LEARN BY DOING

# The Agentic Crew

*Hands-On Guide — Linux Edition*

Rasmus Bornhøft Schlünsen

March 2026

rev 95

# A Note Before We Start

This book is the practical companion to *The Agentic Crew*. Where the main book explains the ideas, this one puts them into practice.

Each chapter is an exercise. You'll set up real tools, work with real repositories, and build real things — starting from scratch. The exercises are written for Linux (Ubuntu-based examples — adapt for your distribution).

You don't need to be a programmer. You do need to be willing to type commands into a terminal and see what happens. That's how you learn this stuff — not by reading about it, but by doing it.

By the end of this book, you'll have a working development environment, hands-on experience with Git and GitHub, and the confidence to collaborate on real projects using AI agents as your co-pilot.

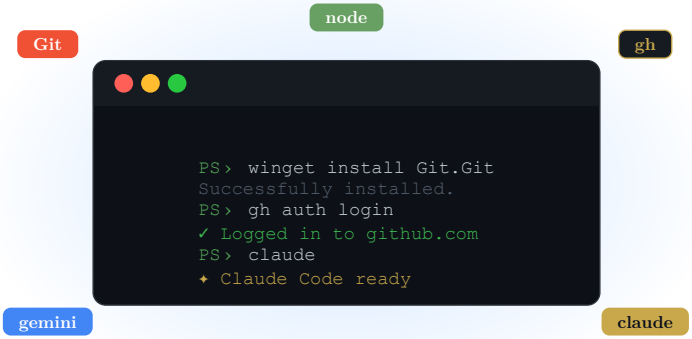
*Rasmus Bornhoft Schlunsen*
*March 2026*

# Exercises

# Setting Up Your Workstation

Before you can contribute to a project, fix a bug, or even read source code properly, you need a working environment. This chapter gets your machine ready — from scratch, on whatever OS you're running.

By the end of this chapter, you'll have a terminal, a package manager, Git, the GitHub CLI, and optionally an AI assistant — all ready to go.




*Your terminal: the workshop where everything begins*

## Open Your Terminal

The terminal is your command line — the place where you'll run everything in this book.

How you open the terminal depends on your desktop environment, but these shortcuts work on most distributions:

- **Ubuntu / GNOME:** `Ctrl + Alt + T`
- **KDE Plasma:** right-click the desktop → **Open Terminal**
- **Any distro:** search for "Terminal" in your app launcher

Verify the shell:

```
echo $SHELL
```

You should see `/bin/bash` or `/bin/zsh`.

> **Why the terminal?** AI coding agents live in the terminal. You can't pair-program with an agent if you don't have a place for it to work. Think of this as setting up your workshop before you start building.

## Install a Package Manager

A package manager lets you install software by typing one command instead of downloading installers and clicking through wizards. Each platform has its own.

**apt / dnf**
Linux distributions ship with a package manager. The most common:

> **What is `sudo`?** On Linux, `sudo` runs a command as an administrator — like "Run as Administrator" on Windows.

You'll be asked for your user password. Nothing is shown as you type; that's normal.

**Ubuntu, Debian, and derivatives:**

```
sudo apt update
```

**Fedora, RHEL, and derivatives:**

```
sudo dnf check-update
```

**Arch Linux:**

```
sudo pacman -Sy
```

No install needed — these come with your OS. The examples in this guide use `apt`; swap for your distro's equivalent.

## Install Git

Git is version control. It tracks every change to every file in a project, and it's how teams collaborate without overwriting each other's work. Every project in this book uses Git.

**Ubuntu / Debian:**

```
sudo apt install git
```

**Fedora:**

```
sudo dnf install git
```

**Arch:**

```
sudo pacman -S git
```

—

Verify:

```
git --version
```

You should see a version number like `git version 2.47.1`.

Now set your identity so Git knows who's making changes:

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

Use the same email as your GitHub account.

## See Git in Action

Git is installed — but you haven't used it yet. Let's do a quick 60-second test so Git isn't a mystery when Chapter 2 arrives.

Create a practice folder and step into it:

```
mkdir test-repo && cd test-repo
```

Tell Git to start tracking this folder:

```
git init
```

Check the status:

```
git status
```

You should see something like: `On branch main — nothing to commit`. That's Git telling you it's watching this folder and there's nothing new to record yet.

In Chapter 2 you'll use `git status` constantly — it's how you check what's changed. Now you've seen what it looks like when everything is clean.

Head back to your home folder when you're done:

```
cd ..
```

> **What just happened?** `git init` created a hidden `.git` folder inside `test-repo`. That folder is where Git stores its

entire history. Every project that uses Git has one. You never need to touch it directly.

## Install the GitHub CLI

The GitHub CLI (`gh`) lets you fork repositories, create pull requests, and manage issues — all without leaving the terminal.

**Ubuntu / Debian** — `gh` isn't in the default package repositories, so you need to add GitHub's official one first. These commands do that, then install `gh`:

```
sudo mkdir -p -m 755 /etc/apt/keyrings
wget -qO- https://cli.github.com/packages/githubcli-
archive-keyring.gpg \
  | sudo tee /etc/apt/keyrings/githubcli-archive-
keyring.gpg > /dev/null
sudo chmod go+r /etc/apt/keyrings/githubcli-archive-
keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/
etc/apt/keyrings/githubcli-archive-keyring.gpg] https://
cli.github.com/packages stable main" \
  | sudo tee /etc/apt/sources.list.d/github-cli.list > /
dev/null
sudo apt update && sudo apt install gh
```

You can paste the entire block at once — the terminal will run each line in sequence. You'll be asked for your password on the first `sudo`.

**Fedora:**

```
sudo dnf install gh
```

**All Linux distributions** — or install directly from github.com/cli/cli/releases by downloading the binary for your architecture.

—

Verify:

```
gh --version
```

## Create a GitHub Account

If you don't have one, sign up at github.com/signup. It's free. You'll need it for everything from Chapter 2 onward.

## Authenticate

Now connect your terminal to your GitHub account:

```
gh auth login
```

When prompted, choose:

- **GitHub.com**
- **HTTPS**
- **Login with a web browser**

It will give you a one-time code and open your browser. Paste the code, authorize, and you're connected.

Verify it worked:

```
gh auth status
```

You should see `Logged in to github.com`.

## (Optional) Install an AI Coding Assistant

This book is about working with AI agents. While you don't strictly need one for the exercises, having an AI assistant in your terminal makes the experience real.

Both Claude Code and Gemini CLI require Node.js. Install it first.

**Install Node.js**
**Ubuntu / Debian:**

```
sudo apt install nodejs npm
```

**Fedora:**

```
sudo dnf install nodejs npm
```

Close and reopen your terminal, then verify:

```
node --version
npm --version
```

**Option A: Claude Code**
Claude Code is Anthropic's AI coding agent. It runs in your terminal and can read, write, and reason about code.

Install it using the official installer (the recommended method):

```
curl -fsSL https://claude.ai/install.sh | bash
```

Alternatively, if you prefer npm:

```
npm install -g @anthropic-ai/claude-code
```

Launch it:

```
claude
```

You'll be prompted to authenticate with your Anthropic account on first run.

**Option B: Gemini CLI**
Gemini CLI is Google's AI coding agent. Similar concept, different model.

```
npm install -g @google/gemini-cli
gemini
```

You'll need a Google API key. Set it in your current session:

```
export GEMINI_API_KEY="your-key-here"
```

To make it permanent so it's set every time you open a terminal, add the `export` line to your shell config file. This file runs automatically when your terminal starts:

Open `~/.bashrc` with:

```
nano ~/.bashrc
```

Add `export GEMINI_API_KEY="your-key-here"` on a new line at the bottom, save, and close. Then run `source ~/.bashrc` to apply the change in your current session.

If you're not sure where to get a Google API key, go to aistudio.google.com, sign in, and create a key under **Get API key**. It's free to start.

**Which one?**
Either works for this book. Claude Code tends to excel at code reasoning and multi-file edits. Gemini CLI has strong integration with Google's ecosystem. Try both if you want — they're free to start with. The exercises will show prompts that work with either tool.

## Verify Everything

Run this quick checklist:

```
git --version
gh --version
gh auth status
```

If all three commands work, you're ready. Your workstation is set up, your identity is configured, and you have a direct line to GitHub.

## The Prompt-First Way

Once Claude Code or Gemini CLI is installed, you don't have to remember every command — you can just describe what you want in plain English. Here's how you'd ask an AI agent to verify your entire setup for you.

Open your AI assistant in the terminal:

```
claude
```

Then try prompts like these:

- *"Check whether Git is installed and properly configured with a name and email."*
- *"Is the GitHub CLI installed and authenticated? Show me the status."*
- *"Run a quick checklist: git, gh, and gh auth status — tell me what's working and what isn't."*

The agent will run the commands, interpret the output, and tell you in plain language what's ready and what still needs attention. If something is missing or broken, ask it:

- *"Git isn't configured with my email — how do I fix that?"*
- *"Walk me through authenticating the GitHub CLI step by step."*
- *"I'm on macOS and Homebrew isn't in my PATH — how do I fix that?"*

> **Commands vs. prompts.** Both approaches get you to the same place. Commands are fast and precise once you know them. Prompts are forgiving — they meet you where you are. As you build experience, you'll find yourself switching between the two naturally.

In the next chapter, we'll put it all to use: you'll fork a real repository, read a real chapter of a real book, write an honest review, and submit your first pull request.

# Your First Pull Request

This chapter is a hands-on exercise. By the end of it, you'll have forked a real repository, created a branch, written a review of a book chapter, and submitted a pull request — the standard way open-source contributors propose changes.

This isn't a simulation. Your pull request will show up on GitHub for real people to read.


*fork → branch → pull request: the open-source contribution loop*

## What You'll Need

Everything from Chapter 1:

- A terminal (open and ready)
- Git installed and configured

- GitHub CLI authenticated
- A GitHub account

## What's About to Happen

Before the first command, here's the complete journey you're about to take — seven steps, one after another:

1. **Fork** — Create your own copy of the book's repository on GitHub
2. **Clone** — Download that copy to your machine
3. **Branch** — Create a safe, isolated version where you can make changes
4. **Write** — Add your review as a new file
5. **Commit** — Save your changes with a message (Git's version of hitting Save)
6. **Push** — Send your changes back up to your GitHub copy
7. **Pull Request** — Ask the author to merge your changes into the original

This is the open-source workflow. Every contribution to every major project in the world follows these same seven steps. You'll have done all of them by the end of this chapter.

Here are the key words you'll see — read them once now, and they'll make sense as you go:

| **Term** | **What it means** |
|---|---|
| **Repository** | A project folder that Git is tracking. Often called a "repo." |
| **Fork** | Your personal copy of someone else's repository, stored on GitHub. |

| | |
|---|---|
| **Clone** | Downloading a repository from GitHub to your machine. |
| **Branch** | A parallel version of the project where you work without touching the main copy. |
| **Commit** | A saved snapshot of your changes, with a message describing what you did. |
| **Push** | Sending your commits from your machine up to GitHub. |
| **Pull Request** | A formal request to merge your branch into the original project. |

You don't need to memorise these. They'll become natural as you use them.

## Fork and Clone the Book

We'll work with *The Agentic Crew* — the book you're reading right now. The source lives on GitHub as a public repository.

Run this single command:

```
gh repo fork schlunsen/theagenticcrew --clone --remote
```

This does three things in one step:
1. Creates your own copy (a "fork") on GitHub
2. Downloads it to your machine (a "clone")
3. Sets up the connection between your copy and the original

Enter the project directory:

```
cd theagenticcrew
```

Verify your setup:

```
git remote -v
```

You should see two remotes:

- `origin` — your fork (where you push changes)
- `upstream` — the original repository (where you pull updates)

## Explore the Project

Before you change anything, look around. The book is written in Typst, a modern typesetting language. The source files are plain text — you can read them in any editor.

See the project structure:

```
ls
ls chapters/
```

The chapters are in the `chapters/` directory, numbered and named. The one we care about is `01-introduction.typ`.

## Read Chapter 1

Open it in a text editor:

```
xdg-open chapters/01-introduction.typ
```

Or with VS Code on any platform, if you have it installed:

```
code chapters/01-introduction.typ
```

You'll see Typst markup — headings start with `=`, emphasis uses `_underscores_`, and the rest is prose. It reads like a regular document.

**Take your time.** Read the full chapter. The introduction covers:

- The author's journey discovering agentic workflows
- Why the ground is shifting for software engineers
- Who the book is for and how to read it
- What the book is — and what it isn't

Form your own opinions as you read. That's the whole point of this exercise.

## Create a Branch

In Git, you don't edit the main copy directly. You create a **branch** — a parallel version where you can make changes without affecting anyone else's work.

```
git checkout -b review/chapter-1-YOUR-GITHUB-USERNAME
```

Replace `YOUR-GITHUB-USERNAME` with your actual username. For example:

```
git checkout -b review/chapter-1-johndoe
```

> **Newer Git versions (2.23+)** offer `git switch -c` as a more explicit alternative to `git checkout -b`. Both do the same thing. You'll see `checkout` used most widely in tutorials and documentation, so that's what we use here.

> **Why branches?** Imagine ten people all editing the same Google Doc at once — chaos. Branches let everyone work independently, then merge their changes one at a time. It's how every serious software project operates.

## Write Your Review

Create a `reviews` folder and your review file.

```
mkdir -p reviews
nano reviews/review-chapter-1-YOUR-GITHUB-USERNAME.md
```

This opens a simple terminal text editor. Type (or paste) your review directly. When you're done, press `Ctrl+X` to exit, then `Y` to confirm saving, then `Enter` to keep the filename.

You can also use any editor you like — `code`, `vim`, or anything else that feels natural.

Here's a template to start with. Type it out or copy it from this page, then fill in your actual thoughts below each heading:

```
# Review — Chapter 1: Introduction

**Reviewer:** @YOUR-GITHUB-USERNAME
**Date:** YYYY-MM-DD

## First Impressions
What stood out to you when you first read this chapter?

## What Worked Well
What parts resonated? What was clear and engaging?

## What Could Be Improved
Be honest but constructive. Confusing sections?
Missing context? Anything that felt off?

## Who Would Benefit From This Chapter
Based on what you read, who is the ideal reader?

## Rating
Your overall rating out of 5, with a one-line summary.
```

Don't overthink it. A genuine paragraph for each section is worth more than a polished essay.

## The Prompt-First Way

The command-by-command approach above is the full manual workflow. Once Claude Code or Gemini CLI is installed, you can describe the entire task in plain English and let the agent handle most of it.


*Two paths, one destination — commands or prompts, the result is the same*

Open your AI assistant from inside the cloned project directory:

```
claude
```

Or:

```
gemini
```

### Let the agent read the chapter

Instead of opening the file yourself, ask the agent to read it and give you a summary:

- *"Read chapters/01-introduction.typ and summarise what it covers."*

- *"What's the main argument of chapters/01-introduction.typ? Who does the author think this book is for?"*

This is useful when you want a fast orientation before reading the whole thing yourself.

**Draft your review together**

Once you've read the chapter (yourself or with the agent's help), use the agent as a thinking partner:

- *"Read chapters/01-introduction.typ and give me your honest take — what works well and what's missing for a book introduction?"*
- *"I think the introduction spends too long on the author's backstory. Do you agree? What would you cut?"*
- *"Help me fill out this review template for chapter 1."* (paste the template into the prompt)

The agent will give you a draft to react to. Agree, disagree, edit — the review should end up in your voice, not the agent's.

**Let the agent do the Git work**

Here's where the prompt-first approach really shines. After you've written your review, you can hand the Git steps to the agent:

- *"Create a branch called review/chapter-1-YOUR-GITHUB-USERNAME, add my review file to it, commit it with a sensible message, push it to my fork, and open a pull request."*

The agent will run each command, show you what it's doing, and flag any problems. You stay in the loop without having to remember the exact syntax.

Or go even further — describe the whole task upfront before you start:

- *"I want to submit a review of chapters/01-introduction.typ as a pull request to this repo. Walk me through it step by step, or just do it for me if I say go."*

The agent will outline the plan, wait for your approval, and execute.

**When to use commands vs. prompts**

| **Use commands when...** | **Use prompts when...** |
|---|---|
| You know exactly what to do | You're not sure what step comes next |
| Speed matters | You want to understand what's happening |
| You're scripting or automating | You're exploring or experimenting |
| The command is short and memorable | The task involves multiple steps |

> **A note on honesty.** Whether you write your review manually or draft it with an AI assistant, the opinions should be yours. Use the agent to sharpen your thinking and handle the mechanical steps — not to replace your judgment. The best reviews have a human voice. An agent can help you find the words for what you already feel, but it can't feel it for you.

## Stage and Commit

Once your review is written and saved, tell Git to track it:

```
git add reviews/
```

Check what's staged:

```
git status
```

You should see your review file listed under "Changes to be committed."

Now commit — this creates a snapshot with your changes and a message explaining what you did:

```
git commit -m "Add Chapter 1 review by YOUR-GITHUB-
USERNAME"
```

## Push to Your Fork

Send your branch to GitHub:

```
git push origin review/chapter-1-YOUR-GITHUB-USERNAME
```

Your changes now exist both on your machine and on GitHub.

## Create the Pull Request

This is the moment. A pull request says: "Hey, I made some changes on my fork — would you like to merge them into the original project?"

```
gh pr create --title "Chapter 1 Review: YOUR-GITHUB-
USERNAME" --body "Honest review of Chapter 1
(Introduction) of The Agentic Crew. Covers first
impressions, strengths, areas for improvement, and target
audience fit."
```

The CLI will output a URL. Open it in your browser — that's your pull request, live on GitHub.

You can also view it anytime:

```
gh pr view --web
```

## What Happens Next

Once you submit your PR:

1. **The author reads it** — honest feedback on a book-in-progress is genuinely valuable
2. **They may comment** — asking follow-up questions or thanking you for a specific insight
3. **It gets merged** — your review becomes a permanent part of the project's history

That's the open-source workflow: fork, branch, change, push, PR. Every contribution to every major project in the world follows this pattern. You just did it for real.

## Troubleshooting

**`git push` asks for a password:** Run `gh auth setup-git` to configure Git to use your GitHub CLI credentials. Works the same on all platforms.

**`nano` isn't installed on your distro:** Use `vi reviews/...` (available everywhere) or install nano: `sudo apt install nano`.

**You made a typo in your branch name:** Create a new branch from main: `git checkout main && git checkout -b review/chapter-1-corrected-name`, then copy your review file over.

**The PR targets the wrong branch:** You can specify the base: `gh pr create --base main --title "..."`.

## Quick Reference

| Task | Command |
|---|---|

| | |
|---|---|
| See your branches | `git branch` |
| See what changed | `git status` |
| View the diff | `git diff` |
| Pull latest from upstream | `git fetch upstream && git merge upstream/main` |
| View your PR | `gh pr view --web` |

# Your First AI-Generated Project

So far you've set up your tools and submitted your first pull request. Now you're going to do something genuinely impressive: take a real open-source project, describe what you want in plain English, and let an AI agent transform it — generating custom illustrations, narration audio, and animated backgrounds using AI models from Hugging Face.

No prior coding experience required. The agent writes the code. You describe the vision.


*One prompt. Three AI-generated assets. A complete presentation.*

## What You'll Build

**Web Presenter** is an open-source presentation framework: pure HTML, CSS, and JavaScript, no build step required. It supports slide-by-slide narration (MP3 files), animated Three.js backgrounds, and smooth CSS transitions. You can run it in any browser with a single command.

By the end of this chapter, you'll have:

1. Forked and cloned the project
2. Chosen a topic for your own presentation
3. Had an AI agent generate custom illustrations for each slide
4. Had an AI agent generate spoken narration for each slide
5. Customised the animated background to match your theme
6. Previewed the finished result in your browser

Every piece of AI-generated content — the images, the audio, the animation colours — will come from Hugging Face's free inference API. Hugging Face hosts hundreds of open AI models that anyone can call without a credit card. You'll talk to it through your AI coding agent.

## What You'll Need

From the previous chapters:

- Git installed and configured
- GitHub CLI authenticated
- Claude Code or Gemini CLI ready in your terminal

New for this chapter:

- A Hugging Face account (free)
- A Hugging Face API token
- Python 3 (to run a local server and call the Hugging Face API)

- The `huggingface_hub` Python library (one `pip install` command — covered below)

**Get a Hugging Face Account and Token**
If you don't have one, sign up at huggingface.co/join. It's free — no credit card required.

Once you're logged in:

1. Click your profile picture (top right) → **Settings**
2. Click **Access Tokens** in the left sidebar
3. Click **New token**
4. Give it a name like `agentic-crew` and select **Read** access
5. Click **Generate a token** and copy it

Keep this token handy — you'll paste it into your terminal in the next step.

**Check Python**
Run:

```
python3 --version
```

You should see Python 3.8 or higher. If Python isn't installed:

```
sudo apt install python3 python3-pip
```

Close and reopen your terminal after installing, then run `python3 --version` again to confirm.

**Install the Hugging Face Library**
The scripts the agent writes will use Hugging Face's official Python library. Install it now:

```
pip install huggingface_hub
```

On some systems you may need `pip3` instead of `pip`. You only need to do this once.

## Set Your Hugging Face Token

Your AI agent will use this token to call Hugging Face's API. Set it as an environment variable so the agent can access it without hard-coding it into any file.

```
export HF_TOKEN="your-token-here"
```

Replace `your-token-here` with the token you copied. Now verify it was set:

```
echo $HF_TOKEN
```

You should see your token printed back. If nothing appears, run the `export` command again — the variable didn't stick.

> **Keep tokens out of files.** Never paste your API token directly into code or commit it to Git. Environment variables keep it in memory only — no risk of accidentally pushing it to a public repository.

This setting lasts for your current terminal session. To make it permanent, add the `export` line to your shell config file (`~/.bashrc`).

## Fork and Clone the Project

This single command creates your own copy of the project on GitHub and downloads it to your machine:

```
gh repo fork schlunsen/web-presenter --clone --remote
```

Step into the project directory:

```
cd web-presenter
```

Verify your connections to GitHub:

```
git remote -v
```

You should see two entries:

- `origin` — your fork (where you push your changes)
- `upstream` — the original project (where you can pull updates)

Each remote is listed twice — once for fetching, once for pushing. Four lines total is correct. If you only see `origin`, something went wrong — open your AI assistant and describe what you see; it will help.

## Explore the Project

Have the agent explain what you're working with. Open your AI assistant from inside the project directory.

If you installed Claude Code, run:

```
claude
```

If you installed Gemini CLI, run:

```
gemini
```

Either works for everything in this chapter. Once it's open, ask:

- *"Read index.html, presentation-script.js, and presentation-styles.css. Explain how this presentation framework works — how slides are structured, how audio narration is loaded, and how the animated background works."*

The agent will read the files and give you a plain-language summary. You'll understand the project in two minutes rather than thirty. It's only reading at this point — nothing is being changed yet.

Once you have a feel for it, ask about the existing assets:

- *"List all the files in presentation-audio/ and presentation-images/. What's already there?"*

You'll see the existing narration clips and images — placeholders you're about to replace with your own.

## Choose Your Topic

Pick something you know or care about. The presentation has ten slides, so you want a topic with enough substance for ten short points. Some ideas:

- A technology you use at work
- A hobby or skill you want to explain to a friend
- A project you're building
- An argument you want to make about something

For the rest of this chapter, we'll use the placeholder **[YOUR TOPIC]** — replace it with whatever you choose.

Write down ten short bullet points — your slide topics. One sentence each. You'll paste these into the agent prompts that follow.

> **Make your bullet points visual.** Each one will become an AI-generated image, so specific and concrete works better than abstract. "A warehouse filled with rows of server racks" will generate a better illustration than "technology infrastructure." "A child reading under a tree at sunset" is better than "education."

## Generate the Illustrations

In this step you'll ask the agent to write a Python script. You won't write it yourself — the agent will. Your job is to provide your ten bullet points and then ask the agent to run the script. Everything else is automatic.

Open your AI assistant (if it's not already open) and give it this prompt. **Do not copy it word-for-word** — replace `[YOUR TOPIC]` with your actual topic and `[paste your bullet points]` with your ten sentences:

- *"Write a Python script using the huggingface_hub library. Use InferenceClient with the token from the HF_TOKEN environment variable. Call client.text_to_image() with the model black-forest-labs/FLUX.1-schnell to generate one image per slide, and save each result to presentation-images/ as slide-01.jpg, slide-02.jpg, and so on. Here are the ten slide topics: [paste your bullet points]. Make the prompts vivid and consistent in style."*

The agent will write the script. Review it — you don't need to understand every line, but check that it reads the token from the environment (`os.environ` or similar) rather than having a value hard-coded in the code.

When you're happy with it, ask the agent to run it:

- *"Run the script."*

Generation takes roughly 10–20 seconds per image. The agent will report progress as each one completes. When it's done, the `presentation-images/` folder will contain ten new files.

**What if an image looks wrong?** Ask the agent to regenerate just that one: *"The illustration for slide 3 doesn't look right — it should show [description]. Regenerate just that one with a better prompt."* Image generation is iterative. A second or third attempt usually lands closer to what you had in mind.

## Generate the Narration

Next: spoken audio for each slide. Hugging Face hosts text-to-speech models that convert text into an audio file. The audio will sound clear and intelligible — not quite human, but natural enough for a presentation.

Give the agent this prompt, replacing the placeholders with your actual narration text:

- *"Write a Python script using the huggingface_hub library. Use InferenceClient with the token from the HF_TOKEN environment variable. Call client.text_to_speech() with the model facebook/mms-tts-eng to generate narration for each slide. Save each result to presentation-audio/ as slide-01.wav through slide-10.wav. Here are the narration texts for each slide: [paste your one-sentence descriptions]."*

The agent will write the script. Run it the same way:

- *"Run the narration script."*

Audio generation is faster than image generation — expect a few seconds per clip. When it finishes, `presentation-audio/` will have ten `.wav` files ready to play.

**Want a more natural voice?** Ask the agent to try `suno/bark-small` instead — same library, different model, slower but more expressive. Just ask: *"Rewrite the narration script to use the model suno/bark-small and run it again."*

## Update the Presentation

`index.html` is the main file your presentation reads from — it contains all the slide content, image paths, and audio file names. Now the agent will update it with your new content.

- *"Update index.html to replace the existing slides with my ten slides about [YOUR TOPIC]. Each slide should use the matching .jpg illustration from presentation-images/ and the matching .wav narration from presentation-audio/. Use the existing slide layouts — title layout for slide 1, two-column or centered for the rest. Keep the HTML structure exactly as it is; just update the content and asset references."*

The agent will make the edits. When it's done, ask it to check its own work:

- *"Read index.html back and confirm all ten slides are present, each with the correct image and audio file."*

This self-check catches missed references before you open the browser.

## Customise the Animated Background

The Three.js background draws a network of nodes and connecting lines. By default it uses cool blues and greys. Ask the agent to match it to your topic's mood:

- *"Update presentation-bg.js to change the network animation colours to [describe your palette — e.g. 'warm amber and dark brown' or 'deep green and soft white' or 'electric blue on black']. Also update the CSS colour variables in presentation-styles.css to match."*

The agent will edit both files. If the result doesn't feel right, describe what you want more precisely:

- *"The background is too bright. Make the nodes smaller and reduce the line opacity to 30%."*

Iterate as many times as you need. Each change takes a few seconds.

## Preview in Your Browser

Before opening your browser, confirm three things:

1. The `presentation-images/` folder contains 10 `.jpg` files (slide-01.jpg through slide-10.jpg)
2. The `presentation-audio/` folder contains 10 `.wav` files (slide-01.wav through slide-10.wav)
3. The terminal you used to run the generation scripts is still active in the `web-presenter` directory

If any files are missing, ask the agent: *"Check the presentation-images/ and presentation-audio/ folders. What files are there, and are any missing?"*

Web Presenter needs a local HTTP server — it uses `fetch()` to load audio files, which browsers block for plain `file://` paths. Ask the agent to start one:

- *"Start a local HTTP server in this directory on port 8000."*

It will run:

```
python3 -m http.server 8000
```

You should see output like:

```
Serving HTTP on 0.0.0.0 port 8000
(http://0.0.0.0:8000/) ...
```

That means the server is ready. **Leave this terminal window open** — closing it stops the server. Now open your browser and go to:

```
http://localhost:8000
```

You should see your presentation. Press the spacebar or right arrow key to advance to the next slide. Each slide will show its illustration, play the narration, and advance automatically when the audio finishes. You can also press right arrow to skip ahead manually.

Press `M` to toggle background music, `A` to toggle narration, and `Esc` to stop playback.

When you're done, go back to the terminal and press **Ctrl+C** to stop the server.

## Commit and Push

Once you're happy with your presentation, save your work to GitHub:

- *"Create a branch called presentation/[YOUR TOPIC], add all the changed files, commit with a message describing what I built, and push to my fork."*

The agent will handle every Git step. When it's done, verify on GitHub:

1. Go to `https://github.com/YOUR_USERNAME/web-presenter`

2. Open the branch dropdown and look for your branch (`presentation/[YOUR TOPIC]`)
3. Click it — you should see your new images, audio files, and updated `index.html`

Your presentation is now saved on GitHub and shareable with anyone via that URL.

## What Just Happened

You generated ten AI illustrations, narrated them, customised animated 3D graphics, and updated a web project — all by describing what you wanted in plain English. The agent handled the mechanics. This is what agentic programming looks like.

| **You brought** | **The agent brought** |
|---|---|
| A topic you care about | API calls to Hugging Face |
| Ten sentences of content | A working Python script |
| A colour palette description | Updated JavaScript and CSS |
| Judgment on what looks right | The mechanics of making it so |

The skill isn't coding. It's knowing what to ask for, how to check the result, and how to iterate when it's not quite right. That's what the next chapters will build on.

## Troubleshooting

**Image generation returns an error about the model being too busy:** The free inference API has rate limits. Wait 30 seconds and try again, or ask the agent to add a delay: *"Add a 10-second sleep between each image generation call."*

**Audio files are silent or not playing:** The TTS model returns WAV files, not MP3s. If index.html references `.mp3` extensions, ask the agent: *"Check whether the audio src attributes in index.html use .wav extensions. Update any that don't match the files in presentation-audio/."* Also confirm file sizes — any `.wav` file under 5KB is probably an error response that needs regenerating.

**Slides show broken image icons:** The image path in the HTML doesn't match the actual filename. Ask: *"Check all image src attributes in index.html against the actual files in presentation-images/. Fix any mismatches."*

**My HF_TOKEN isn't found by the script:** The variable was set in a different terminal session. Set it again in the current session (`export HF_TOKEN="..."`), then run the script again.

**Nothing loads or the page is blank:** Make sure the server is running in the `web-presenter` directory, not a parent folder. Open your browser's developer tools (F12) and check the Console tab — missing files are listed there. Tell the agent what you see and it will fix them.

**The Three.js animation has stopped working after editing:** Ask the agent: *"The background animation has stopped working. Read presentation-bg.js and check for any syntax errors or missing function calls."*

**The wrong branch shows on GitHub after pushing:** Make sure the branch name has no spaces — use hyphens instead. Ask the agent: *"What branch did we push? Show me the output of git branch -a."*

If you hit an error not listed here, describe it to your AI agent — it's seen most errors before and will usually know what to do.

## Quick Reference

| **Task** | **Prompt or command** |
|---|---|
| Verify HF token is set | `echo $HF_TOKEN` |
| Explore the project | *"Read index.html and explain how slides work"* |
| Install HF library | `pip install huggingface_hub` |
| Generate images | *"Use InferenceClient.text_to_image() with FLUX.1-schnell…"* |
| Generate narration | *"Use InferenceClient.text_to_speech() with facebook/mms-tts-eng…"* |
| Update slide content | *"Update index.html with my ten slides…"* |
| Change background | *"Update the animation colours to…"* |
| Check generated files | *"List the files in presentation-images/ and presentation-audio/"* |
| Start local server | `python3 -m http.server 8000` |
| View presentation | `http://localhost:8000` |
| Stop the server | `Ctrl+C` in the server terminal |

*The best way to learn*
*is to ship something real.*
*This book is your first commit.*