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.

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.

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`

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.

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
```

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.
`(\ /)`
`(0.o)`
`(> <)` Bunny approves these changes.

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.