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.

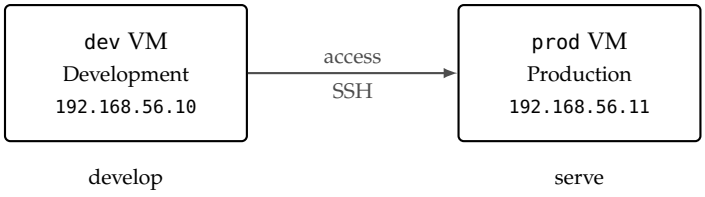

Figure 1: The two-machine setup. The dev VM is where code is written and tested; the prod VM is where the deployed application runs. Code travels from dev to prod through a deliberate deployment step over SSH.

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.

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.

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.

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.

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.

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.

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.

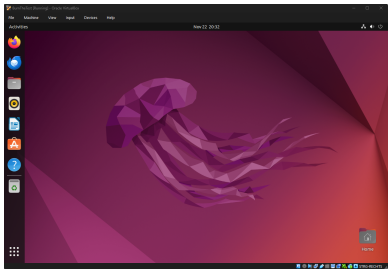

Figure 3: A running Ubuntu guest inside the Oracle VirtualBox Manager. The guest believes it owns a physical machine; the hypervisor is what makes that illusion hold.

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?