only ever pulls, it is a deployment target, never an authoring machine. HTTPS clone of a public repository requires no key, no token, no extra setup, which matches how real deployment servers usually have read-only access to the source repository. If your repository is private or you later need to push from `prod`, generate a second SSH key on `prod` and register it on GitHub the same way you did on `dev`.

Self-Reflection and Recap

Reflection questions:

- Why does changing a single character in a file produce a completely different commit hash?
- What is the difference between `git add` and `git commit`, and why does Git separate the two steps?
- When would you use a branch instead of committing directly to `main`, and what happens when a merge produces a conflict?
- Why should you never commit `.env` files or API keys, and what is the role of pull requests in catching mistakes before they reach `main`?
- Why does PixelWise use SSH keys rather than passwords for Git operations across both VMs?

Milestone: Your project lives on GitHub with a README, a `.gitignore`, and a tagged release. The server has the repo cloned at `/opt/pixelwise`. Both machines share the same history. In the next block we turn the manual package installs into a reproducible setup script and tackle dependency management.

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`).

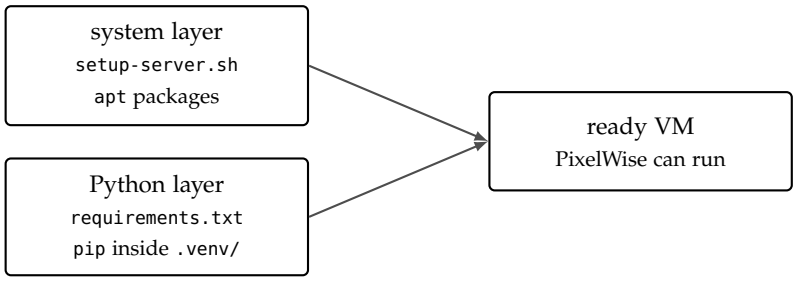

Figure 4: Two layers, two files. `setup-server.sh` provisions the operating system; `requirements.txt` provisions Python inside an isolated virtual environment. Together they reconstruct the environment from scratch on any fresh VM. System packages, the things you install with `apt`, are captured in 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
```

Verify the switch. `which python` should print a path inside `.venv/bin/`. `pip list` should show only `pip` and `setuptools`:

```
which python
pip list
```

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.

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.

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.

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?

(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.

Commit the requirements:

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

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

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.

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.

```
# 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
```

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.

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.

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 in 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?

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.

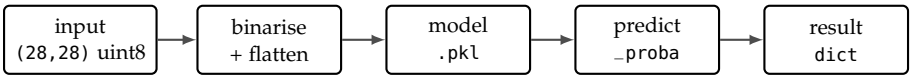

Figure 5:
The inference pipeline for one image: the caller binarises a raw `(28,28)` `uint8` array at threshold 128 and flattens it into a `(1,784)` row vector before feeding it to the loaded `.pkl` model. The model's internal `Binarizer` is a redundant idempotent guard on already-binarised input, and `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.

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.

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.

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.

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.

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",
  )
```

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`

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.

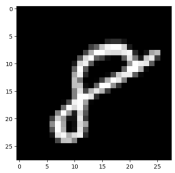

Figure 6: An MNIST sample: a handwritten digit on a 28×28 greyscale grid, the input shape every consumer of `digit_classifier_v1` must respect.

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_)
```

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}")
```

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.

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 os 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.

The Backend: APIs & Services

2026-05-10 · happy pear Nachtigall

API First. Anyone who doesn't do this, will be fired.

The Why

Block 4 left us with `app/classifier.py`, a function that turns a 28-by-28 array of pixels into a prediction dictionary. The function works. `predict.py` proves it on real MNIST samples. What it cannot do is be reached from anywhere except the same Python process that imported it.

This Block. is the first that exposes the application to the outside world. Every client that follows, whether a browser frontend, an automated test runner, or a monitoring probe, talks to the service through the contract you design here.

HTTP is the protocol that every browser, every CLI tool, and every web service in this course speaks. Figure **??** shows the path a single request takes through the stack you will build in this Block.

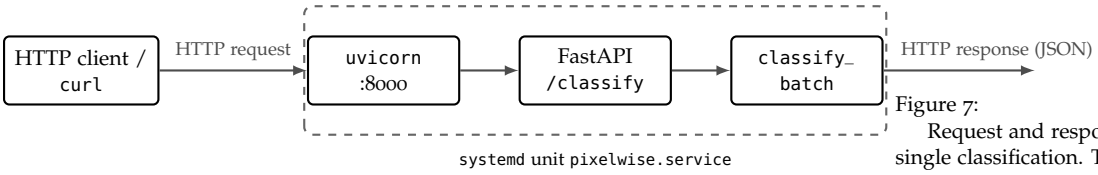

Figure 7: Request and response flow for a single classification. The client sends an HTTP request to `uvicorn`; FastAPI dispatches it to the `/classify` handler, which calls `classify_batch` from Block 4. The dictionary returned by `classify_batch` is serialised to JSON and travels back along the same path as the HTTP response. The whole stack runs under a `systemd` unit that restarts it if it crashes.

A service is what `predict.py` becomes when you wrap it in HTTP and run it under a supervisor. `uvicorn` is the HTTP server that hosts the FastAPI application; `systemd` is the supervisor that keeps `uvicorn` running, restarts it when it crashes, and captures its logs, turning it into someting that survives.

Hands On Experience

THE CONTRACT BECOMES THE PRODUCT. Once the service [1] is running, the request and response shapes are what every other component in the system depends on. Change them silently and the frontend breaks; document them clearly and new clients can integrate without ever reading your Python. Ones an API is public, it will be digested, users will build on it, and it will be hard to change [2].

THIS FIFTH BLOCK introduces the backend service layer for PixelWise. By the end you will have

- understood HTTP and REST well enough to design and probe an endpoint,
- written request and response contracts as Pydantic models,
- wrapped `classify_batch` in a FastAPI application,
- added API key authentication and basic rate limiting,
- run the service under `systemd`, observed crashes, and read its logs with `journalctl`.

[1] T. Winters, T. Manshreck, and H. Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, Sebastopol, CA, 2020. URL https://scholar.google.com/scholar?q=Software+Engineering+at+Google+Winters+Manshreck+Wright

[2] F. P. B. Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley, Reading, MA, 2 edition, 1995. URL https://scholar.google.com/scholar?q=The+Mythical+Man-Month+Brooks

THE SCRIPT-TO-SERVICE JUMP. A script runs once, prints, exits. A service runs continuously, accepts requests, returns responses, survives crashes. The work in this block is what turns the first into the second; the classifier itself does not change.

Optional Exercise: Catch Up to `v0.3`

CATCHING UP VIA TAG. This catch-up extends the one in Block 4. If you have not done that one, do it first: It lands you at `v0.2`, the end-of-Block 3 state. The exercise here picks up from there and rewinds to `v0.3`, the end-of-Block 4 state, with the classifier module in place, the model artefact pulled into `models/`, and `.env` pinned to the model release.

TAG MAP. `v0.3` marks the end of Block 4, `v0.4` will mark the end of this block. Browse the full set at `https://github.com/schutera/pixelwise/tags`.

DEV ONLY. Block 4 deliberately left `prod` behind: The classifier had no HTTP entry point, so pulling the code there would have added an unused `app/classifier.py` to the production checkout. This catch-up does the same. Bring `dev` to `v0.3` and leave `prod` where Block 4 left it; the `systemd` exercise later in this block is what finally promotes the service to `prod`.

THREE LAYERS TO RESTORE ON `dev`. Block 4 produced Git-tracked changes, the new `app/classifier.py` and `predict.py`, an updated `setup-server.sh`, and a refreshed `.env.example`. It also produced unversioned state, the model artefact `digit_classifier_v1.pkl` in `models/`, plus three new `.env` entries, `MODEL_REPO`, `MODEL_VERSION`, and `MODEL_PATH`. `git reset --hard v0.3` restores the first layer, `cp .env.example .env` seeds the third, and the extended `setup-server.sh` pulls the artefact once `.env` is in place.

RUN THE CATCH-UP ON `dev`. Rewind the repository to `v0.3`, refresh `.env` from the updated example, run `setup-server.sh` to pull system packages and the model artefact, reinstall pinned dependencies if your virtual environment was wiped, and run `predict.py` to confirm the integration works end to end. The exact commands are in the margin.

SEQUENCE ON `dev`.

```
cd ~/pixelwise
git fetch origin --tags
git reset --hard v0.3
cp .env.example .env
bash setup-server.sh
source .venv/bin/activate
pip install -r requirements.txt
python predict.py
```

If `.venv/` is missing, run `python3 -m venv .venv` before the `source` step.

YOU ARE NOW at the state where Block 4 ended: `dev` is at `v0.3` and `prod` still trails behind, ready to start Block 5.

ALREADY HAVE A FORK? If your own fork carries the `v0.3` tag, swap `origin` on `dev` with `git remote set-url origin git@github.com:<you>/pixelwise.git` so push and pull continue to target your account. The HTTPS clone on `prod` stays read-only.

HTTP and REST

HTTP is the request and response protocol that sits underneath every browser call, every `curl` invocation, and every API in this course. A reference overview of the methods, status codes, and headers lives here.

EVERY WEB INTERACTION IS A REQUEST AND A RESPONSE. The client sends a request that names a method, a path, and an optional body; the server sends back a status code, headers, and an optional body. The protocol carries no state of its own: each request is interpretable on its own, without knowledge of any previous request from the same client.

METHODS AND QUERIES YOU WILL USE.

- `GET` retrieves,
- `POST` submits,
- `PUT` replaces,
- `DELETE` removes.

PixelWise uses `POST` for `/classify` when the client submits a drawing, and `GET` for `/results` and `/health` when the client reads state.

STATUS CODES YOU WILL SEE MOST. Three-digit numbers grouped by leading digit. `2xx` means success, `4xx` means the client made a mistake, `5xx` means the server made a mistake.

- `200` OK,
- `201` Created,
- `400` Bad Request,
- `401` Unauthorized,
- `404` Not Found,
- `422` Unprocessable Entity,
- `429` Too Many Requests,
- `500` Internal Server Error.

STATELESS is the load-bearing property of HTTP. The server keeps no per-client memory between requests; everything the server needs to act is in the current request, including any authentication token. This is what lets you scale.

REST PRINCIPLES layer convention on top of the protocol. Every distinct thing the API exposes is a resource with a stable URL: `/classify`, `/results`, `/health`. Methods do what their names suggest. Responses are JSON. Two clients written by two teams, six months apart, agree on what the API does without ever speaking, because the conventions remove ambiguity.

REST is the architectural style that maps the HTTP verbs onto application resources. Each resource has a stable URL; each verb has a clear meaning. We use REST because it is the convention every frontend, every tool, and every framework in this stack expects, not because it is the only valid choice. A reference introduction to the style and its conventions lives here.
HTTP status codes reference:
`https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`
REST overview:
`https://restfulapi.net/`

THE INTERACTIVE DEFINITION OF AN API is the documentation that ships with the running service. FastAPI generates an OpenAPI specification from your code, exposes the raw JSON at `/openapi.json`, and renders it through two built-in interfaces: Swagger UI at `/docs` and ReDoc at `/redoc`. Exposing them changes the social contract: A teammate does not need to read your source to integrate, it's your responsibility to fulfill the stated contract. Block 3's `DEBUG=true` flag controls whether the documentation endpoints are exposed, so the same service is self-documenting in development and silent in production.

ONE SPEC, MANY RENDERERS. `/openapi.json` is the contract; Swagger UI and ReDoc are two ways to look at it. The same spec format is what Flask, Django, Express, and Spring all emit, and the same renderers and SDK generators work against any of them.
OpenAPI: `https://www.openapis.org/`
Swagger Editor: `https://editor.swagger.io/`

Exercise: HTTP, REST, and `curl`

PROBE A PUBLIC API before writing your own. `httpbin.org` is a free request-and-response echo service: It returns a JSON document describing exactly what it just received, so you can read status codes, headers, and the body of your own request side by side with the response. On `dev`, with the virtual environment active:

```
curl -i https://httpbin.org/get
curl -i -X POST https://httpbin.org/post \
    -H "Content-Type: application/json" \
    -d '{"hello": "world"}'
```

READ EACH SECTION OF THE RESPONSE. `HTTP/2 200` is the status line. The lines that follow are headers; the blank line marks the boundary; the JSON below it is the body. Every HTTP response in this course has the same shape.

THE BODY IS JSON. JSON is the universal serialisation format for web APIs: A text-shaped tree of objects, arrays, strings, numbers, booleans, and `null`. Your `GET` response body is short enough to read in full:

```
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.81.0"
  },
  "origin": "203.0.113.42",
  "url": "https://httpbin.org/get"
}
```

JSON SYNTAX Objects `{...}`, arrays `[...]`, strings in double quotes, numbers and booleans and `null` unquoted, commas between items but never trailing. Spec: `https://www.json.org/`

Curly braces enclose an object, each key is a quoted string, and each value is a primitive, an array, or another object. PixelWise's `/classify` response will use the same shape, with keys `prediction`, `confidence`, and `scores`, parsed identically by any client in any language.

PICK THE VERB for each PixelWise endpoint and say what success looks like:

- `/classify`: Submit a drawing for prediction.
- `/results`: Read the last 10 predictions.
- `/health`: Check whether the service is alive.

ANSWERS. `POST /classify`: Image data does not fit in a query string; `POST` carries it in the body and is not cached. `200 OK` with the prediction.

`GET /results`: Safely cacheable, no side effect. `200 OK`.

`GET /health`: Reads liveness, no side effect. `200 OK`, or `503 Service Unavailable` when unhealthy.

FastAPI in Practice

FastAPI is a modern Python web framework built on top of Pydantic and the ASGI standard. It generates interactive API documentation automatically, validates inputs and outputs against the models you declared, and runs on `uvicorn`, an asynchronous HTTP server fast enough that your bottleneck will be the model, not the framework.

Pydantic. Python data-validation library. A class inherits from `BaseModel` and declares typed fields; Pydantic validates incoming data against the schema and raises a structured error on mismatch. FastAPI runs the same check at the HTTP boundary and returns `422 Unprocessable Entity` on failure.
https://docs.pydantic.dev/

FastAPI: https://fastapi.tiangolo.com/
Pydantic: https://docs.pydantic.dev/
uvicorn: https://www.uvicorn.org/
ASGI: https://asgi.readthedocs.io/

A path operation is the unit of routing in FastAPI: A Python function whose decorator binds it to one HTTP method and one URL path. When the application starts, FastAPI walks the decorated functions and builds a routing table that `uvicorn` consults on every incoming request. The type hints on the parameters are not just documentation: They are read at registration time and turned into a validator that runs before your function body executes. A request that fails the validator never reaches your code. It is rejected at the boundary with a `422` response whose body lists exactly which field was wrong and why, so the caller can fix the request without guessing at your implementation.

The decorator is the contract. `@app.post("/classify", response_model=ClassifyResponse)` encodes three things at once: The method clients must use, the path they must hit, and the schema the response must satisfy. The first two shape the routing table. The third shapes what leaves the server. FastAPI runs the value your function returns through `ClassifyResponse` on the way out, drops fields that are not declared on the model so internal state cannot leak by accident, and fills the OpenAPI schema served at `/docs` from the same class. A handler that returns the wrong shape fails inside the framework on the first request, long before a downstream client notices that the documented contract has drifted from the implementation.

`curl` is a command-line HTTP client. It writes the bytes of a request directly to a socket and prints the bytes of the response,

with no browser, no JavaScript, and no implicit defaults beyond what you pass on the command line. That is what makes it useful for debugging: The request you send is exactly the request you typed, so the only place a bug can hide is in your invocation or in the server. The interactive `/docs` page, by contrast, does a lot on your behalf. It JSON-encodes the body for you, sets the `Content-Type: application/json` header on every `POST`, and, once you click `Authorize`, attaches the API key header to subsequent calls. A request that succeeds from `/docs` and fails from `curl` therefore points at something the form was adding silently, almost always a header you forgot on the command line. A request that fails from both rules out the client and points back at the handler. The exercise below gives the exact `curl` invocations against the running service so you can compare the two side by side.

curl: `https://curl.se/`
`-X` sets the HTTP method.
`-H` sets a header.
`-d` sets the request body.
`-i` prints the response headers alongside the body, useful when a status code surprises you.

THREE ENDPOINTS cover the surface this block exposes: `POST /classify` performs inference, `GET /results` is a stub that a later persistence layer will fill in, and `GET /health` reports liveness so a monitor can tell whether the service is up.

Exercise: Building `app/main.py`

INSTALL THE RUNTIME DEPENDENCIES into the dev virtual environment and pin them.

```
cd ~/pixelwise
source .venv/bin/activate
pip install fastapi uvicorn[standard] \
    slowapi pydantic
pip freeze > requirements.txt
```

THE `[standard]` EXTRA on `uvicorn` pulls in optional but recommended packages that ship alongside the core server: `httptools` for a faster HTTP parser, `uvloop` for a faster event loop on Linux, `websockets` for the WebSocket protocol, and `watchfiles` so `--reload` can detect code changes during development. Without the extra you still get a working server, just slower and without auto-reload.

WIRE THE PATH OPERATIONS FROM THE THEORY ABOVE INTO A WORKING SERVICE. From the repository root on `dev`, create the file:

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

CTRL + R IN THE TERMINAL starts a reverse search through your shell history. Hit it, type a few characters of a command you ran earlier, such as `pip freeze`, and the most recent match appears. Press Ctrl + R again to step further back, Enter to run the match, or the right arrow to edit it before running. This is much faster than retyping or looking up commands again or scrolling with the up arrow.

```
from fastapi import FastAPI
from pydantic import BaseModel
import numpy as np
from app.classifier import classify_batch

class ClassifyRequest(BaseModel):
    pixels: list[list[int]]

class ClassifyResponse(BaseModel):
    prediction: str
    confidence: float
    scores: dict[str, float]

app = FastAPI()

@app.get("/health")
def health():
    return {"status": "ok", "model_version": "v1"}

@app.get("/results")
def results():
    return {"results": [], "note": "persistence not yet implemented"}

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

RUN IT LOCALLY from the repository root, so that `uvicorn` can resolve `app.main:app` as a Python import against the `app/` package on disk:

```
cd ~/pixelwise
uvicorn app.main:app --reload --port 8000
```

The `--reload` flag restarts on every code change, which is what you want in development and never want in production.

TEST IT FROM THE BROWSER. Open `http://localhost:8000/docs` to see the auto-generated OpenAPI page.

TRY `GET /health`. Click "Try it out" and "Execute". A `200` with `{"status": "ok", "model_version": "v1"}` is the first end-to-end loop. The first real `POST /classify` round-trip lands in Block 7, when the frontend sends an actual digit drawn on a canvas.

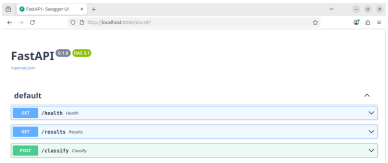

Figure 8: The Swagger UI rendered at `/docs`. FastAPI generates this page from the type hints and Pydantic models in `app/main.py`. Each endpoint expands to show its request schema, an example payload, and an interactive button that issues a live request against the running service.

OPTIONAL: TRY `POST /classify` ANYWAY. A Python list comprehension is not valid JSON, which is why the default `[[0]]` returns `422`. Print a real 28-by-28 body in a Python REPL with

```
import json
print(json.dumps(
  {"pixels":
  [[0]*28 for _ in range(28)]}))
```

and paste the printed line into Swagger's request body field.

OPTIONAL: DESIGN A FOURTH ENDPOINT. The three endpoints above cover inference, retrieval, and liveness. What is missing? Before reading on, sketch one more endpoint on paper. Pick the HTTP method and path, write the Pydantic request and response models, and decide what status code each error case returns. Plausible candidates include `GET /version` so a deployment can report the running build and model hash, `POST /feedback` so a frontend can flag wrong predictions for later retraining, or `POST /classify/batch` so a client can submit many images in a single round trip. Additional endpoints can help where the current contract has gaps, and to feel how naming, shape, and status codes are decisions you make before any code runs.

Authentication and Rate Limiting

Who is calling, and how often?

API KEY AUTHENTICATION is the simplest form that fits this course. The client sends a secret in a header; the server compares it to the value in `.env` and rejects requests that do not match with `401 Unauthorized`. FastAPI exposes the header through a small dependency function that runs before every protected handler, which the exercise wires up.

WHY A HEADER? Headers travel separately from the body and the URL, so the secret never ends up in the path component of an access log. Putting the key in a query parameter would write it into every log line the request touched on the way to the server.

RATE LIMITING prevents both abuse and accident. `slowapi` attaches a per-route decorator that counts requests per client IP and rejects them with HTTP `429` when the limit is exceeded. A budget like "30 requests per minute per IP" is enough to stop a runaway client without inconveniencing a normal one.

WHERE DOES RATE LIMITING BELONG? Application-level limits are easy to write and travel with the code. Reverse-proxy limits, a topic we will revisit later in the course, run in front of the application and protect it even when it is overwhelmed. Both layers are useful; the latter is harder to bypass.

slowapi: `https://github.com/laurentS/slowapi`

PYDANTIC, THE API KEY MIDDLEWARE, AND THE RATE LIMITER together form a three-stage filter at the boundary of the application. A request that arrives must parse as JSON, match the schema, present a valid key, and stay within the rate budget before any inference code runs. Each stage rejects with a different status code, so a client failure tells you exactly which contract was violated.

(Optional) Exercise: API Key Auth and Rate Limiting

ADD THE DEFENSIVE LAYER FROM THE THEORY ABOVE TO THE SERVICE. Reopen `app/main.py` from `~/pixelwise` on `dev`:

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

Add the API key dependency alongside the existing imports:

```
from fastapi import Header, HTTPException, Depends
import os

def verify_api_key(x_api_key: str = Header(...)):
    if x_api_key != os.getenv("SECRET_API_KEY"):
        raise HTTPException(
            status_code=401, detail="Invalid API key")
```

Then attach `verify_api_key` to the `/classify` route by extending its existing decorator with `dependencies=[...]`; the handler body itself does not change:

```
@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]
    return classify_batch(arr)[0]
```

THE HEADER TRAVELS IN EVERY REQUEST. `X-API-Key` is a custom header name; FastAPI's `Header(...)` maps it to the function parameter automatically. A missing header returns 422 from Pydantic, a wrong value returns 401 from your handler.

ADD RATE LIMITING with `slowapi`. Add the imports and the limiter setup near the top of the file:

```
from fastapi import Request
from slowapi import Limiter
from slowapi.util import get_remote_address
from slowapi.middleware import SlowAPIMiddleware

limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter
app.add_middleware(SlowAPIMiddleware)
```

Then stack the limiter decorator above the route decorator and add a `request: Request` parameter so `slowapi` can identify the caller:

```
@app.post("/classify",
          response_model=ClassifyResponse,
          dependencies=[Depends(verify_api_key)])
@limiter.limit("30/minute")
def classify(request: Request,
             req: ClassifyRequest):
    arr = np.array(req.pixels,
                   dtype=np.uint8)[np.newaxis]
    return classify_batch(arr)[0]
```

TEST `POST /classify` FROM SWAGGER. Reload `/docs`, expand the endpoint, click "Try it out", and fill the key from your `.env` into the `x_api_key` header field that now sits alongside the request body. Press "Execute". Without a key the classify request returns `401`; with the right key the same request goes through.

OPTIONAL: ADD A GLOBAL AUTHORIZE BUTTON. Swap `Header(...)` for `APIKeyHeader` from `fastapi.security` to register the API key as an OpenAPI security scheme. Swagger then renders a global "Authorize" button at the top of `/docs`; after clicking it once, every Try-it-out includes the header automatically.

OPTIONAL: WATCH A 429 HAPPEN. A small Bash loop firing 40 `curl` requests at the endpoint inside the same minute should see the last ten return `429 Too Many Requests`. The remaining requests stay rejected until the window rolls over.

From Dev to Production: `systemd`

`uvicorn` RUNS FINE, but one Ctrl+C and it is gone. A closed terminal kills it. A reboot kills it. A panic in a worker kills it without restart. We got the interface right, but none of the above is the behaviour of a robust service.

`systemd` is the OS-level supervisor that comes with Ubuntu. It starts services on boot, restarts them when they crash, captures their output to a structured log, and exposes a uniform interface for operators. The contract between you and `systemd` is a unit file, an INI-style description with three sections: An `[Unit]` block for metadata and dependencies, a `[Service]` block naming the user, working directory, environment file, command to run, and restart policy, and an `[Install]` block declaring when the unit should be started. The exercise below has the full file PixelWise needs.

WE USE A FRACTION of what `systemd` can do. It also manages timers (cron with dependency awareness), socket activation (start on connection), resource limits (cap CPU/memory per service), and dependency ordering (start the database before the app). It is the init system of virtually every modern Linux server.

systemd: `https://www.freedesktop.org/wiki/Software/systemd/`

`systemctl` is the operator's interface to `systemd`. `start`, `stop`, and `restart` are the verbs you reach for during deployment; `status` reports the current state and the last few lines of output; `enable` marks the service as one that should start on boot; `daemon-reload` tells `systemd` to re-read its unit files after you edit one. The exercise runs the full sequence on `prod`.

`journalctl` is where you go first when something breaks. `journalctl -u pixelwise` prints every log line from your service since boot; `-f` follows new lines as they arrive, the way `tail -f` follows a file; `--since "5 min ago"` narrows the window when the bug is recent.

`gunicorn` is what production Python deployments put in front of `uvicorn` to spawn multiple worker processes, giving concurrency and crash isolation. For our classroom setup, a single `uvicorn` worker behind `systemd` is sufficient; students who deploy beyond this course will encounter `gunicorn` within the first week.

Exercise: Running as a `systemd` Service

Promote the service from a foreground `uvicorn` to a supervised `systemd` unit. Author the unit file on `dev` alongside the app code first, then refer to the marginnote for exact commands.

Author the unit file on `dev` at `deploy/pixelwise.service` in the repository. Keeping it under version control means anyone can rebuild `prod` from a clean VM by cloning, pulling, and copying, instead of remembering what was typed into `nano` six months ago.

```
# deploy/pixelwise.service
[Unit]
Description=PixelWise API
After=network.target

[Service]
User=produser
WorkingDirectory=/opt/pixelwise
EnvironmentFile=/opt/pixelwise/.env
ExecStart=/opt/pixelwise/.venv/bin/uvicorn \
    app.main:app --host 0.0.0.0 --port 8000
Restart=always
RestartSec=3

[Install]
WantedBy=multi-user.target
```

Production has some catching up to do. First dev push then prod pull.

```
git add deploy/pixelwise.service
app/main.py requirements.txt
.env.example
```

```
git commit -m "Add systemd
service file and FastAPI service
with auth and limits"
```

```
git push
```

```
ssh produser@192.168.56.11
cd /opt/pixelwise
git pull
source .venv/bin/activate
pip install -r requirements.txt
```

Teach `setup-server.sh` to install the unit. Open the file in nano on `dev` and append a small block at the bottom so any future catch-up or fresh provision picks it up:

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

```
# Install, start, and report the systemd unit on prod
if [ -f deploy/pixelwise.service ] && \
   command -v systemctl >/dev/null 2>&1 && \
   id produser >/dev/null 2>&1; then
    sudo cp deploy/pixelwise.service /etc/systemd/system/pixelwise.service
    sudo systemctl daemon-reload
    sudo systemctl enable pixelwise
    sudo systemctl restart pixelwise
    sudo systemctl status pixelwise
fi
```

Why ship the unit file in the repo? The unit file is part of how the app runs, so it belongs next to the code. Diffs are reviewable, deploys are reproducible, and there are no secrets inside, since `EnvironmentFile` only points at `/opt/pixelwise/.env`, which stays on `prod`.

Prod-only by design. The block runs only when `produser` exists, which marks the prod VM. On `dev` the user is missing, so `setup-server.sh` skips the whole step instead of copying a unit that references a user who does not exist there. On `prod`, the script installs, enables, restarts, and reports the service status on every run, picking up any change to `deploy/pixelwise.service` from the latest `git pull`. The `--no-pager` flag keeps the output inline so the script does not stop.

INSTALL THE UNIT ON PROD after the pull. Run `setup-server.sh`; it copies the unit, reloads `systemd`, enables the service, restarts it, and prints the status block in one shot:

```
bash setup-server.sh
```

KILL IT ON PURPOSE. A service that has never crashed is a service whose recovery has never been tested.

```
sudo systemctl stop pixelwise
sudo systemctl status pixelwise
sudo systemctl start pixelwise
journalctl -u pixelwise --since "5 min ago"
```

READ THE STATUS OUTPUT. `Active: active (running)` is what you want. `Active: failed` means the unit started and exited; the last few log lines below the status block usually say why. `Active: inactive (dead)` means the unit was never started.

FIRST END-TO-END LOOP ON THE NETWORK. From `dev`, hit the unprotected health endpoint on prod:

```
curl http://192.168.56.11:8000/health
```

A response of `{"status":"ok","model_version":"v1"}` comes back from the `systemd`-supervised `uvicorn` on prod. No body, no key, just a cross-machine round-trip.

THE SYSTEM IS REACHABLE ACROSS MACHINES ON A SHARED NETWORK FOR THE FIRST TIME.

Self-Reflection and Recap

SELF-REFLECTION questions to guide your thoughts during and after the exercises:

- What is the difference between running `uvicorn` manually and running it under `systemd`?
- Why does Pydantic validation matter for security as well as correctness?
- What happens when the `systemd` service crashes, and how does `Restart=always` help?
- Why is `GET /results` a stub right now, and what would have to arrive to fill it in?
- How would you debug a failing endpoint using only `journalctl`?
- Which of the three rejection points (Pydantic, API key, rate limiter) is hardest to bypass, and why?

RECAP of key concepts:

- HTTP is a stateless request-response protocol; REST layers conventions on top so APIs are predictable across teams.
- Pydantic models are the API contract; FastAPI enforces them at the boundary and renders them as live documentation.
- `POST /classify`, `GET /results`, `GET /health` cover the surface this block exposes; `/results` is a deliberate stub awaiting a persistence layer.
- API keys answer "who is calling"; rate limits answer "how often".
- `systemd` turns a script into a service: starts on boot, restarts on crash, logs to `journalctl`.

MILESTONE. PixelWise runs as a managed `systemd` service on the `prod` VM, listening on `http://192.168.56.11:8000`. The endpoints answer requests from `dev` and from your host machine over the host-only adapter; the machines outside the network cannot reach the service yet. `POST /classify` returns live digit predictions, `GET /results` is stubbed pending persistence, and `GET /health` reports liveness. You have killed and restarted the backend at least once and read the logs with `journalctl`. The contract is now the load-bearing surface of your backend: Every component that follows will speak through it.

SECURITY THREAD. The API key lives in `.env`, never in code. Pydantic models are the input validation boundary; rate limits cap any single caller; the `X-API-Key` header keeps the secret out of access logs. Transport-layer encryption is a thread we will revisit later in the course.

TEASER. Where do all those predictions go once the response has been sent?

What is missing is memory. The service answers each request and forgets it the moment the response goes out. That is fine for a smoke test and broken for everything else: A user looking back at their last ten drawings has nothing to show, an operator investigating a wrong prediction cannot see what was sent, and a retraining loop cannot use real-world inputs because none of them survived. Block 6 puts PostgreSQL behind the service so every classification becomes a row, `GET /results` starts returning real history, and the service grows the audit trail the rest of the course depends on.

Databases & SQL

2026-05-10 · happy pear Nachtigall

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.

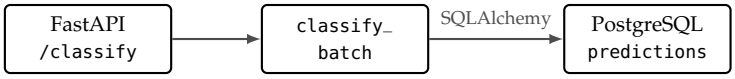

Figure 9: The persistence flow. `POST /classify` computes a prediction with `classify_batch`, the handler maps the result through the `Prediction` SQLAlchemy model, and a row lands in the `predictions` table in PostgreSQL before the response goes out.

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.

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. Schemas evolve through migrations, which we cover at the end of the block as an optional path.

DESIGNING SCHEMAS is its own discipline. We design the `predictions` table once and keep it that way for the mainline of the course. The optional Alembic exercise at the end lets you feel the cycle of design, deploy, regret, migrate on a real database.

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

- written your first SQL queries against a PostgreSQL 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,
- configured database credentials 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`.

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`.

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.

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.

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`.

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"...}`.

YOU ARE READY PLAYER ONE.

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 record must follow; the database refuses inserts or queries that violate it.

STRUCTURED QUERY LANGUAGE (SQL) is the language you use to talk to a relational database. The database is a single file, there is no server to install, no users, no network configuration. For a single-developer prototype or a desktop application, this is exactly what you want. 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;

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

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 columns that uniquely identify each row. Most tables use an auto-incrementing integer or a UUID as their key.

id	prediction	confidence	model_version
1	7	0.94	v1
2	3	0.81	v1
3	5	0.99	v1

WHAT YOU WILL MEET LATER.

- `JOIN` combines rows from multiple tables.
- `UPDATE` adjusts existing values without changing the schema.
- `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.

PostgreSQL for Production

POSTGRESQL IS A PRODUCTION DATABASE. It runs as a separate process, accepts connections over the network, has real users with real permissions, and supports concurrent reads and writes without serialising the whole file. `psql` is the command line client.

WHY A DEDICATED USER? LEAST PRIVILEGE The `postgres` superuser can drop any database, create any user, and read any table. A dedicated `pixelwise` user can only do what its privileges allow, which by default is read and write its own tables. The principle is the same as the non-root `produser` on your server. You provision a `pixelwise` role and a `pixelwise` database owned by that role inside PostgreSQL, then verify the credentials with `psql` before any Python is involved.

WHERE SQLITE STOPS SHORT? SQLite has no user authentication, no password protection, and limited concurrency - due to file locking.

PostgreSQL: `https://www.postgresql.org/`
SQLite: `https://www.sqlite.org/`

Exercise: PostgreSQL on the Server

While you could set things up manually on `prod`, it is a best practice to automate every step of the server configuration through `setup-server.sh`; you edit it on `dev`, commit, push, and `bash setup-server.sh` carries the work on both VMs. The one manual step is writing the secret `DB_PASSWORD` into the local `.env` on each VM, since secrets never travel through git. Provisioning on `dev` too means you can write `models.py`, run `init_db.py`, and inspect rows with `psql` without ever ssh'ing to `prod`.

Extend `setup-server.sh` on `dev`. Add `postgresql` to the `apt install` line, then append a stanza that provisions the `pixelwise` role and database wherever `.env` is present:

```
cd ~/pixelwise
nano setup-server.sh
```

Why `set -a`? `set -a` tells bash to export every variable defined from now on. `source .env` reads the file as shell commands, so the assignments happen in the current shell. `set +a` turns the auto-export back off. The same pattern lets a script load `.env` before running anything that depends on it.

Resolve the script directory at the top. Add this line to `setup-server.sh` right after the shebang, so every later stanza can locate the repo root regardless of the caller's working directory:

```
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" \
  && pwd)"
```

Append the provisioning stanza after the systemd block:

```
# Provision the pixelwise role and database on every VM
if command -v psql >/dev/null 2>&1 && \
   [ -f "$SCRIPT_DIR/.env" ]; then
    set -a; source "$SCRIPT_DIR/.env"; set +a
    sudo -u postgres psql -tAc \
        "SELECT 1 FROM pg_roles WHERE rolname='pixelwise'" \
        | grep -q 1 || \
    sudo -u postgres psql -c \
        "CREATE USER pixelwise \
         WITH PASSWORD '$DB_PASSWORD';"
    sudo -u postgres psql -tAc \
        "SELECT 1 FROM pg_database WHERE datname='pixelwise'" \
        | grep -q 1 || \
    sudo -u postgres createdb -O pixelwise pixelwise
fi
```

Idempotent by design. Each `psql -tAc` probe asks PostgreSQL whether the role or database already exists; the `||` short-circuits the create when the answer is yes. Run the script ten times, the role and database are still there exactly once. `SCRIPT_DIR` resolves to whichever checkout invoked the script, so the same stanza runs on `dev` and `prod`; the systemd block from Block 5 stays gated behind `id produser` since the service only ever runs on `prod`.

Document the new secret in `.env.example` on `dev`, then commit and push:

```
echo "DB_PASSWORD=" >> .env.example
git add setup-server.sh .env.example
git commit -m "Provision PostgreSQL"
git push
```

Apply on `dev` first. Write the password into your local `.env` and run the script:

```
echo "DB_PASSWORD=database" >> .env
bash setup-server.sh
```

Repeat on `prod`. Same shape, plus the `git pull` to bring in the committed script:

```
ssh produser@192.168.56.11
cd /opt/pixelwise
git pull
echo "DB_PASSWORD=database" >> .env
bash setup-server.sh
```

The first run on each VM installs `postgresql` and provisions the `pixelwise` role and database; subsequent runs change nothing because the provisioning stanza is idempotent.

Verify the connection on `prod` with the application user:

```
psql -U pixelwise -d pixelwise -h localhost
```

`pixelwise=>` prompt means the role and database are wired up. If `psql` fails to connect, fix the credentials before going any further. A working `psql` is the foundation of everything the rest of the block builds on. If it worked you know

Did it take on `dev`?

```
psql -U pixelwise -d pixelwise -h localhost
```

Enter the password from `.env`. A `pixelwise=>` prompt means the role exists, the database is yours, and the password matches. If the connection is refused, re-read the script output before moving on to `prod`.

Wipe and start fresh. The provisioning stanza skips creation when the role or database already exists, so editing `DB_PASSWORD` after a first run does not propagate. If `psql` rejects the password or reports a missing database, drop both and re-run:

```
sudo -u postgres dropdb pixelwise
sudo -u postgres dropuser pixelwise
bash setup-server.sh
```

What each flag does.

- `-U pixelwise` selects the role to log in as.
- `-d pixelwise` selects the database to connect to.
- `-h localhost` forces a TCP connection to the local machine, which makes `psql` prompt for the password from `.env` instead of trusting the Unix socket's peer authentication on the `postgres` system user.

Role and database share a name. The setup script creates both:

- A role `pixelwise` with a password.
- A database `pixelwise` owned by that role.

PostgreSQL keeps roles and databases as independent objects, so the matching names are a readability convention, not a requirement; the application could just as well live in a database called `predictions` owned by a role called `api`.

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/

Python is ergonomics. Two reasons, why choosing to use an ORM: 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, so user input never ends up concatenated into a SQL statement. SQL injection becomes a footgun the framework refuses to hand you.

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 call once per request; `Base.metadata.create_all(engine)` creates the table on first run.

The connection string is a single URL: `postgresql://user:pass@host/dbname`. It encodes everything the driver needs to reach the database, and it lives in `.env` for access restrictions and security. A leaked `DATABASE_URL` is a leaked database.

Exercise: The `Prediction` Model

Install the driver and the ORM on `dev` into the virtual environment, then pin the new packages to `requirements.txt` so `prod` picks them up on the next deploy.

Install.

```
cd ~/pixelwise
source .venv/bin/activate
pip install sqlalchemy
psycopg2-binary
pip freeze > requirements.txt
```

Write `app/models.py` on `dev` with the schema for one prediction, the class label, the confidence, and the model version:

```
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)
```

Three columns plus bookkeeping. `prediction`, `confidence`, and `model_version` carry the answer; `id` and `created_at` are bookkeeping the database fills in for you. Every later question, distribution per digit, average confidence per model version, error rate over time, lives in this table.

Document `DATABASE_URL` in `~/pixelwise/.env.example` on `dev`, and add the same line to your local `.env` on both VMs so the engine can find the database. The URL interpolates `DB_PASSWORD` from the same file, so the secret lives in one place:

```
DATABASE_URL=postgresql://pixelwise:${DB_PASSWORD}@localhost/pixelwise
```

Create the table once from a small initialisation script saved as `init_db.py` at the repository root on `dev`:

```
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)
```

Initialise the schema. by running `python init_db.py` on `dev`.

Commit and deploy. Stage the new and changed files on `dev`, push, and on each VM install the pinned dependencies and run `python init_db.py` to create the schema. Then confirm with `psql` that the `predictions` table is listed:

```
psql -U pixelwise -d pixelwise -h localhost -c "\dt"
```

Commit and deploy.

On `dev`:

```
git add app/models.py init_db.py
  requirements.txt .env.example
git commit -m "Add Prediction
model"
git push
source .venv/bin/activate
python init_db.py
```

On `prod`:

```
cd /opt/pixelwise && git pull
source .venv/bin/activate
pip install -r requirements.txt
python init_db.py
```

Optional: One-shot deploy. Make `bash setup-server.sh` also install pinned dependencies and create the schema by appending two stanzas after the Postgres provisioning block:

```
if [ -d "$SCRIPT_DIR/.venv" ] &&
   [ -f
"$SCRIPT_DIR/requirements.txt"
]; then
   source
"$SCRIPT_DIR/.venv/bin/activate"
   pip install -r

"$SCRIPT_DIR/requirements.txt"
fi

if [ -f "$SCRIPT_DIR/init_db.py"
]; then
   (cd "$SCRIPT_DIR" && python
init_db.py)
fi
```

Keep the order: The init stanza relies on the venv the first stanza activates.

Exercise: Wiring the API to the Database

The database is ready. Let's make the API populate it with real data.

Edit `app/main.py` on `dev` so `POST /classify` writes a row before returning the response, and `GET /results` returns real rows instead of the current scaffold. The change lands in three pieces: Two new imports at the top of the file, an addition to the classify handler, and a real body for results.

```
from app.models import Prediction, SessionLocal
```

Two new imports. `Prediction` is the row class you defined in `app/models.py`; `SessionLocal` is the session factory built from the engine. Add the line at the top of `main.py` next to the existing imports.

```
@app.post("/classify",
          response_model=ClassifyResponse,
          dependencies=[Depends(verify_api_key)])
@limiter.limit("30/minute")
def classify(request: Request,
             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
```

Classify now writes a row. The decorators and signature from Block 5 stay untouched. The body picks up one new intermediate, `result`, so the same dict can be persisted and returned, then a five-line session block that opens a `SessionLocal`, adds the new `Prediction`, commits, and closes.

A session is the unit of database work: Open it at the start of the handler, close it at the end. FastAPI documents a more elegant variant through `Depends`; the manual version here is the simplest one that shows what is happening. The returned `result` dict is unchanged, so the response body stays identical to the contract we defined in Block 5.

```
@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,
```

Results runs one query. It pulls the twenty newest rows from `predictions`, ordered by `created_at` descending, and shapes each row into the JSON object the frontend will expect in the next block. The Block 5 stub that returned an empty list is gone.

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

Restart the service. On `dev` the running `uvicorn --reload` picks up the new code on save. On `prod` restart through systemd so the unit re-reads `app/main.py`:

```
sudo systemctl restart pixelwise   # prod only
```

Smoke test the wiring with one unauthenticated `GET /results` call. An empty table is exactly what you want here: A successful response with an empty list proves the engine opened, the session works, the query ran, and the JSON shape is right. Any of those broken and the call returns a 500, not an empty list:

```
# on dev, against the local uvicorn
curl -s http://localhost:8000/results \
  | python3 -m json.tool

# on dev, against prod via the API gateway
curl -s http://192.168.56.11:8000/results \
  | python3 -m json.tool
```

Commit and tag `v0.5`.

```
git add app/main.py app/models.py
  init_db.py requirements.txt
  .env.example
git commit -m "Add PostgreSQL
persistence"
git tag v0.5
git push --follow-tags
```

Stage on `dev`, commit with a message that names the change, tag the result, and push both the commit and the tag in one shot.

Optional: Feed it payload. So far our flow has never been hit by real digit drawings, and on that note testing really came short. Build a 28-by-28 zero payload once, perhaps wrapped in a `probe_init.py`, then use it to test the API and the database.

Optional Exercise: First Migration with Alembic

SCHEMAS ALWAYS CHANGE. You design the table, deploy it, store 500 predictions. Then you realise you forgot a column. 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. This exercise is optional: The mainline of the course does not depend on it, but running it once gives you the feel of the migration cycle on a real database. The example below adds a `client_ip` column for debugging, but any new column works.

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

INITIALISE ALEMBIC ON DEV. Alembic is the migration tool for SQLAlchemy; the cycle is `init`, `revision --autogenerate`, `upgrade head`, run in that order. All file edits in this exercise happen on `dev`. `alembic init` writes its scaffold into the current directory, so run everything from the repository root:

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.

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

The `pip freeze` step captures the new package in `requirements.txt` so `prod` picks it up on the next deploy.

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. In `~/pixelwise/app/models.py` on `dev`, add `client_ip = Column(String)` between the `confidence` and `model_version` lines.

GENERATE THE MIGRATION from the repository root on `dev`:

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

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

WIRE `client_ip` INTO THE HANDLER in `~/pixelwise/app/main.py` on `dev`. `Request` is already imported and `request: Request` is already a parameter on `classify` from the Block 5 rate limiter, so the only change is one keyword argument in the `Prediction(...)` call:

```
db.add(Prediction(
    prediction=result["prediction"],
    confidence=result["confidence"],
    client_ip=request.client.host,
    model_version="v1"))
```

COMMIT AND DEPLOY. On `prod`, order matters: Install the new pinned dependency and apply the migration first, then run `setup-server.sh`. `setup-server.sh` restarts the service with code that references the new column, so if the migration has not run yet, the handler will try to INSERT a column that does not exist in the database. From `dev`, stage the changes and push:

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

Then on `prod`, pull, install the new dependency, run the migration, and only then restart the service:

```
cd /opt/pixelwise
git pull
source .venv/bin/activate
pip install -r requirements.txt
alembic upgrade head
bash setup-server.sh
```

VERIFY ON `prod` WITH `psql`:

```
psql -U pixelwise -d pixelwise -h localhost \
  -c "SELECT prediction, client_ip
      FROM predictions
      ORDER BY created_at DESC LIMIT 5;"
```

Old rows show `NULL` for `client_ip`; new predictions store the caller's address now that the field is wired in. That gap is the visible signature of a migration: The column exists everywhere, but its values populate only from the moment the new code shipped.

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 is the difference between dropping and recreating a table versus running a migration?
- If you ran the optional Alembic exercise, why do old rows show `NULL` for the new column while later rows carry a value?
- What happens if you store the database password directly in your Python code instead of in `.env`?
- Why is a dedicated database user with minimal privileges better than connecting as the superuser?
- How does the ORM make SQL injection harder, and where does the protection break down?

RECAP of key concepts:

- Relational databases store data in typed tables; SQL is the language that queries and changes them.
- PostgreSQL is the production database for this course; SQLite is too limited for a multi-user web service.
- SQLAlchemy maps tables to Python classes and uses parameterised queries by default.
- Migrations let schemas change without losing rows; Alembic generates and applies them.
- Database credentials live in `.env`, never in code; the application user has minimal privileges.

TEASER. We classify, we store, we query, but still no user can touch our backend.

BRIDGE. The service classifies, stores, and queries, but the only way in is `curl` with an API key on a non-standard port. No browser, no URL, no human-friendly surface. This needs to change next we put `nginx` in front of the FastAPI process, serves a small drawing frontend over plain HTTP and HTTPS, and route browser traffic into the same `POST /classify` and `GET /results` handlers you just wired up. The database stays where it is; the prediction it stores will start arriving from a canvas in a browser rather than a shell command.

Index

.env, 39, 50
.env.example, 50
.gitignore, 29, 69
.pkl, 57, 60
/etc, 17
/opt, 17
__pycache__, 69
12-factor app, 38, 50

abstraction, 54
Alembic, 91, 104
  autogenerate, 104
alembic.ini, 104
Ansible, 44
API, 73, 75
  design, 83
APIKeyHeader, 86
apt, 39
assertion, 67, 68
assessment, 9
authentication, 84

Bash, 16, 44
batch inference, 67

chmod, 16
chown, 16
classifier, 55
conda, 46
containers, 54
continuous integration, 36
course work, 9
curl, 80, 89, 103
CVE, 48, 49

data hygiene, 26
database, 91
DATABASE_URL, 99
dependencies, 38
deployment, 73
dev VM, 11
distribution, 16
Docker, 54
Dockerfile, 54
DVC, 26

Ed25519, 18
environment variables, 50
exercises, 14, 17, 20, 25, 27, 31, 33, 34, 37, 41, 45, 47, 49, 51, 52, 59, 61, 65, 68, 77, 79, 82, 85, 88, 95, 97, 100, 102, 104

FastAPI, 73, 80
fork, 35

Git, 21
  -u flag, 35
  blob, 26
  branch, 26
  branching, 32
  clone, 35
  commit, 26
  data model, 26
  default branch, 33
  diff, 47
  fetch, 35
  fork, 35, 43, 71, 77
  HEAD, 26
  history, 30
  HTTPS clone, 37, 43
  install, 27
  LFS, 26
  merge conflict, 32
  merging, 32
  pull, 35, 52
  push, 34, 35
  remotes, 35
  repository, 29
  reset, 43
  staging, 29
  staging area, 29
  status, 33
  tag, 41, 59, 69, 77, 95
  tags, 34
  tree, 26
  working directory, 29
GitHub, 21, 35
  username, 37
gunicorn, 87

HTTP, 73, 75, 78
  429, 86
  headers, 84
  methods, 78, 79
  status codes, 78
httpbin, 79
Hugging Face, 26, 60
hypervisor, 13

idempotence, 44, 61, 88, 97
inference, 55
inference interface, 67
infrastructure as code, 88
interface contracts, 24
isolation, 46
issues, 35

joblib, 48, 55, 60, 63
journalctl, 87
JSON, 79

KeePass, 18

least privilege, 16, 96
license, 2
Linux, 10, 16

lock file, 53
LogisticRegression, 61

migration, 104
MNIST, 11, 55, 65
model artefact, 57
model card, 63
model contract, 63
model integration, 55
model versioning, 60, 61, 71
MODEL_PATH, 67
MODELCARD.md, 63

nano, 19, 33
NumPy, 63

OpenAPI, 78
OpenSSH, 18
Oracle VirtualBox Manager, 10
ORM, 91, 99
os.getenv, 50

pandas, 65, 68
path operation, 80
persistence, 93
pickle, 60
pip, 38, 48
pip audit, 48, 49
PixelWise, 11, 39, 57, 75, 93
poetry, 53
PostgreSQL, 91, 96
primary key, 96
project report, 9
project structure, 38
Projektbericht, 9
provisioning, 44, 45, 61, 62
psql, 96, 98
pull request, 35
PuTTY, 18
Pydantic, 73, 80

PyPI, 48
pyproject.toml, 53
python-dotenv, 48, 50

rate limiting, 84
recap, 21, 55, 72, 90, 106
recovery, 25, 41, 59, 77, 95
ReDoc, 78
reflection, 21, 55, 72, 90, 106
relational database, 93, 96
Remote-SSH, 18
repository layout, 45
reproducibility, 39
requirements.txt, 39, 48, 49, 68
REST, 73, 78

schema, 100
  initialisation, 100
scikit-learn, 48, 63, 65
secrets, 30
security, 72, 90
semantic versioning, 34, 48
server, 11
service, 76
set -a, 97
set -euo pipefail, 44
setup-server.sh, 39, 44, 61, 88, 97
shebang, 44
shell prompt, 47
slowapi, 84
snapshot, 13, 25
SQL, 91, 96
  GROUP BY, 96
  JOIN, 96
SQL injection, 99
SQLAlchemy, 91, 99
  session, 102
SQLite, 96
SSH, 10, 18
  config, 27, 42
  key, 27, 42, 43
  keygen, 27
  known_hosts, 42
  login, 37
  private key, 28
  public key, 28
ssh-agent, 18, 42
stateless, 78
sudo, 27
Swagger UI, 78, 83, 86
systemctl, 19, 87, 89
systemd, 19, 73, 75, 87

tig, 34
touch typing, 17
train.py, 60
train/serve skew, 57, 63
transitive dependencies, 53
type-1 hypervisor, 13
type-2 hypervisor, 13

Ubuntu, 16
  terminal, 27
Unix philosophy, 16
uv, 53
uvicorn, 73, 75, 83, 87

validation, 58
venv, 46
verification, 58, 65
version control, 21, 23
Virtual Machines, 10, 13
virtualenv, 38, 46
VS Code, 18

Weights & Biases, 26
whoami, 16

X-API-Key, 85