Your Dev Environment: A Guide for AI-Assisted Developers

Castor canadensis
(The North American Beaver)

Lee Harrington

Your Dev Environment

A Guide for AI-Assisted Developers

Lee Harrington

Introduction

You cloned the project. You asked your AI coding tool to get it running. It wrote a setup script, installed some packages, maybe even edited a config file. You ran the command it gave you.

And then something broke. An error message you've never seen before. Something about a Python version, or a missing module, or a port that's already in use. You asked the AI to fix it. It gave you another command. You ran that one too. Now there are two error messages.

This is the wall. Not the code — you can write code. The wall is the layer underneath the code: the environment your code runs in. And when that layer breaks, your AI coding tool gets quieter, more circular, or more confidently wrong.

This book is about that layer.

What You're Actually Looking At

The project you cloned lives in a folder on your machine. But it also depends on things that aren't in that folder: a specific version of Python, a collection of JavaScript packages, a set of access credentials for GitHub, a terminal shell with the right paths configured. When the project runs, it's not just running your code. It's running your code inside a specific environment. That environment either matches what the project expects, or it doesn't.

When it doesn't match, you get errors that have nothing to do with your code. Errors that say things like `command not found` or `module 'xyz' not found` or `ENOENT: no such file or directory`. These aren't bugs. They're mismatches. And the fix isn't writing better code — it's understanding what the environment is supposed to look like.

That's the thesis of this book: you don't need to be a sysadmin. You need to recognize what you're looking at.

Your AI coding tool is good at writing code. It can generate a function, refactor a class, write a test suite. What it wasn't designed to do — not alone, anyway — is understand your specific machine, your shell configuration, your credential store, the Python version you installed three months ago and forgot about. That context lives on your computer, not in the AI's context window. When something in the environment breaks, the AI can guess at it. But you're the one who knows what's actually there.

This book gives you the mental models to understand what's actually there.

The Project

Throughout this book, you'll be working through a single scenario. You've cloned a project called `neighborhood-meals` — a small full-stack web app for managing community meal pickups. It has a Python backend and a JavaScript frontend.

This is a real companion repository for the book, not a made-up example name. The book-facing branch is `main`, which intentionally preserves setup and workflow friction so you can see the problems this book is about. There's also a `reference-working` branch — a known-good baseline you can use later when you need to answer a different question: "is this the environment, or is the app itself broken?"

```
neighborhood-meals/
  README.md
  backend/
    app.py
    pyproject.toml
    requirements.txt
  frontend/
    package.json
    src/
      App.jsx
```

The project doesn't run cleanly on the first try. That's intentional. The errors you hit in this scenario are the same category of errors you'll hit on real projects. Each part of the book is a wall: first git gets confusing, then the Python environment is wrong, then Node throws a version

mismatch, then there's a system-level problem that requires a different tool entirely.

You're not memorizing solutions to these specific errors. You're building the mental map that lets you recognize what kind of problem you're looking at — and tell your AI what it actually needs to know to help.

What Your AI Handles vs. What You Need to Own

Your AI coding tool is a code tool. It thinks in functions, files, and syntax. When you say "add a search feature" or "fix this bug," it knows what to do. That's its terrain.

The environment is different terrain. The environment is the operating system's concern, the shell's concern, the package manager's concern. It involves things like which version of a language runtime is active, how authentication tokens are stored, which directory your shell is currently pointed at. These are not code problems. They're infrastructure problems.

Your AI can often fix infrastructure problems when you describe them accurately. But here's the catch: to describe them accurately, you need to understand what you're looking at. If you just paste an error message and say "fix this," the AI is working blind. It might get lucky. It might send you down a path that fixes the symptom and breaks something else.

The division of responsibility in this book is: the AI runs the commands, but you know what the commands are doing. Not memorized — understood. There's a difference.

What the Four Parts Cover

Part 1: Git and GitHub

Git is version control — it tracks changes to your code. GitHub is a hosting service where git repositories (folders tracked by git) live online. They're related but different, and the distinction matters when your AI is pushing changes, managing branches, or running into authentication errors.

Part 1 covers what git is tracking, what the common states of confusion look like, and how GitHub authentication works when your AI coding tool is doing the pushing. You'll hit this wall first with `neighborhood-meals`, before you even get the project running.

Part 2: Python Environments

Python has a problem: different projects often need different versions of Python or different sets of packages, and they can't all share the same installation without stepping on each other. The solution is isolated environments — separate containers of packages per project.

When your AI sets up a Python project, it's usually creating one of these environments. Part 2 explains what it's creating, why, and what goes wrong when the environment isn't active or isn't the right version.

Part 3: Node and npm

Node is a JavaScript runtime. npm is its package manager. The `frontend/` directory in `neighborhood-meals` runs on Node, and like Python, Node has a version problem: projects specify which version they expect, and if your machine has a different one, things break in confusing ways.

Part 3 covers how Node versions work, what npm is actually doing when it installs packages, and how to read the errors that come out of a Node version mismatch.

Part 4: Warp for System-Level Problems

There's a category of problem that sits below your AI coding tool's reach: things wrong with the shell configuration, the file system, running processes, or system permissions. Your AI coding tool can tell you what command to run. It can't always see what your system is doing.

Warp is a terminal designed for this kind of diagnostic work. Part 4 covers what Warp does that a standard terminal doesn't, and how to use it when your AI coding tool says "it should work" but it doesn't.

What You Need Before You Start

You need a machine with Git, Python, and Node installed. You need an AI coding tool — Claude, Codex, Cursor, or similar. You don't need to know how to use any of them deeply. You don't need prior experience with virtual environments, package managers, or shell configuration.

If you can open a terminal and run a command when someone tells you what it is, you're ready.

A Note on Commands

This book shows terminal commands. You are not expected to memorize them. When a command appears, the point is to understand what it's doing — not to write it down for later. Your AI will write the commands. Your job is to know whether the result is what you expected.

When something looks wrong, you'll know what question to ask. That's the skill.

Part 1 starts with the thing that confuses almost everyone who's new to working with an AI coding tool on a real project: git state, and what it means when the AI starts talking about branches and commits before you've even gotten the project running.

Chapter 1: What Git Actually Is

You just cloned `neighborhood-meals` and asked your AI to check the project state. It ran a command and handed you this:

```
On branch main
Your branch is up to date with 'origin/main'.
Changes not staged for commit:
```

```
  (use "git add <file>..." to update what will be staged)
  (use "git restore <file>..." to discard changes in working directory)
■modified:   frontend/src/App.jsx

Untracked files:
  (use "git add <file>..." to include in what will be committed)
■frontend/src/styles.css

no changes added to commit (use "git add" and/or "git commit -a")
```

If you don't know what you're looking at, this is noise. Words like "staged," "untracked," and "working directory" have specific meanings here, and without them the output tells you nothing. Or maybe you didn't even get that far — maybe you opened the terminal in the wrong folder and got this instead:

```
fatal: not a git repository (or any of the parent directories): .git
```

That one stops everything. The AI can't do git work from outside the project folder. Before anything else works, you need to know what git is, where it lives, and what it's actually tracking.

Git Is a Snapshot Machine

Most people come to git thinking of it as something like a backup system or a change tracker — a record of what got edited and when. That's close, but it's not right, and the gap between that model and the actual one causes real confusion.

Git is a snapshot machine. Every time you commit, git takes a complete picture of your files at that moment and saves it. Not a record of what changed — a full snapshot. If you look at the history of a project, you're not looking at a list of diffs. You're looking at a stack of complete states of the project, each one labeled with a message, a timestamp, and a unique ID.

The practical implication: you can go back to any of those states exactly. Not reconstruct it — go back to it. The snapshot is there.

This is why the error message `fatal: not a git repository` is specific about what's missing. Git's snapshot history lives in a folder called `.git` inside your project. No `.git` folder means git has

nowhere to read from or write to. It can't operate.

When you ran `git clone` to get `neighborhood-meals`, git created that `.git` folder automatically. It's hidden by default in most file browsers, but it's there, at `neighborhood-meals/.git`. That folder is the database. Every commit you've made, every branch, every piece of the project's history — it's in there.

> **Don't Do This:** Do not delete the `.git` folder. There's no reason to, but sometimes people do it when cleaning up a project or troubleshooting. If you delete it, you lose all history. The AI knows not to touch `.git`. You should too.

The Three Places Your Code Lives

At any moment, the files in your project exist in one of three places. Understanding these three places is what makes git's output legible.

The working directory is your filesystem. The files you open in your editor, the files you can see and change. When you edit `frontend/src/App.jsx`, that change exists only here. Git hasn't heard about it yet.

The staging area is a holding zone. When the AI runs `git add`, it moves changes from the working directory into staging. Think of it as a preparation table: you're assembling what you want the next snapshot to contain. A file can be partially staged — some of its changes going in, some not — but practically speaking, you'll usually see the AI add whole files.

The committed history is the snapshot stack. When the AI runs `git commit`, it takes everything in staging and locks it into a new snapshot. That snapshot goes into `.git`. It's permanent — or as close to permanent as digital things get.

The output from `git status` is a report on where your files are. That's all it is. The "Changes not staged for commit" section is telling you what's in the working directory but not in staging. The "Untracked

files" section is telling you what files git has never seen — they're not even in staging consideration yet.

A clean `git status` looks like this:

```
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

That last line is the signal. "Working tree clean" means nothing is in the working directory that isn't also reflected in the last snapshot. No uncommitted edits, no untracked files.

A dirty `git status` means something is in the working directory that hasn't been committed. It's not an error. It's a report.

The Five Commands You'll Watch the AI Use

Your AI is going to run these five commands constantly. You don't need to memorize them — you need to know what each one is doing when you see it.

`git status` — Checks the current state. What's modified, what's staged, what's untracked. The AI runs this before almost every other git operation to understand what it's working with.

`git add` — Moves changes from the working directory into staging. `git add frontend/src/App.jsx` stages one file. `git add .` stages everything in the current directory. The AI will usually be specific about what it adds.

`git commit` — Takes everything in staging and saves it as a new snapshot. Always comes with a message describing what changed: `git commit -m "update app styles"`. Without a message, git refuses.

`git log` — Shows the snapshot history. Each entry has a commit ID (a long string of letters and numbers), an author, a timestamp, and the message. The AI reads this to understand where the project came from and to reference specific commits.

`git diff` — Compares states. With no arguments, it shows the difference between the working directory and staging. With commit IDs, it compares two snapshots. The AI uses this to understand what changed before deciding what to commit.

None of these commands change anything on GitHub. They're all local. Your working directory, your staging area, your committed history — all of it lives on your machine. GitHub is separate.

What the Wrong-Directory Error Is Actually Saying

Back to that error:

```
fatal: not a git repository (or any of the parent directories): .git
```

Git looked for a `.git` folder in your current directory, then in every parent directory above it. It found nothing. The project's git database doesn't exist at this location.

This usually happens one of two ways. Either you're in the wrong directory — maybe you're in your home folder or in `Desktop` instead of `neighborhood-meals` — or you're in a subdirectory of the project that git is treating as unrelated. Running `git status` from inside `neighborhood-meals/backend/` works fine. But if you opened a new terminal window and forgot to navigate back into the project, you'll get this.

> **What the AI Does:** When the AI sees this error, it checks your current directory, finds the correct project path, and re-runs the command from there. It may run `cd neighborhood-meals` first, or it may note the path in its next command. If it can't find the project, it will ask you where it is.

> **Watch For:** After the AI navigates to the correct folder, the next `git status` should return output about your project — branch name, file states — instead of the `fatal:` error.

The fix is not a git fix. It's a navigation fix. That's worth naming, because a lot of git errors are actually shell errors in disguise.

The Reader's One Job

The AI handles the commands. It decides when to add, when to commit, when to check status. What you need to own is one piece of awareness: is `git status` showing a clean working tree or a dirty one?

Clean means the AI is working from a stable baseline — the last committed snapshot reflects what's on disk. Dirty means there's uncommitted work somewhere, and the AI needs to account for it before doing things like switching branches or pulling from GitHub.

You don't need to know what to do about it. You need to notice it. If your AI is about to do something and you see untracked or modified files in the status output, that's the moment to say: "there are uncommitted changes here — should we commit or discard them first?" The AI will take it from there.

> **Key Takeaway:** Git saves snapshots of your project, not diffs. The `.git` folder is the database. The three areas — working directory, staging, committed history — are where your files live at any moment. A clean working tree means git's snapshot matches your disk. A dirty one means it doesn't yet.

Now that you know what git is tracking, the question is where that tracking lives when you share your code.

Chapter 2: GitHub — Where Your Code Lives

You asked your AI to push some changes. It ran a command and referenced something called `origin`. Or maybe it told you the push succeeded, and you went to look for your code on GitHub and couldn't

find it — different repository, different account, not where you expected it. Or the AI mentioned fetching from `origin/main` and you're not sure if that's the same `main` you're looking at in your project folder.

`origin` is not magic. It's a name. But understanding what it names changes how you read everything the AI does with GitHub.

Git Is Local First

This is the thing that trips people up most. Git doesn't require a network connection. Git doesn't require GitHub. When you commit code, it goes into the `.git` folder on your machine. That's it. Nothing leaves your computer unless you explicitly push it somewhere.

GitHub is a separate thing. It's a hosting service — a computer somewhere that stores a copy of your `.git` database and gives it a web interface. When the AI clones `neighborhood-meals`, git copies that `.git` database from GitHub's servers to your machine. From that moment on, you have a complete, independent copy of the project history. You can commit, branch, and review history without ever touching GitHub again.

The connection between your local copy and GitHub exists only when you push or pull. And that connection has a name: `origin`.

What `origin` Is

When you clone a repository, git automatically creates a shorthand name for where you cloned it from. That shorthand is `origin` by default. It points to the URL of the source repository.

For this book's companion repo, the canonical GitHub URL is:

```
https://github.com/leebase/neighborhood-meals.git
```

Depending on how the machine is set up, `origin` may use that HTTPS form directly or the SSH form of the same repository. The important point is not the transport. It's that `origin` names the remote copy of this repo.

To see what `origin` is in `neighborhood-meals`, the AI can run:

```
git remote -v
```

The output looks something like this:

```
origin■git@github.com:leebase/neighborhood-meals.git (fetch)
origin■git@github.com:leebase/neighborhood-meals.git (push)
```

That's it. `origin` is an alias for a URL. When the AI says `git push origin main`, it means: "push the commits on my local `main` branch to the URL called `origin`, into the branch called `main` on that remote."

> **Key Takeaway:** `origin` is not a place — it's a name for a place. It points to a specific URL on GitHub. The AI uses it so you don't have to read the full URL every time.

Nothing about `origin` is automatic after the initial clone. If someone sends you a project folder that wasn't cloned from GitHub — just copied — there might be no `origin` set up at all. If you try to push, you'll get an error saying there's no remote configured. The AI will add one, but it's worth knowing why it's needed.

Local vs. Remote: They Are Not the Same

Your local copy of `neighborhood-meals` and the copy on GitHub are separate databases that happen to share history. Changes you make locally do not appear on GitHub until you push. Changes someone else pushes to GitHub do not appear on your machine until you pull.

This is not a bug. It's the design. Git was built for offline work and distributed teams. The local-remote split is intentional.

The practical effect: if you commit something locally and then look at the GitHub repository, you won't see it. The AI knows this. When it runs `git push origin main`, it's explicitly sending your local commits to GitHub. When it runs `git pull` or `git fetch`, it's bringing GitHub's commits down to your machine.

Watch For: After a successful push, the AI may confirm with output like this:
``
To git@github.com:yourname/neighborhood-meals.git
abc1234..def5678 main -> main
``
That's git confirming it sent commits from your local `main` to the remote's `main`. The two hex strings are commit IDs — where the remote branch was, and where it is now.

If the AI pushed and you don't see this output — or if you see an error — that means nothing was sent, regardless of what the local commits look like.

Public vs. Private Repositories

Every GitHub repository is either public or private. Public means anyone on the internet can view the code. Private means only people you've explicitly given access can see it.

This distinction matters in two places.

The first is obvious: if your project has credentials, API keys, or any secrets hardcoded into files, a public repository exposes them to everyone. The `.gitignore` file in `neighborhood-meals` exists partly for this reason — it lists files that should never be committed. The AI knows to check `.gitignore` before committing, but you should know why it matters.

The second is less obvious: when you share context with an AI tool, you're often pasting code, error messages, or file contents into a conversation window. That conversation may not be private depending on your service settings. Knowing whether your repository is public or private helps you calibrate how careful to be with what you paste.

For `neighborhood-meals`, the repository is private. The AI doesn't need to know your GitHub credentials to work with a private repository — but your machine does, and that's what the next chapter covers.

One more detail matters for this repo specifically: `main` is the book-facing branch, not the "everything is already smooth" branch. If you need to sanity-check whether a failure is environmental or whether the app itself is broken, the known-good branch is `reference-working`. The AI handles the branch switch. You just need to know why that branch exists.

What a Fork Is

You'll see this word eventually, usually when you're working on a project you don't own.

A fork is a personal copy of someone else's repository on GitHub. When you fork a project, GitHub creates a new repository under your account that starts as an identical copy of the original. From that point, it's yours to modify. The original stays untouched.

Forks are common in open-source work: someone publishes a project, you fork it, make changes, then propose those changes back to the original through a pull request.

For `neighborhood-meals`, you probably cloned directly from a repository you own or have write access to. No fork involved. But if you ever see the AI reference an `upstream` remote in addition to `origin`, that means the project was forked — `upstream` points to the original, `origin` points to your fork.

You don't need to manage this. The AI handles it. But when you see `upstream` in a git command, you'll know what it means.

What the README Does

The `README.md` in `neighborhood-meals` is not decoration. It's the front door of the project.

When an AI coding tool starts working on a project, one of the first things it reads — if it has file access — is the README. The README tells it what the project does, how to set it up, what the main commands are, and often what the expected environment looks like. A well-written README gives the AI enough context to make reasonable decisions about what to change and what to leave alone.

For you, the README is orientation. When you can't remember what the backend entry point is, or how to run the tests, or what environment variable you need to set — the README usually has it. The AI will often refer to it. When it does, that's not a stall. That's the AI using available context.

> **Don't Do This:** Don't ignore the README because you're having the AI set things up. The README describes the intended state of the environment. If the AI's setup diverges from the README — installs a different version, skips a step — that's worth noticing. The README is your reference for "what this project is supposed to look like."

Why the Remote Matters for AI-Assisted Work

Your AI coding tool runs commands on your machine. It edits local files, commits to your local git history. But collaboration — getting changes to a teammate, opening a pull request, deploying — requires GitHub. That means pushing to `origin`.

If `origin` isn't configured, or if the machine isn't authorized to push to it, the AI hits a wall. It can keep working locally, but nothing reaches GitHub. That's a limited loop: the AI does work, nothing ships.

When the AI encounters a push failure, it will usually tell you what's wrong — `remote: Repository not found` if the URL is incorrect, or the permission denied error if authentication isn't set up. Those are fixable problems, but they require your machine to be set up correctly. The AI can diagnose them. It can't fix them without your help on the auth side.

> **What the AI Does:** When the AI needs to push and the remote isn't configured, it will run `git remote add origin <url>` with the correct GitHub URL. Once the remote is set, it pushes. If authentication fails after that, you'll see an error — and that's the subject of the next chapter.

What You Own Here

You don't need to manage `origin` manually. The AI handles it. What you need to own is the distinction: when you see changes locally, they are not on GitHub until something pushes them. If you're looking at `neighborhood-meals` on GitHub and not seeing a recent change your AI made, the question isn't "why didn't it save" — it's "did we push."

Ask the AI. It'll check.

Before the AI can push to that remote, your machine needs permission — and that's what the next chapter is about.

Chapter 3: Getting Your Computer Authorized

The AI tried to push your changes to `neighborhood-meals` and got this:

```
git@github.com: Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

Nothing was pushed. GitHub rejected the connection before it even started.

This isn't a git problem. Git did everything right — it built the commit, it connected to the right URL, it tried to authenticate. GitHub looked at

your machine and said no.

Why GitHub Requires Proof

GitHub doesn't recognize your machine by address. Anyone can make a TCP connection to GitHub's servers. What GitHub needs is proof that the machine making requests is actually allowed to access this repository.

For a long time, that proof was a username and password. You'd push, GitHub would ask who you are, you'd type your credentials. GitHub stopped accepting account passwords for git operations in 2021. The reason: passwords can be phished, reused across sites, guessed. GitHub moved to credential models designed for machine-to-machine communication: SSH keys and personal access tokens.

Your AI coding tool is a machine making requests on your behalf. It needs one of these credential models set up on your machine, or it can't push.

SSH Keys: The Mental Model

SSH stands for Secure Shell — a protocol for encrypted communication between machines. The authentication works like a matched lock and key.

When you generate an SSH key pair, you get two files: a private key and a public key. The private key stays on your machine and never leaves. The public key is a string you give to GitHub. When your machine connects to GitHub, GitHub sends a challenge. Your machine uses the private key to answer it. GitHub checks the answer against the public key it has stored. If they match, you're in.

No username. No password. No prompts.

The private key is usually stored at `~/.ssh/id_ed25519` (or `~/.ssh/id_rsa` for older keys). The public key is the corresponding `.pub` file. You give GitHub the contents of the `.pub` file — not the

private key. Never the private key.

The practical upshot: once the public key is on GitHub and the private key is on your machine, every git operation over SSH just works. The exchange happens invisibly. The AI pushes, GitHub checks, the handshake completes.

> **Don't Do This:** Do not regenerate your SSH key pair after setting it up unless you have a specific reason. Regenerating creates a new key pair, which means the old public key on GitHub no longer matches. Every push will fail until you update GitHub with the new public key. The AI may regenerate keys if you ask it to "fix" auth by starting over — confirm this is what you want before it does.

HTTPS Tokens: The Alternative

The other option is HTTPS with a personal access token (PAT). You'll notice this if the project's remote URL looks like:

```
https://github.com/yourname/neighborhood-meals.git
```

rather than:

```
git@github.com:yourname/neighborhood-meals.git
```

The `git@` prefix means SSH. The `https://` prefix means HTTPS.

With HTTPS, git needs a username and a token instead of a password. A personal access token is a long string of characters you generate on GitHub — think of it as a purpose-built application credential. It has specific permissions (read-only, write, etc.) and can be revoked without changing your account password.

On macOS, git can store HTTPS tokens in the system keychain so you don't have to paste them every time. The AI may set this up for you with `git credential-helper osxkeychain` or similar. Once configured, it behaves the same as SSH — the credential exchange happens invisibly.

SSH is more common for developer machines. HTTPS tokens are common in CI systems and automated environments. For

`neighborhood-meals` running locally, SSH is the default setup you'll likely see.

How to Check If Auth Is Set Up

Before the AI tries to push and fails, it can verify auth with one command:

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

This sends a test connection to GitHub using SSH. If auth is working, GitHub responds with:

```
Hi yourname! You've successfully authenticated, but GitHub does not pro
```

That message looks odd but it's correct. GitHub is confirming who you are and reminding you that SSH access to GitHub is for git operations only — not for interactive shell sessions.

If auth is not working, you get the same error as the push failure:

```
git@github.com: Permission denied (publickey).
```

> **What the AI Does:** The AI will run `ssh -T git@github.com` as a diagnostic step when it suspects an auth problem. If the test fails, it will walk you through generating an SSH key pair and adding the public key to GitHub. The steps involve: running `ssh-keygen` to create the key pair, copying the public key with `cat ~/.ssh/id_ed25519.pub`, and going to GitHub's SSH key settings page to paste it in.

> **Watch For:** After you add the public key to GitHub and rerun `ssh -T git@github.com`, you should see the "Hi yourname!" confirmation. That's the signal that auth is fixed. The AI can then push without needing anything else.

Walking Through the Setup

If you hit the permission denied error and need to set up SSH from scratch, the AI will run something like this sequence:

First, check if any SSH keys already exist:

```
ls ~/.ssh
```

If you see `id_ed25519` and `id_ed25519.pub` already, the key pair exists. The issue might be that the public key isn't on GitHub yet — or that you're using a different account than you think.

If there are no keys, the AI will generate them:

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

This creates the key pair at `~/.ssh/id_ed25519`. It will ask for a passphrase — you can leave it empty for local development use, or set one for extra security. The AI will handle the prompts.

Then it reads the public key:

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

The output is a long string starting with `ssh-ed25519`. That entire string gets added to GitHub at: Settings → SSH and GPG keys → New SSH key.

After you paste it in and save, the `ssh -T git@github.com` test should succeed.

One-Time Setup

This is a one-time configuration per machine. Once SSH is working on your laptop, you don't touch it again. The AI won't ask you to redo it. If you get a new machine, you do it again — either transfer the key or generate a new one and add it to GitHub.

The frustrating thing about auth errors is that they feel like code problems because they happen while you're doing code work. They're not. They're infrastructure configuration, and they have clean binary outcomes: either the key is set up and pushes work, or it isn't and pushes fail. There's no partial state to diagnose.

What You Own

Two things.

First, following the setup steps. The AI will guide you through them, but it can't add the public key to GitHub on your behalf. That requires you to be logged into GitHub in a browser. The AI will tell you exactly what to paste and where.

Second, not undoing the setup. The two ways auth breaks after being correctly set up: regenerating the key pair (new private key, old public key on GitHub no longer matches), or changing the remote URL from `git@github.com:...` (SSH) to `https://github.com/...` (HTTPS) without configuring the HTTPS credential helper. Either change breaks auth silently — git will try to connect and fail, and the error will look the same as if nothing was ever set up.

If the AI suggests changing the remote URL for some reason, ask why. That's not a routine operation.

> **Key Takeaway:** SSH auth is a matched key pair: the private key on your machine, the public key on GitHub. The test command is `ssh -T git@github.com`. Success looks like "Hi yourname!" — that line confirms the handshake worked. Setup is one-time per machine.

Now that the machine is authorized, the next thing you'll notice is that the AI rarely works directly on `main` — it creates branches, and sometimes something called a worktree.

Chapter 4: Branches and Worktrees

You asked your AI to add a theme switcher to `neighborhood-meals`. It ran some commands and then you noticed your terminal prompt changed — it now shows `feat/theme-switcher` where it used to show `main`. Or maybe you ran `git status` yourself and saw:

```
On branch feat/theme-switcher
nothing to commit, working tree clean
```

Nobody asked it to create a branch. It just did. And now you're not on `main` anymore.

Or you looked at your filesystem and found a second copy of the project:

```
~/code/neighborhood-meals/
~/code/neighborhood-meals-theme-switcher/
```

Two project folders. The AI created one you didn't ask for, and you're not sure if it's a duplicate, a backup, or something that matters.

Both of these things are intentional. They're how the AI protects your work.

What a Branch Actually Is

The word "branch" suggests a copy — a diverging path of code. That's the intuition, but it's not how git implements it.

A branch is a label. It's a pointer to a specific commit in your snapshot history. That's all. When you create a new branch, you're not copying any files. You're creating a new label that points to the commit you're currently on.

When the AI created `feat/theme-switcher`, it did something equivalent to putting a sticky note on the current snapshot that says "this is where theme-switcher work starts." Then it moved to that label and began committing. Each new commit on `feat/theme-switcher` advances that label forward, while `main` stays pointing at the last commit before the branch was created.

The practical result: `main` doesn't change while the AI works on `feat/theme-switcher`. When the feature is ready and reviewed, the AI merges the branch — bringing those commits into `main`'s history. Until then, `main` is clean.

You can see all branches with:

```
git branch
```

The output will look like:

```
* feat/theme-switcher
  main
```

The `*` marks the branch you're currently on.

Why the AI Uses Branches

The AI creates a branch for feature work because `main` is supposed to be stable. In most projects, `main` represents code that works — code that could be deployed or shared right now. If the AI commits in-progress, experimental, or broken code directly to `main`, it corrupts that baseline.

On `feat/theme-switcher`, the AI can make commits freely, break things, fix them, make five commits and undo three of them, without `main` ever seeing any of it. When the work is done and confirmed to work, merging it into `main` is one clean operation.

This also means if the theme switcher approach doesn't pan out — if you decide to scrap it — you delete the branch and `main` is exactly where it was. No cleanup needed.

> **What the AI Does:** Before starting any significant feature, the AI will typically run `git checkout -b feat/feature-name` (or `git switch -c feat/feature-name` on newer git). This creates the branch and moves to it in one step. It will tell you what branch it's on before it starts writing code.

What Actually Changes When the AI Switches Branches

This is the part that feels spooky the first few times you see it.

The files in your folder can change even when you didn't edit them manually. That's because the branch determines which snapshot git puts on disk. If `main` points to one snapshot and

`feat/theme-switcher` points to a later one, switching branches changes which version of the files you are looking at.

That doesn't mean the AI created a second secret project state out of nowhere. It means git replaced one checked-out snapshot with another. Same repository. Same history. Different point in that history made visible in the folder.

This is why the terminal prompt matters. When your prompt shows `feat/theme-switcher`, that's not decoration. It's telling you which snapshot label is currently active. If you ever feel disoriented, the grounding command is:

```
git status
```

It tells you three things that matter immediately: which branch you're on, whether files have been changed, and whether there is anything uncommitted that would block a switch. For a new developer, that's often enough context to stop the panic. You don't need to reconstruct git's entire model. You just need to know where you are.

What a Worktree Is

A worktree is a second checked-out copy of the same repository, in a different folder.

When you have `neighborhood-meals-theme-switcher` appearing next to `neighborhood-meals`, that's a worktree. Both folders share the same `.git` database — there's only one copy of the project history. But each folder has a different branch checked out, so the files on disk are different.

The AI uses worktrees when it needs to work on two things at once without switching back and forth. Maybe it's building the theme switcher in one worktree while the main project is open and running in the other. Switching branches in a single checkout would interrupt whatever is running. With worktrees, it doesn't have to.

To see all worktrees, the AI can run:

```
git worktree list
```

The output shows both locations:

```
/Users/you/code/neighborhood-meals              abc1234 [main]
/Users/you/code/neighborhood-meals-theme-switcher  def5678 [feat/theme-
```

The first column is the path on disk. The second is the current commit ID in that worktree. The third is the branch name in brackets.

> **Key Takeaway:** A worktree is not a duplicate project. It's the same project, same history, second folder. The `.git` folder in `neighborhood-meals` is shared. The `neighborhood-meals-theme-switcher` folder exists because the AI needed to work on `feat/theme-switcher` separately. You can treat it as a window into that branch.

How to Stay Oriented When a Worktree Appears

The confusing part about worktrees is not the git model. It's the filesystem feeling. You see two folders with almost the same name and your first reaction is reasonable: which one is real?

Both are real. The question is which branch each one is showing you.

When this happens, don't inspect random files hoping the answer becomes obvious. Ask git directly:

```
git worktree list
```

That output gives you the map. One path will show `[main]`. One path will show `[feat/theme-switcher]`. Once you know that, the folders stop feeling mysterious. One is your stable baseline. One is the feature workspace. The AI may have your app running from one while editing code in the other. That's not duplication. That's separation of responsibilities.

The important thing to recognize is that deleting the feature folder manually in Finder is not the same as cleaning up a worktree correctly. If the AI created it with `git worktree`, the AI should remove it with `git worktree remove` when it's done. Let git clean up git's own

structures.

The Worktree Wall

If you try to check out `feat/theme-switcher` in your main `neighborhood-meals` folder while a worktree already has it checked out, git will stop you:

```
fatal: 'feat/theme-switcher' is already checked out at '/Users/you/code
```

Wait — that error message says it's checked out at `/Users/you/code/neighborhood-meals`, but that's the main folder, not the worktree. The error message in older versions of git can be confusing. What git is actually telling you is: this branch is already in use somewhere — it can't be in two places at once.

A branch can only be checked out in one worktree at a time. If `feat/theme-switcher` is in the `neighborhood-meals-theme-switcher` worktree, you can't also check it out in `neighborhood-meals`. Git protects you from this because two checked-out copies of the same branch would diverge immediately and create a conflict.

> **What the AI Does:** When it hits this error, the AI will check `git worktree list`, find the worktree that has the branch, and either switch to working in that directory or remove the worktree if it's no longer needed. It won't try to force the checkout.

> **Watch For:** If the AI runs `git worktree remove ../neighborhood-meals-theme-switcher`, it's cleaning up a worktree that's done. The folder will disappear from your filesystem. That's expected. The branch and its commits still exist in `.git` — the worktree was just the window into it.

The Dirty Working Tree During a Branch Switch

There's a related wall that comes up around branches — not specific to worktrees, but common enough to cover here.

You edited `frontend/src/App.jsx`. Then the AI tried to switch branches and got stopped:

```
error: Your local changes to the following files would be overwritten b
■frontend/src/App.jsx
Please commit your changes or stash them before you switch branches.
Aborting
```

Git is not being difficult. It's protecting you. The file on disk has changes that haven't been committed. If git switches to a different branch — where `App.jsx` looks different — it would overwrite your edits and they'd be gone. Git refuses to do that without explicit instruction.

The options are: commit the changes first (make them a permanent snapshot), or stash them (save them temporarily in a git-managed holding area, switch branches, then restore them afterward). The AI will usually commit if the changes are meaningful, or stash if they're scratchwork.

This error isn't a sign that something went wrong with your setup. It's git working as designed.

What You Don't Need to Manage

The AI creates branches. The AI creates worktrees. The AI merges, deletes, and cleans up. You don't need to know the commands to do any of this from scratch.

What you need is the recognition layer: if you see a second folder appear, it's a worktree and it's not a mistake. If you see a branch name you didn't create, the AI made it intentionally. If you see the worktree wall error, you know what it means and can tell the AI "there's a worktree conflict — you'll need to sort out which copy to use."

The model is: branch = label on a snapshot, worktree = second folder looking at a different label. Everything else follows from those two

definitions.

Branches protect `main`, but they can't protect you from making changes you didn't mean to — and that's where git's safety mechanisms become visible.

Chapter 5: When Git Goes Wrong

You edited `frontend/src/App.jsx` directly — you were experimenting with a color change, just looking at it. Then the AI tried to switch to a different branch and git stopped everything:

```
error: Your local changes to the following files would be overwritten b
■frontend/src/App.jsx
Please commit your changes or stash them before you switch branches.
Aborting
```

Nothing broke. No data was lost. Git saw the unsaved edit and refused to proceed. But the AI is stuck now, and you need to understand what happened before deciding how to proceed.

Git has a small set of error states that come up repeatedly. Each one has a distinct message, a clear cause, and a specific resolution. Once you've seen them once, you'll recognize them on sight.

Error State 1: Dirty Working Tree

This is the error shown above. You have uncommitted changes in the working directory, and git is refusing to do something that would overwrite them.

What git is protecting: Your edits. The file on disk contains changes that don't exist anywhere in git's snapshot history. If git proceeds with the branch switch, it would replace your version of the file with the version from the other branch. Those edits disappear. Git won't do that silently.

What the AI does: The AI has two options. If the changes are worth keeping — if they're meaningful work, even incomplete — it commits them first:

```
git add frontend/src/App.jsx
git commit -m "wip: color experiment"
```

Now the changes are in a snapshot. The branch switch proceeds.

If the changes are not worth keeping — you were poking at something and don't need the result — the AI discards them:

```
git restore frontend/src/App.jsx
```

This brings the file back to the last committed state. The edits are gone. The branch switch proceeds.

A third option is stashing: `git stash` saves the changes in a temporary holding area outside your commit history, lets the branch switch happen, and you can retrieve them later with `git stash pop`. The AI uses this when the changes matter but you're not ready to commit them.

> **Watch For:** After the AI commits, stashes, or restores, it will run the branch switch again. This time it should succeed silently — no output except the confirmation that you're on the new branch. A clean switch means the dirty working tree is resolved.

Error State 2: Detached HEAD

You run `git status` and see:

```
HEAD detached at abc1234
nothing to commit, working tree clean
```

Or maybe the AI mentioned it was in detached HEAD state and you don't know what that means.

What git is telling you: You're not on a branch. You're floating on a specific commit.

Normally, when you're on a branch, your position is tied to a label. When you commit, that label advances — `main` or

`feat/theme-switcher` moves forward with you. In detached HEAD state, there's no label attached to your position. If you commit here, those commits are not connected to any branch. They exist in the `.git` database temporarily, but git can't easily find them and will eventually discard them during cleanup.

This usually happens when the AI checks out a specific commit by its ID rather than by branch name. It's useful for inspecting history but risky if you start working.

What the AI does: If it needs to do work from a detached HEAD, it creates a branch at that position:

```
git checkout -b recovery-branch
```

This attaches a label to where you are now. Any commits you make from here are saved under that branch name. Nothing gets lost.

> **Watch For:** After the AI creates a branch, `git status` should show `On branch recovery-branch` (or whatever name it chose) instead of `HEAD detached at...`. That means you're anchored. Commits are now safe.

> **Don't Do This:** Do not commit in detached HEAD state without asking the AI to anchor you first. It looks like it works — the commit goes through — but if you switch branches after, those commits become orphaned. They're hard to find and will eventually be garbage-collected by git.

Error State 3: Merge Conflict

Two people (or the AI on two branches) edited the same part of the same file. Now git is trying to combine them and doesn't know which version to use:

```
CONFLICT (content): Merge conflict in frontend/src/App.jsx
Automatic merge failed; fix conflicts and then commit the result.
```

If you open `frontend/src/App.jsx`, you'll see this inside it:

```
<<<<<<< HEAD
```

```
  background-color: #ffffff;
=======
  background-color: #1a1a2e;
>>>>>>> feat/theme-switcher
```

What git is telling you: The section between `<<<<<<< HEAD` and `=======` is the version from the branch you're merging into. The section between `=======` and `>>>>>>> feat/theme-switcher` is the version from the branch you're merging in. Git doesn't know which one is right, or whether you want to combine them somehow.

What the AI does: The AI reads both versions, understands what each one is doing, and resolves the conflict — either by choosing one version, or by writing a third version that incorporates both intentions. It then removes the conflict markers, saves the file, and runs:

```
git add frontend/src/App.jsx
git commit
```

That commit finalizes the merge.

> **Watch For:** After the AI resolves conflicts and commits, run `git status`. It should show `nothing to commit, working tree clean` with no mention of conflicts. If any file still shows `UU` (unmerged) in the status output, there are unresolved conflicts remaining.

> **Don't Do This:** Don't edit a conflict-marked file manually unless you know exactly what you're doing. The conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`) are git's syntax. If you accidentally delete one marker but not the others, git's merge state gets corrupted and the resolution process fails.

Error State 4: Rejected Push (Non-Fast-Forward)

The AI tried to push to GitHub and got:

```
 ! [rejected]        main -> main (non-fast-forward)
error: failed to push some refs to 'git@github.com:yourname/neighborhoo
hint: Updates were rejected because the remote contains work that you d
```

```
hint: Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
```

What git is telling you: Someone else pushed commits to `main` on GitHub after you last pulled. Your local `main` and the remote `main` have diverged. Git won't push because that would overwrite the other person's work.

This happens in team contexts — or if you pushed from another machine and forgot to pull on this one. The remote has commits your local copy doesn't have. Before your commits can go up, you need to bring those commits down first.

What the AI does: It pulls first:

```
git pull origin main
```

or, more precisely, it fetches and then rebases or merges. `git pull --rebase origin main` is a common pattern — it brings down the remote commits, then replays your local commits on top of them. The result is a clean linear history.

After the pull succeeds, the push goes through.

Watch For: After a successful pull and push, the output should include a line like:
``
To git@github.com:yourname/neighborhood-meals.git
abc1234..def5678 main -> main
``
If the push still fails after a pull, something else is wrong — possibly a force-push restriction on the branch (common on `main` in team repositories). The AI will tell you if that's the case.

Don't Do This: Do not run `git push --force` unless the AI explicitly tells you to and explains why. Force-pushing rewrites the remote's history. If anyone else pulled that branch, their copy is now out of sync in a way that's hard to recover from. It's occasionally necessary, but it's not a default fix for a rejected push.

The Pattern Across All Four

Every one of these errors follows the same structure. Git encountered something it couldn't proceed past safely, and it stopped and told you exactly what it found.

Git doesn't silently overwrite your work. It doesn't guess. It doesn't proceed when things are ambiguous. The errors are not failures — they're reports. The AI reads them, identifies what git is protecting or what state it found, and takes the appropriate action.

Your part in this is the same as it's been throughout Part 1: recognize which error you're looking at. When you can name the state — dirty working tree, detached HEAD, merge conflict, rejected push — you can tell the AI what it's dealing with. That's faster and more accurate than pasting the full error and asking "what does this mean."

These four states cover most of what you'll see. There are edge cases beyond them, but the pattern is consistent: git tells you exactly what it found, in words you can read. The message is not obscure. It's precise.

> **Key Takeaway:** Git stops rather than overwrites. Every error state in this chapter is git protecting something — your uncommitted edits, the integrity of the history, someone else's pushed work. When the AI resolves one of these states, it's not fighting git. It's working with the information git gave it.

Git is the foundation, but once the code is in the right place, the next wall is usually not about the code at all — it's about whether the right Python is running it.

Chapter 6: Why Python Has an Environment Problem

You cloned `neighborhood-meals`. Git is sorted. Now you want to run the backend. You ask the AI to start it, and it tries `python backend/app.py`. This comes back immediately:

```
Traceback (most recent call last):
  File "/Users/you/code/neighborhood-meals/backend/app.py", line 1, in
    from flask import Flask
ModuleNotFoundError: No module named 'flask'
```

Python is installed on your machine. You can run `python --version` and get an answer. So why doesn't it work?

The error isn't telling you Python is missing. It's telling you that Flask — the library the backend is built on — isn't available to the Python that just ran. Those are two different things.

Python Is Installed, But That's Not Enough

When you install Python on your machine, you get a Python interpreter. That's the program that reads and runs Python code. It comes with a small set of built-in tools. It does not come with Flask, or SQLAlchemy, or any of the libraries that `neighborhood-meals` needs to function.

Libraries get installed separately. When you install a library, it gets placed somewhere on your machine — a folder that the Python interpreter looks in when your code says `from flask import Flask`. If Flask isn't in that folder, you get the error above.

Here's the part that trips people up: Python can be installed in multiple places on the same machine, and each installation has its own separate folder of libraries. Running `python backend/app.py` uses whichever Python your terminal found first. That might not be the one with Flask installed. Or more precisely, it probably isn't — because Flask hasn't been installed anywhere yet for this project.

That's the problem. Not that Python is missing. Not that Flask doesn't exist. It's that this project's libraries haven't been installed into an isolated space that this project uses.

The Mental Model: Rooms

Think of each Python environment as a room. The room has its own Python interpreter and its own set of packages on the shelves. When your code runs, it runs inside one specific room. If Flask isn't on the shelves in that room, the code can't find it — even if Flask is sitting in a completely different room down the hall.

Right now, `neighborhood-meals` doesn't have a room. When the AI tried to run `backend/app.py`, Python went looking for Flask in whatever generic room the terminal happened to be using. Flask wasn't there. Hence the error.

The fix isn't to install Flask globally — that is, into the default room every Python program shares. That approach breaks eventually. If one project needs Flask 2.x and another needs Flask 3.x, they can't share the same room. They'll conflict.

The fix is to build `neighborhood-meals` its own room. In Python, that room is called a virtual environment.

Why Projects Need Their Own Rooms

Picture two projects on the same machine. Project A is an older app that depends on Flask 2.3. Project B is `neighborhood-meals`, which needs Flask 3.0. If you install both into the global Python, one overwrites the other. Both can't be the current version simultaneously.

Virtual environments solve this by giving each project a private folder — usually named `.venv` — that contains its own copy of an interpreter and its own library shelves. Project A's `.venv` has Flask 2.3. Project B's `.venv` has Flask 3.0. They don't know about each other. They can't conflict.

This is the reason Python environments exist. Not because someone wanted to make things complicated, but because real projects live on machines with real constraints and real version disagreements.

What the AI Does

When the AI sees that `ModuleNotFoundError`, it doesn't just try to install Flask. It first checks whether the project has a virtual environment set up at all.

What the AI Does:
The AI checks for `backend/.venv`. That directory doesn't exist yet. So it creates the virtual environment first, then installs the project's dependencies into it.
The commands look like this:
``
cd backend
uv venv .venv
uv pip install -e .
``
You'll see output like:
``
Using Python 3.11.8
Creating virtual environment at: .venv
Resolved 12 packages in 0.4s
Installed 12 packages in 1.2s
+ flask==3.0.3
+ werkzeug==3.0.2
...
``

The `uv venv .venv` command creates the room. The `uv pip install -e .` command reads the project's package list and puts the right libraries on the shelves. After this, `backend/.venv` exists on disk, and Flask is inside it.

The next time you run the backend, Python will use that room. The error will be gone.

> **Watch For:**
>
> The line `Creating virtual environment at: .venv` confirms the room was built. The list of installed packages confirms the shelves were stocked. If you see those, the setup succeeded.

What "Activation" Actually Changes

You'll often see one more step after the environment is created:

```
source .venv/bin/activate
```

This is the part that feels abstract until someone names it plainly. Activating the environment changes which Python your current shell reaches for by default. Before activation, typing `python` might mean the system Python, or a Homebrew Python, or whatever interpreter your shell finds first on PATH. After activation, typing `python` means the interpreter inside `backend/.venv`.

Nothing magical happened to Python itself. Your shell just changed which room it walks into first.

That's why activation is session-scoped. Open a new terminal and the shell starts over with its normal PATH. The environment isn't "broken" — it's just no longer the default for that new session until the AI activates it again. Once you see activation as a shell-state change rather than a Python concept, the behavior stops feeling random.

The Other Wall: Wrong Python Version

Sometimes the error isn't about a missing library. Sometimes the AI runs the install step and you get this instead:

```
ERROR: Package 'neighborhood-meals-backend' requires a different Pythor
```

This is a different problem. The library isn't missing — the Python version itself is wrong. The project was written for Python 3.11 or

newer, and the machine's default Python is 3.9.6.

This happens because macOS ships with an older Python. Your terminal found that one first. You're not in the right room — you're not even using the right building.

The AI will look for Python 3.11 on the machine. If it's installed somewhere (via Homebrew, pyenv, or similar), it will use that. If it isn't, it will prompt you to install it. Either way, the virtual environment needs to be built with the correct Python version before any packages get installed.

Watch For:
Once this is resolved, the setup output will say `Using Python 3.11.x` rather than `3.9.x`. That's the confirmation that the right Python is running.

What You Own

The AI created `backend/.venv`. You don't need to understand its internals, and you shouldn't need to touch it. But there are two things that will break the setup if you do them:

Don't delete `backend/.venv`. It's a generated folder — nothing in it was handwritten, and nothing irreplaceable lives there. But deleting it means the next time you run the backend, Python goes back to the wrong room. The AI can recreate it, but only if you ask, and only if you know that's what broke.

Don't run Python commands from outside the project's virtual environment accidentally. If you open a new terminal window and run a Python command without the environment active, you're back in the generic room. The error returns. You'll know this happened if the `ModuleNotFoundError` shows up again on a project that was previously working.

There's a subtle version of this mistake that's worth naming. You see the backend running correctly in one terminal, open another terminal to test

something, run `python backend/app.py`, and assume it should work the same way because it's the same project on the same machine. But shells don't share state. The first terminal had the environment active. The second one doesn't. Same repo, different room.

> **Don't Do This:** Do not install Flask or any other library globally with `pip install flask` to make the error go away faster. A global install can mask the problem temporarily, but it doesn't set up the project correctly. The next time someone else clones the project, or you set up a new machine, the install step will fail in the same way.

The reason the global install feels tempting is that it can appear to work immediately. The import error disappears, which makes it feel like the diagnosis was correct. But what you actually did was put Flask on the generic shelf instead of the project shelf. The backend might start, but you've taught yourself the wrong lesson about what the project needs. The next project with different dependencies collides with it, and now the machine feels inconsistent again.

> **Key Takeaway:** "Python is installed" and "this project's packages are installed" are two different conditions. A virtual environment is what connects them — it's the private room where this project's dependencies live.

Now you know what a virtual environment is — but you may have noticed there are several tools that create them, and the AI chose one called `uv`. That choice matters.

Chapter 7: venv, conda, uv — Which One and Why

The AI created the virtual environment using a tool called `uv`. But you've also seen tutorials that show `python -m venv`. Someone on your team mentioned conda. A blog post you found recommended

virtualenv. Are all of these the same thing? Do they produce different results? Can you mix them?

The short answer: they solve the same problem in different ways. You don't need to choose between them for `neighborhood-meals` — the AI already made that call. But you do need to know enough to recognize what you're looking at when the environment comes up in conversation, in documentation, or in a teammate's message.

There Are Several Tools, and They All Create Rooms

The problem from Chapter 6 — each project needs its own isolated Python with its own packages — has been around since Python's early days. Several tools were built to solve it. They all create that private room, but they differ in how much they do, how fast they are, and what kinds of projects they're built for.

Here are the three you'll encounter most often.

python -m venv

`python -m venv` is built into Python. Starting with Python 3.3, every Python installation includes it. You don't install it separately; it comes along for the ride.

When you run `python -m venv .venv`, it creates a `.venv` folder with a self-contained Python interpreter and an empty library shelf. Nothing else. It doesn't read your `pyproject.toml`. It doesn't know what packages you need. It just creates the room. You then install packages separately with `pip`.

This minimalism is its strength and its limitation. It's predictable, well-documented, and works anywhere Python is installed. It's also slower and more manual than newer alternatives. You're doing two steps — create the environment, then install packages — and you manage Python versions separately.

You'll see this tool in older tutorials and many official Python documentation pages. If you're reading something that shows `python -m venv`, that's this tool.

conda

`conda` comes from the Anaconda distribution, which was built for data science and scientific computing. It's different from the others in an important way: it manages Python versions directly, not just packages.

With `python -m venv` and `uv`, you need to have the right Python version installed on your machine before creating the environment. `conda` can fetch and install different Python versions as part of creating an environment. It also manages packages that aren't Python packages at all — system-level libraries that scientific tools sometimes depend on.

This makes `conda` powerful and popular in machine learning and data analysis work, where projects often depend on tools like NumPy, CUDA, or TensorFlow that have complicated non-Python dependencies. For a web backend like `neighborhood-meals`, that extra machinery isn't needed.

`conda` environments are also not typically placed inside the project folder as `.venv`. They live in a central location managed by Anaconda, and you switch between them by name rather than by directory. If a teammate says "activate the base environment" or mentions `conda activate`, this is the tool they're using.

uv

`uv` is newer than both of the above. It was released in 2024 and written in Rust, which is partly why it's fast — environment creation and package installation that takes seconds in `pip` can happen in milliseconds in `uv`.

More importantly for a project like `neighborhood-meals`: `uv` is project-aware. When you run `uv venv .venv` and then `uv pip`

`install -e .`, it reads the project's `pyproject.toml` to understand what packages are needed and what Python version is required. It also creates the environment inside the project folder as `.venv`, which keeps everything local and self-contained.

`uv` is what modern Python projects are moving toward. It's faster, it handles both environment creation and package installation in a unified workflow, and it natively reads the `pyproject.toml` format that newer projects use to declare their dependencies.

Why the AI Used uv for neighborhood-meals

The backend has a `pyproject.toml` that declares its dependencies and Python version requirement. `uv` reads that file natively. Creating the environment and installing the right packages is a single coherent workflow.

Using `python -m venv` would have required the AI to create the environment first, then separately figure out what to install and run `pip install` for each dependency. Using `conda` would have been heavier than the project needs and would have placed the environment outside the project folder, making it harder to keep the project self-contained.

`uv` is the right fit for a web backend with a `pyproject.toml`. That's the call the AI made.

> **Key Takeaway:** The tool used to create the environment doesn't change what a virtual environment is. All three tools make a private room for the project. They differ in speed, scope, and how much they handle automatically. For `neighborhood-meals`, `uv` is the right choice because the project already has a `pyproject.toml` and doesn't need the data-science machinery that `conda` brings.

How to Tell Whether the Environment Is Active

Once the virtual environment is set up, you need to know whether it's actually being used when you run a command. The signal for this is in

the terminal prompt itself.

When a virtual environment is active, your prompt changes. Where it previously showed something like:

```
you@machine neighborhood-meals %
```

It now shows:

```
(.venv) you@machine neighborhood-meals %
```

That `(.venv)` prefix is confirmation. It means the terminal is currently pointing at the Python and packages inside `backend/.venv`. When you — or the AI — runs a Python command, it uses that environment.

The name in parentheses matches the name of the environment directory. If the virtual environment had been named `backend` instead of `.venv`, the prefix would say `(backend)`. The prefix is just the folder name.

Watch For:
When the AI activates the environment, it runs:
``
source backend/.venv/bin/activate
``
The prefix appearing in the prompt is how you confirm activation worked. No prefix means the environment is not active.

What "Activated" Actually Means

The prefix isn't cosmetic. When a virtual environment is activated, the terminal rewires one specific thing: which Python binary it uses.

Your machine has a Python installed somewhere in a system folder. Your terminal finds it by looking through a list of directories — your PATH — until it finds a program named `python`. Normally that's the system Python.

Activation inserts the environment's directory at the front of that list. Now when the terminal looks for `python`, it finds the one inside `.venv` first, before it ever gets to the system Python. Everything you run in that terminal session — every Python command, every package — goes through the environment's Python until you close the session or explicitly deactivate.

You don't need to manage this list yourself. You just need to know that the prefix tells you whether the rewiring happened.

The Environment Doesn't Survive a New Terminal

This is where people get caught. The virtual environment is activated for a session — one terminal window, one shell session. When you close that terminal and open a new one, the activation is gone. The new terminal starts fresh with the system Python.

If the backend was working yesterday and today `ModuleNotFoundError` is back, this is probably why. Nothing broke. The environment still exists on disk. It just isn't active in the new terminal.

> **Don't Do This:** Do not reinstall the entire environment because the error came back. Check whether the prefix is visible in your prompt first. If it isn't, the fix is reactivation, not rebuilding from scratch.

What You Own

The AI handles which tool to use and how to activate the environment. You own one thing: noticing whether the prefix is there.

If you open a new terminal and start working, check the prompt before you ask the AI to run the backend. If `(.venv)` is missing, tell the AI. It will activate the environment before running anything. That takes a second and prevents a cascade of errors.

If the prefix disappears mid-session — which can happen if something resets the shell — the same applies. Mention it. The AI will re-activate.

The rest of the environment management is not your job. The AI knows which commands to run, which tool to use for this project, and how to recover if something is broken. Your job is to report what you see.

The AI knows which tool to use — but it also needs to know what to install, and that's recorded in two files the backend carries around.

Chapter 8: Reading requirements.txt and pyproject.toml

Before the AI installs anything into the virtual environment, it opens two files: `backend/requirements.txt` and `backend/pyproject.toml`. It reads them first. It doesn't just run `pip install flask` and call it done.

If you've watched the AI work, you may have seen it open these files without explanation and then move straight into the install step. This chapter explains what those files say, why they both exist in the same project, and what can go wrong if you edit them without understanding what they declare.

The Packing List

Before a project can run, someone has to tell the installer what to install. These two files do that. Think of them as a packing list: before the AI stocks the shelves in the virtual environment, it checks the list to know what belongs there.

Without these files, the AI would have to guess — or you would have to tell it every package by hand each time. Neither is sustainable. The packing list is how the project carries its own requirements, so that anyone who clones it (you, a teammate, a deployment server) can set up

the same environment.

requirements.txt

`requirements.txt` is the older of the two formats. It's been the default way to list Python dependencies for over a decade. You'll find it in almost every Python project that predates 2022 or so, and still in many that are more recent.

The format is a flat list. Each line is a package name, usually with a version constraint:

```
flask==3.0.0
sqlalchemy>=2.0.0
python-dotenv>=1.0.0
```

The `==` operator means exactly this version. `flask==3.0.0` tells the installer to get Flask 3.0.0 specifically — not 3.0.1, not 3.1.0. This is useful for production projects where you want to make sure every machine installs the exact same version.

The `>=` operator means this version or newer. `sqlalchemy>=2.0.0` tells the installer to get SQLAlchemy 2.0.0 or any later release that satisfies the constraint. This gives more flexibility but less precision.

`requirements.txt` is readable without any special tools. You can open it in any text editor and see exactly what the project needs. That's part of why it became so widely used — it's simple, flat, and self-explanatory.

Its limitation is equally simple: it only knows about packages. It doesn't know what Python version the project needs. It doesn't declare the project's name, its version, or how to build it. It's a list of packages and nothing more.

pyproject.toml

`pyproject.toml` is the newer standard, introduced to replace the patchwork of older configuration formats that Python projects

accumulated over the years. It's richer than `requirements.txt` because it holds more than just the package list.

Open `backend/pyproject.toml` and you'll see something like this:

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "neighborhood-meals-backend"
version = "0.1.0"
requires-python = ">=3.11"
dependencies = [
    "flask>=3.0.0",
    "sqlalchemy>=2.0.0",
    "python-dotenv>=1.0.0",
]
```

The `[project]` section is where the core information lives. The `name` and `version` fields identify the project. The `requires-python` field is critical — it declares the minimum Python version the project was designed for. The `dependencies` list is the equivalent of what lives in `requirements.txt`.

That `requires-python = ">=3.11"` line is not advisory. It's a constraint the installer checks before it installs anything. If the virtual environment was built with Python 3.9, the installer sees that `3.9.6 not in '>=3.11'` and stops. That's the exact error from Chapter 6. The constraint is what triggers it.

`pyproject.toml` also carries configuration for development tools — linters, formatters, test runners — in additional sections. It's the single file modern Python projects use for project-level configuration. `uv` reads it natively, which is one reason the AI prefers `uv` for projects that have it.

Why This Project Has Both

If you look at the `backend/` folder, you'll find both files sitting there. That might seem redundant. It isn't a mistake.

`neighborhood-meals` is in transition. It started with `requirements.txt` as its dependency list, which is how almost all Python projects started. At some point the project added `pyproject.toml` to adopt the newer standard and to declare the Python version requirement. But the `requirements.txt` was left in place because older documentation, scripts, and notes still referenced it.

This coexistence is common. It happens on real projects when teams migrate from one tooling convention to another. Both files are valid. They describe similar things in different formats. The authoritative source for what to install is now `pyproject.toml`, but `requirements.txt` hasn't been deleted because removing it would break any process that still reads it.

When the AI runs the install step, it uses `uv pip install -e .`, which tells `uv` to read `pyproject.toml` as the source of truth. If something reads `requirements.txt` instead — an older script, a teammate's manual install command — it will install from that list. The two files should stay in sync. In `neighborhood-meals`, they do.

In this repo, that coexistence is also part of the teaching surface. The book-facing `main` branch keeps both files present because that is what readers actually encounter on real projects: clean migrations are rare, and transitional toolchains are normal. If you need to verify that the app works without wondering whether the roughness is deliberate, that's what `reference-working` is for. The dependency story is the same; the setup question becomes less ambiguous.

> **Key Takeaway:** Having both files isn't a problem to fix. It's the project in the middle of a migration. The AI knows to read `pyproject.toml` with `uv`. If you see documentation that shows `pip install -r requirements.txt`, that's the older approach working from the older file. Both work; they just serve different toolchains.

What Version Constraints Actually Enforce

The version constraints in these files aren't just documentation. They're instructions the installer checks before it acts.

`flask>=3.0.0` tells the installer: any version of Flask from 3.0.0 onward is acceptable. If Flask 3.1.0 is the latest release when you install, that's what you'll get. This is useful for libraries that follow good release practices — newer patch and minor versions are generally safe.

`flask==3.0.0` pins the version precisely. You'll get exactly 3.0.0 regardless of what newer versions are available. This is useful when you want full reproducibility — every developer and every server installs the exact same thing.

`requires-python = ">=3.11"` is the version constraint for Python itself. The project was written for Python 3.11 and relies on language features or library behaviors that exist in 3.11 but not in older versions. A machine running Python 3.9 doesn't meet that constraint. The installer stops and tells you so.

This is where the wrong-interpreter error comes from. When the system Python is too old, the install fails before any packages get added, because the constraint check happens first.

What the AI Does With These Files

The AI doesn't guess at dependencies. Before it installs anything, it reads these files to understand what the backend actually expects.

What the AI Does:
After creating the virtual environment, the AI runs:
``
uv pip install -e .
``
This tells `uv` to install the current project in editable mode. `uv` reads `pyproject.toml`, checks the `requires-python` constraint against the environment's Python version, then resolves and installs every package in the `dependencies` list.

You'll see output like:

```
Resolved 12 packages in 0.3s
Installed 12 packages in 0.9s
+ flask==3.0.3
+ sqlalchemy==2.0.29
+ python-dotenv==1.0.1
...
```

The packages listed in the output correspond to what `pyproject.toml` declared. If the constraint check passes and all packages resolve, the environment is ready. The AI is not improvising — it's executing the packing list.

Watch For:

The line `Resolved N packages in X.Xs` tells you `uv` successfully read the dependency declarations and found a compatible set. `Installed N packages` tells you they were placed in the virtual environment. Both lines appearing without errors means the install succeeded.

What You Own

The AI manages the install. You own the files the AI reads.

`requirements.txt` and `pyproject.toml` are the source of truth for what the project needs. If you add a new package to `pyproject.toml` correctly, the AI will install it on the next setup. If you add it incorrectly — wrong field name, broken TOML syntax, incompatible version constraint — the install step fails.

Don't Do This: Do not add a package to `pyproject.toml` by guessing at the format. If you need to add a dependency, ask the AI

to do it. It knows the correct section (`[project.dependencies]`), the correct syntax, and whether the package name you have in mind is what's actually published on PyPI.

Don't Do This: Do not edit `requirements.txt` to add a package you installed manually with `pip install` without also updating `pyproject.toml`. The two files should describe the same set of dependencies. A package in one but not the other creates drift — the project behaves differently depending on which file the installer reads.

The other thing you own is context. You know, now, that this project has both files for historical reasons. If a teammate tells you to run `pip install -r requirements.txt` and you do that from outside the virtual environment, you'll install packages into the wrong room. The mental model from Chapter 6 applies here: the right packing list, installed into the right room.

With the backend dependencies installed and the right Python running, the frontend is next — and it has its own runtime, its own version problem, and its own package list.

Chapter 9: What Node Is (and Why It's on Your Machine)

The backend is running. The Python environment is set up, `backend/.venv` exists, and the AI was able to start the Flask server without complaint. Now you want to run the frontend.

You ask the AI to start it. It tries to check the environment first, and this comes back:

```
zsh: command not found: node
```

The AI was looking for Node before it could do anything with the frontend. The shell doesn't know what Node is. Or more precisely: the shell can't find a program called `node` anywhere it knows to look.

This is not a code problem. It's a runtime problem. And understanding what Node is — and why it needs to exist on your machine — is what makes the rest of Part 3 make sense.

What Node Actually Is

Node is a JavaScript runtime. That's the precise description, and it's worth knowing what it means.

JavaScript was originally designed to run inside web browsers. A browser can open a `.js` file and execute it — that's been true since the late 1990s. But JavaScript couldn't run on its own, outside a browser, on your machine. You couldn't run a JavaScript file from the command line the same way you can run `python script.py`.

Node changed that. Released in 2009, it took the JavaScript engine from Google's Chrome browser and made it available as a standalone program. Now JavaScript can run on your machine, directly, without a browser in the middle. That's what a runtime is: the program that executes code. Python is a runtime for Python code. Node is a runtime for JavaScript code.

The `neighborhood-meals` frontend is a JavaScript application. It uses a framework called React, written in JavaScript and JSX (JavaScript with HTML-like syntax mixed in). Running that app — even just for local development — requires a JavaScript runtime. That runtime is Node.

When you ask the AI to start the frontend dev server, it needs Node to exist on your machine. When the shell says `command not found: node`, it means that requirement isn't met.

Why Node Lives on Developer Machines

You might expect that a frontend app runs in the browser, so why would you need Node on your development machine at all? The browser handles JavaScript — shouldn't that be enough?

It's enough for running the finished app. It's not enough for building and developing it.

Modern frontend development involves a build step. Your source files — the `.jsx` files in `frontend/src/` — get processed, bundled, and transformed before the browser sees them. Tools do that processing. Those tools are JavaScript programs. They need Node to run.

The development server — the program that lets you open `localhost:3000` in your browser and see the app — is also a JavaScript program. Same situation.

This is why Node ends up on machines that have nothing to do with writing JavaScript. A Python developer building a full-stack app still needs Node for the frontend build pipeline. A data scientist publishing a documentation site may need it for the static site generator. The JavaScript tooling ecosystem runs on Node, and that ecosystem has become part of how a lot of development infrastructure works.

For you, working on `neighborhood-meals`, the requirement is direct: the frontend tooling runs on Node, so Node has to be there.

What "Command Not Found" Actually Means

When the shell says `command not found: node`, it's being specific about the problem. It's not saying Node is broken. It's saying the shell can't locate it.

The shell finds programs by looking through a list of directories called the PATH. When you type `node`, the shell checks each directory in that list for a file named `node`. If it finds one, it runs it. If it doesn't find one in any of the listed directories, you get `command not found`.

This means there are two possible situations:

Node isn't installed. There's no `node` binary anywhere on the machine. This is the clean case — the AI knows exactly what to do.

Node is installed, but the shell can't see it. Node exists in a directory that isn't in the PATH. This happens more than you'd expect. Some Node installation methods put the binary in a non-standard location, or the PATH entry that points to it hasn't been added to the shell configuration. The AI checks both possibilities.

Either way, the AI's starting point is the same: verify whether Node is present at all.

What the AI Does:
The AI runs `node --version` to check whether Node is available and, if so, which version is active.
``
node --version
``
If Node is missing entirely, the response is:
``
zsh: command not found: node
``
That confirms the situation. The AI will proceed to install Node using nvm (Node Version Manager).

Node and npm Are Different Things

Before going further, this distinction matters: Node and npm are not the same program, even though they're installed together.

Node is the JavaScript runtime. It executes JavaScript code.

npm is the package manager for JavaScript. It handles downloading and managing libraries — the same role that pip plays for Python. npm

stands for Node Package Manager, and it ships bundled with Node. When you install Node, you get npm too.

You'll see both names in the AI's output. You'll see `node --version` checks and `npm install` commands in the same session. They're related tools, but they do different things. Node runs code. npm manages the libraries that code depends on.

The analogy to Python is close: Node is to JavaScript as Python is to Python code. npm is to JavaScript libraries as pip is to Python packages. The tools are parallel in concept, different in detail.

How the AI Installs Node

When Node is missing, the AI doesn't install it the way you might install most software — by downloading an installer, running it, and hoping for the best. It uses nvm.

nvm stands for Node Version Manager. Chapter 10 covers it in depth. For now, what matters is that nvm is the standard tool for managing Node installations, and using it rather than a direct installer is intentional. Projects often require specific Node versions. nvm makes it possible to install multiple versions and switch between them. If the AI installs Node directly without nvm, version management becomes harder later.

What the AI Does:
If nvm is already installed on the machine, the AI checks the project's Node version requirement and installs the right version:
``
nvm install 20.11.1
nvm use 20.11.1
``
You'll see output like:
``

Downloading and installing node v20.11.1...
Now using node v20.11.1 (npm v10.2.4)
``

If nvm itself is not installed, the AI will install it first before installing Node. That's a longer process, but the output is still readable: you'll see nvm's install script complete, then the Node install.

Watch For:

After the install, `node --version` should return `v20.11.1`. That confirms the runtime is present and the PATH is configured correctly. If you see a version number, Node is ready.

Why This Is the Frontend's Version of the Python Problem

You saw something similar in Part 2. The Python error — `ModuleNotFoundError: No module named 'flask'` — wasn't about Python being missing. It was about the right environment not being set up.

The Node situation has the same shape. The shell returned `command not found: node` not because JavaScript doesn't work, but because the runtime that JavaScript needs for local development wasn't in place.

The fix is the same kind of fix: establish the runtime, then proceed. In Python, that meant creating a virtual environment with the right interpreter. In Node, it means getting the right version of Node active on the machine.

What's slightly different is the version layer. With Python, the machine often had some Python installed — just the wrong version. With Node, the machine might have nothing at all. Both cases look like "command not found" from the outside. The AI distinguishes between them by checking what's installed before deciding what to do.

What You Own

The AI handles the install. You own knowing what you're looking at when the error happens again.

If you open a new terminal and try to do something with the frontend and see `command not found: node`, your first question should not be "did I break something?" It should be: "is Node active in this terminal?" There are reasons it might not be, and Chapter 10 explains the main one.

> **Don't Do This:** Do not install Node by downloading the package from nodejs.org and running it directly, even if it looks like the easiest path. That method can work, but it puts Node in a location that conflicts with nvm's version management. Once the AI switches to nvm-managed Node, you can end up with two Node installations that fight for the PATH. Let the AI choose the installation method.

> **Key Takeaway:** `zsh: command not found: node` is a runtime missing problem, not a code problem. Node is the JavaScript runtime the frontend tooling needs. The AI installs it via nvm. Once `node --version` returns a version number, that wall is down.

Node exists on the machine now — but the version the project requires might not be the version currently active, and that's what the next chapter is about.

Chapter 10: nvm and Node Versions

Node is on the machine. The AI verified that with `node --version`. So it proceeds to the next step: installing the frontend's dependencies. It runs:

```
npm install
```

And gets this:

```
npm ERR! code EBADENGINE
npm ERR! engine Unsupported engine
npm ERR! engine Not compatible with your version of node/npm: neighborh
npm ERR! notsup Not compatible with your version of node/npm: neighborh
npm ERR! notsup Required: {"node":">=20.11.0"}
npm ERR! notsup Actual:   {"npm":"10.2.4","node":"18.19.0"}
```

Node is installed. npm is installed. The install command still failed.

The error is precise: the project requires Node `>=20.11.0`. The machine has Node `18.19.0`. Those two things don't satisfy each other, and npm stopped before it could install anything.

This is not a broken installation. It's a version mismatch. And it leads directly to the concept that makes Node version management make sense.

Why Node Projects Pin to Versions

With Python, you might have one Python on your machine for years and it mostly works across projects. Node has more of a versioning culture. Projects tend to be specific about which Node version they expect, and for good reason.

Node releases follow a predictable cycle. Even-numbered versions (16, 18, 20) become Long-Term Support (LTS) releases — the ones that get security patches and bug fixes for several years. New JavaScript language features, new built-in APIs, and new npm behaviors land in different major versions. A project that was built and tested against Node 20 may use features or behaviors that don't exist in Node 18.

The `neighborhood-meals` frontend requires `>=20.11.0`. That's not arbitrary. The project's dependencies, build tools, and possibly some code itself depend on things that exist in Node 20 and not before. The npm engine check is enforcing that requirement before it installs anything that might fail at runtime.

You're likely to encounter this pattern regularly. A project you clone might have been written six months ago targeting Node 20. Another

project might still be on Node 18. A tool you pull in for your own work might require Node 22. This is normal. The machine handles it by having multiple Node versions available at once.

What nvm Is

nvm is the program that makes multiple Node versions manageable. The name stands for Node Version Manager.

Think of it as a switchboard. Without nvm, your machine has one Node installation, and every project uses that same version. With nvm, your machine has as many Node versions as you've installed, and you switch between them depending on which project you're working on.

When you switch Node versions with nvm, the PATH is updated so that `node` and `npm` commands point to the version you selected. The other installed versions are still there — they're just not currently active. A session using Node 18 doesn't interfere with a session using Node 20. They coexist on disk, and nvm routes commands to the right one.

The AI uses nvm to:

• Check which Node version is currently active

• Check which version the project needs

• Switch to the right version, or install it if it isn't on the machine yet

• Proceed with `npm install`

Without nvm, the AI's options are more limited. It can't easily switch Node versions in a single shell session. nvm is why that's possible.

What `.nvmrc` Is

Open `frontend/` and there's likely a file called `.nvmrc`. It's short and contains a single line:

```
20.11.1
```

That's it. That file is the project's declaration of which Node version it expects. When the AI runs `nvm use` without specifying a version, nvm reads `.nvmrc` and switches to whatever version is listed there.

You don't need to create or maintain `.nvmrc`. It's already in the project. The AI reads it. The point of knowing it exists is so that when you see the AI run `nvm use` with no version argument, you understand what it's reading — and why the version it switches to is the right one.

If `.nvmrc` isn't present, nvm won't know which version to switch to automatically. The AI will handle that case by reading the `engines` field in `package.json` instead. Both paths lead to the same place: the correct Node version.

What the AI Does

When the AI sees the `EBADENGINE` error, the sequence is straightforward.

What the AI Does:
First, the AI checks for `.nvmrc` in the frontend directory:
`` cat frontend/.nvmrc ```
Output:
`` 20.11.1 ```
Then it checks if that version is installed:
`` nvm ls 20.11.1 ```

If the version isn't installed yet:

```
nvm install 20.11.1
```

You'll see:

```
Downloading and installing node v20.11.1...
Downloading https://nodejs.org/dist/v20.11.1/node-v20.11.1-darwin-arm64.tar.xz...
Computing checksum with sha256sum
Now using node v20.11.1 (npm v10.2.4)
```

Then:

```
nvm use 20.11.1
```

And it retries:

```
npm install
```

Watch For:

After the switch, `node --version` should return `v20.11.1`. When `npm install` runs again, the `EBADENGINE` error should be gone. You'll see the install proceed — downloading packages, logging progress, finishing with `added N packages in Xs`. That's the engine check passing and the install completing.

The Session Caveat

Here's the part that catches people off guard: `nvm use` only applies to the current terminal session.

When you open a new terminal window, the shell starts fresh. It reads your shell configuration files (`.zshrc` for zsh, `.bashrc` for bash), and unless nvm has been configured to read `.nvmrc` automatically on directory change, the active Node version resets to nvm's default — or to whatever was last set as the default.

This means: if you were on Node 20.11.1, close the terminal, open a new one, and try to run the frontend, you might be back on Node 18. The commands fail. The project looks broken. It's not broken — the version just reset.

There are two ways this gets handled. One is that the AI switches Node at the start of each session before doing anything with the frontend. The other is that nvm can be configured to automatically run `nvm use` when you enter a directory that has a `.nvmrc` file. That's a persistent configuration — once it's set, nvm handles the switch without the AI needing to do it explicitly each time.

What the AI Does:
If nvm's auto-switch isn't configured, the AI will add a hook to your shell's configuration file (`.zshrc`) that enables automatic version switching. The hook watches for `.nvmrc` files when you change directories and runs `nvm use` for you.
You'll see the AI add something like this to `~/.zshrc`:
``
autoload -U add-zsh-hook
load-nvmrc() {
local nvmrc_path
nvmrc_path="$(nvm_find_nvmrc)"
if [-n "$nvmrc_path"]; then

```
    nvm use
  fi
}
add-zsh-hook chpwd load-nvmrc
load-nvmrc
```

After this, opening a new terminal and navigating to `frontend/` will automatically switch to Node 20.11.1.

The Python Parallel

This is the Node version of a problem you've already seen.

In Chapter 6, the Python error wasn't about Python being absent — it was about the wrong Python being active, or the right packages not being installed into the right environment. The fix involved setting up the right environment and making sure Python commands ran inside it.

Here, the Node error isn't about Node being absent. It's about the wrong Node version being active. The fix involves switching to the right version, verifying it, and then proceeding.

The parallel even holds in the session behavior. With Python, activating a virtual environment only applies to the current session — close the terminal and the environment is no longer active. With Node, `nvm use` only applies to the current session unless automatic switching is configured.

Both tools solve the same underlying problem: a machine can have multiple versions of a runtime, and different projects need different versions. The mechanism is different (`.venv` vs. nvm), but the concept is the same.

What You Own

The AI manages the nvm install and the version switching. You own one thing: noticing when the version has reset.

If you open a new terminal, navigate to the frontend, and something starts failing that was working before, version reset is one of the first things to check. The AI can confirm by running `node --version`. If it shows `v18.19.0` when the project needs `v20.11.1`, that's the explanation.

You don't need to fix it yourself. You can ask the AI to switch the version. But knowing that this can happen — and recognizing it when it does — is what keeps it from being a mystery.

> **Don't Do This:** Do not use a Node installer from nodejs.org or a system package manager (like Homebrew's `brew install node`) to install or upgrade Node if nvm is already managing Node on your machine. Mixing install methods creates conflicts. The system-installed Node and the nvm-managed Node may fight over the PATH, and the AI's `nvm use` commands may not have the effect you expect. If nvm is in place, it should be the only thing managing Node versions.

> **Key Takeaway:** `EBADENGINE` means the Node version on the machine doesn't satisfy what the project declared. nvm is the tool that fixes this — it switches the active Node version to the one the project needs. The fix is a version switch, not a reinstall.

The right version of Node is active — but `npm install` also revealed that `frontend/node_modules` was empty, and that's what the next chapter explains.

Chapter 11: package.json and node_modules

Node 20.11.1 is active. The AI has confirmed it with `node --version`. Now it tries to start the frontend development server by running the project's dev script:

```
npm run dev
```

This comes back immediately:

```
> neighborhood-meals-frontend@0.1.0 dev
> vite

sh: vite: command not found
```

`vite` is listed in `frontend/package.json`. The project knows it needs vite. It says so explicitly. So why can't the shell find it?

Because listing something in `package.json` doesn't install it. The manifest and the installed packages are two different things. `frontend/node_modules/` — the directory where installed packages live — doesn't exist yet.

The Manifest and the Shelf

`package.json` is the frontend's manifest. It declares what the project is, what scripts it can run, and what packages it depends on. Think of it as the packing list from Chapter 8 — the document that describes what belongs in the project's environment.

But a packing list doesn't pack itself. Reading it doesn't fill the box.

`node_modules/` is the shelf. It's the local directory where the actual package files live — the downloaded code for every dependency the project declared. When vite is in `node_modules/`, the shell can find it. When `node_modules/` doesn't exist, the shell can't find anything, regardless of what `package.json` says.

That's what happened here. `package.json` correctly declares vite as a dependency. But `npm install` — the step that reads the manifest and downloads everything into `node_modules/` — hasn't run yet. The shelf is empty. The error is accurate.

What package.json Contains

Open `frontend/package.json` and you'll see something like this:

```
{
  "name": "neighborhood-meals-frontend",
  "version": "0.1.0",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0"
  },
  "devDependencies": {
    "vite": "^5.0.0",
    "@vitejs/plugin-react": "^4.2.0"
  },
  "engines": {
    "node": ">=20.11.0"
  }
}
```

Each section does something specific.

`name` **and** `version` identify this package. These fields matter if the package were published to the npm registry, but for a private frontend app, they're mostly just labels.

`scripts` is where the AI finds the commands it runs. When you ask the AI to start the dev server, it runs `npm run dev`. npm looks up `dev` in this section and runs `vite`. That's the whole mechanism — `npm run` is just a shorthand that looks up the command in `scripts` and executes it. The `sh: vite: command not found` error happened because npm found `dev: "vite"` in `scripts`, tried to run `vite`, and the shell couldn't locate the binary.

`dependencies` is the list of packages the app needs to run. React and React DOM live here. These packages are required for the built app to work in a browser.

`devDependencies` is the list of packages needed for development but not for the built app itself. Vite is here because it's a development server and build tool — users of the final app never encounter it.

Separating dev and runtime dependencies keeps the production build smaller.

`engines` declares what runtime versions this project supports. This is where `"node": ">=20.11.0"` lives — the same constraint that caused the `EBADENGINE` error in Chapter 10. npm reads this field and enforces it during install.

Why node_modules Isn't in Git

If you look at `frontend/.gitignore`, you'll find `node_modules` listed there. It's intentionally excluded from version control.

The reason is size and redundancy. `node_modules/` for a typical frontend project can contain thousands of files and tens of thousands of subdirectories. Committing all of that would bloat the repository enormously, slow down every clone and pull, and create noise in every diff.

The other reason is that the directory is fully reproducible. Everything in `node_modules/` can be recreated from `package.json` (and from `package-lock.json` for exact versions). If you clone the repository and run `npm install`, you get the same `node_modules/` that any other developer gets. It's generated output, not source.

This is the same logic behind excluding `backend/.venv` from Git. Virtual environments and `node_modules/` are both local, generated, and reconstructible. They belong on disk but not in the repository.

What package-lock.json Is

When `npm install` runs, it creates or updates a file called `package-lock.json`. This file is in the repository — unlike `node_modules/`, it's checked in.

Here's the distinction between the two files:

`package.json` uses version ranges. `"vite": "^5.0.0"` means: any version of vite from 5.0.0 up to (but not including) 6.0.0 is acceptable. On the day `npm install` runs, npm picks the latest version that satisfies that range.

`package-lock.json` records the exact version npm actually chose. If npm picked vite 5.1.4 today, the lockfile says `"version": "5.1.4"`. Tomorrow, if vite 5.2.0 is released, the lockfile still says `5.1.4`. When the AI installs from a lockfile, it installs that exact version — not whatever is latest.

This is reproducibility. Every developer cloning the project and running `npm install` gets the same versions. The lockfile is what makes that possible.

When the lockfile exists, the AI uses `npm ci` (clean install) rather than `npm install` in some contexts. `npm ci` reads the lockfile and installs exactly what it records, without resolving ranges. For setting up an existing project, that's more reliable.

What npm install Does

`npm install` is the step that connects the manifest to the shelf.

What the AI Does:
With Node 20.11.1 active, the AI runs:
``
cd frontend
npm install
``
npm reads `package.json` and `package-lock.json`, then downloads every declared package into `frontend/node_modules/`. You'll see output like:
``
added 312 packages, and audited 313 packages in 14s

102 packages are looking for funding
run `npm fund` for details
found 0 vulnerabilities
``
`added 312 packages` is the important line. That's `node_modules/` being created and filled.
Watch For:
After the install, `frontend/node_modules/` exists as a directory. The AI can confirm this by checking for vite specifically:
``
ls frontend/node_modules/.bin/vite
``
If that path exists, the binary is there. `npm run dev` will work.

After the Install

With `node_modules/` populated, the dev script resolves.

```
npm run dev
```

Now produces:

```
> neighborhood-meals-frontend@0.1.0 dev
> vite

  VITE v5.1.4  ready in 312 ms

  ➜  Local:   http://localhost:3000/
  ➜  Network: use --host to expose
```

The shell found `vite` because it's now in `frontend/node_modules/.bin/`. npm knows to look there when running scripts. The dev server started. In this project, the frontend is configured to serve on `localhost:3000`, so that's where the app appears.

This is the same shape as the Python resolution in Chapter 6. Once the packages were installed into the right place, the import error went away. Here, once `node_modules/` was created with the right contents, the command not found error went away. The pattern is the same: declare dependencies in the manifest, install them locally, use them.

What You Own

The AI manages the install. You own not undoing it — and knowing what to do if it gets undone.

If you delete `node_modules/` accidentally, the fix is `npm install` from the `frontend/` directory. The AI can run that. The directory will be recreated from the lockfile. The install takes a minute or two.

If `node_modules/` becomes corrupted or mismatched — sometimes this happens after a Node version switch or a partial install — the fix is to delete the directory and reinstall clean. The AI handles this with `rm -rf node_modules && npm install`. This is one of the few cases where deleting the directory is intentional.

> **Don't Do This:** Do not manually edit the version numbers in `package.json` to try to resolve a conflict or update a package. Version constraints interact with each other — bumping vite's version might break a plugin that depends on the previous version. If a package needs to be updated, ask the AI to do it. It knows how to check compatibility and update the lockfile correctly.

> **Don't Do This:** Do not run `npm install <package-name>` to add a package without knowing what it adds. Running `npm install react-router-dom`, for example, adds an entry to `package.json` and updates `package-lock.json`. That's a project-level change that should go through the AI, not an ad hoc command. The AI will install what's needed when it's needed, and it will add it to the correct section (`dependencies` vs `devDependencies`).

The `package.json` is the source of truth for what the frontend needs. If that file is accurate, the AI can set up the environment on any machine. If it drifts — packages installed manually, versions edited by hand, entries deleted without reinstalling — the project becomes harder to reproduce.

> **Key Takeaway:** `package.json` declares what the frontend needs. `node_modules/` is the local copy of those packages. The gap between them is closed by `npm install`. Until that step runs, nothing declared in `package.json` exists locally.

The frontend is running — but some problems can't be fixed by installing the right packages or switching to the right version, because some problems live at the machine level, and that's where the next part begins.

Chapter 12: The Gap Warp Fills

You've made it past the biggest walls. Git is tracking your work. The Python virtual environment is active. Node 20.11.1 is running. Both the backend and the frontend can start when everything is clean.

And then something goes wrong that has nothing to do with any of that.

You try to start the frontend dev server and get this:

```
Error: listen EADDRINUSE: address already in use :::3000
```

The code hasn't changed. The packages are all there. The error isn't in a file — you couldn't edit your way out of this even if you wanted to. So you paste the error into your AI coding tool. It explains that port 3000 is already occupied and suggests you kill whatever is using it. It might even suggest a command.

But here's the gap: the AI coding tool can't actually look at your machine right now. It can't run `lsof` in your terminal. It can't see which process is on port 3000 or what its PID is. It can tell you what *kind* of

problem this is, and it can suggest what the fix probably looks like — but it can't do the diagnosis itself.

This isn't a bug in your AI coding tool. It's the edge of what that tool was designed for.

Why the AI Coding Tool Has This Gap

Your AI coding tool — whether that's Claude, Cursor, Copilot, or something else — is a code tool. It reads files. It writes files. It understands what's in `backend/app.py` and `frontend/src/App.jsx`. It can trace the logic of a function, find a bug, refactor a module, generate a test.

What it doesn't have access to is your machine's running state: the processes currently executing, the ports they're listening on, the PATH your shell is searching when you type a command, the environment variables that are (or aren't) set in your current terminal session.

Those things live outside the files. They're not in the project folder. They exist in the operating system itself — a list of running processes, a shell configuration file, a dynamic lookup table the shell consults every time you run a command. The AI coding tool works in the file layer. Machine state is a different layer entirely.

This is why the AI can tell you "you probably need to kill whatever is on port 3000" but can't actually do it. The command that would do it — and the confirmation that it worked — has to happen in a terminal, on your machine, interacting with the OS directly.

That's the gap. And Warp fills it.

The Three Walls You'll Hit

The problems in this layer have a recognizable shape. You've already seen the port conflict. Here are the other two.

The PATH Wall

The AI just helped you install `gh`, the GitHub CLI. It ran an install command, it completed without errors. Then you try to use it:

```
zsh: command not found: gh
```

The binary is on your disk. The installation succeeded. But when the shell tries to run `gh`, it searches a list of directories called the PATH and can't find the program there.

PATH is the shell's search list. When you type a command, the shell doesn't look everywhere on your machine — that would be slow. It looks in a specific list of directories, in order, and runs the first match it finds. If `gh` was installed in a directory that isn't on that list, the shell will never find it.

The AI coding tool can tell you what PATH is and explain how it works. But it can't look at your current PATH, see which directory is missing, and write the fix to your shell configuration file — not without a live terminal session to interact with.

The Persistence Wall

You've been working with an API key. You set it in the terminal:

```
export OPENAI_API_KEY=abc123
```

Everything works. You close the terminal and open a new one. You try to check the value:

```
echo $OPENAI_API_KEY
```

Nothing. The key is gone.

`export` sets a variable for the current shell session. When the session ends, the variable goes with it. If you want a variable to be there every time you open a terminal, it has to live in a shell configuration file — specifically something like `~/.zshrc` or `~/.bash_profile`. Those files run automatically every time a new terminal session starts.

The AI coding tool can absolutely tell you what to add to `~/.zshrc`. But "telling you" and "doing it" are different things when the file in question is on your machine, not in the project. You need a terminal that can interact with your shell configuration directly.

What These Problems Have in Common

Port conflicts, PATH problems, and environment variable persistence are all the same kind of problem: machine state.

Not code. Not packages. Not dependencies. The actual running state of your operating system — what's executing, what the shell can find, what survives from one session to the next.

When you're debugging a Python import error or a broken React component, the AI coding tool is the right place to look. It has the code in front of it. It can trace the problem.

When you're debugging why port 3000 is occupied, why a command can't be found, or why an environment variable keeps disappearing, the problem isn't in a file. The AI coding tool can advise, but it can't diagnose.

That's the class of problems Warp is built for.

Is Warp Required?

No.

If you're comfortable at the terminal — if you know enough to look up commands, run diagnostics, and modify your shell config without too much friction — you don't need Warp. The class of problems this part of the book covers can be solved with a regular terminal and a search engine.

Warp is included here because it's what the author uses, and genuinely finds valuable. The use case is specific: you're capable enough at the command line to get things done, but you're not a systems administrator. You know your way around Linux or macOS well enough to work in it — you just don't have every `lsof` flag memorized, and you'd rather not open a browser to ask ChatGPT how to do something in the terminal when you're already in the terminal. Warp fills that gap. Ask in plain English, get a command, review it, run it — without breaking your focus or switching contexts.

If that describes you, read on. If you're already comfortable administering your own machine, this part of the book is context, not instruction.

What Warp Is

Warp is a terminal emulator. Like Terminal.app on macOS, or Windows Terminal, or GNOME Terminal on Linux — it's the application where you type commands and see output. You can use it for anything you'd use a regular terminal for.

That's where it started. Warp existed before it had meaningful AI features, and it was excellent — better organized output, better keyboard navigation, better search. Worth using just for that.

The AI features came later, and they changed what Warp is useful for.

The architectural advantage

Here's what makes Warp different from your AI coding tool in a fundamental way: Warp *is* the terminal. Claude Code, Cursor, and Codex run *in* a terminal. Warp is the environment itself.

That distinction matters practically. Your AI coding tool can suggest a command and tell you to run it. Warp generates the command in the same place where it will execute, with full access to the machine's running state: processes, ports, the file system, shell configuration. Anything you can do in a terminal, Warp can do for you — not via a file it edited, but by interacting with the OS directly.

How it works

At the prompt, type #. Describe what you want in plain English. Warp generates the command. Review it, then run it.

You're not memorizing commands. You're describing situations. "What is on port 3000." "This command isn't being found but I just installed it." "Set this environment variable permanently so it doesn't disappear when I close the terminal." Warp translates those descriptions into the specific

shell commands you need.

Warp also changes how terminal output works. Every command you run produces a **block** — the command and its output grouped together, with an indicator showing whether it succeeded or failed. Blocks are self-contained. You can navigate between them, copy just the output you want, and search within a single block's output. In a traditional terminal, output streams past and scrolls away. In Warp, it stays organized.

A note on cost and scope

Warp has grown into a full agentic platform — it now runs multi-step coding tasks and competes with Claude Code and Codex for software development work. You might be tempted to use it for coding.

Be aware of the tradeoff. Your AI coding tool's subscription is subsidized by significant investment capital. Warp's isn't. If you use Warp for coding at any real volume, you'll exhaust your monthly token allotment in roughly three days. It's not designed for that workload at its price point.

What it *is* designed for — and what earns its subscription — is system administration. Port conflicts. PATH problems. Shell configuration. Process management. Things you need occasionally but don't want to memorize. At that pace, the cost is low and the value is high. That's the use case this book covers.

One note on availability: Warp is fully available on macOS. Linux support arrived in February 2024. Windows support arrived in February 2025. Whatever machine you're on, you can use it.

The Division of Responsibility

Here's the model that will carry you through the next four chapters:

Code problems — in files, in logic, in syntax — go to your AI coding tool. It reads and writes files. That's its terrain.

Machine-state problems — processes, ports, PATH, shell configuration, environment variables — go to Warp. It interacts with the OS through a live terminal session. That's its terrain.

These tools don't compete. They're not alternatives to each other. You'll have both open at the same time. When the backend throws a Python error, you switch to the AI coding tool. When the server won't start because a port is occupied, you switch to Warp.

The friction comes when you bring the wrong tool to the problem. When you keep pasting error messages into the AI coding tool hoping it will diagnose something it can't see. When you spend twenty minutes trying to get the AI to fix a PATH issue when Warp would do it in thirty seconds.

Knowing which tool to reach for first is most of the battle.

> **Key Takeaway:** Your AI coding tool works in the file layer. Machine-state problems — ports, PATH, environment variables, shell config — live outside that layer. Warp handles them through a live terminal session with AI built in. Both tools are open at the same time; you're switching between them based on the kind of problem you're looking at.

Now that you know what Warp is for, the next chapter shows you how to use it.

Chapter 13: Warp Basics

You have Warp open. It looks like a terminal. There's a prompt, probably a blinking cursor, and if you type a command and press Enter it runs — just like any other terminal.

The difference isn't in how it looks. It's in two things: how output is organized, and what happens when you type #.

Blocks

In a traditional terminal, output streams past. You run a command, get ten lines of output, run another command, get thirty more lines. Everything accumulates from top to bottom, and if you want to find something from three commands ago, you scroll up and hunt for it in an undifferentiated wall of text.

Warp organizes output differently. Every command you run produces a **block** — the command you typed and the output it produced are grouped together, with a visual indicator showing whether the command succeeded or failed. Green indicator: it exited cleanly. Red indicator: something went wrong.

Blocks are self-contained. You can navigate between them with keyboard shortcuts instead of scrolling. You can copy just the output of a specific command. You can search within a single block's output using Cmd+F (macOS) or Ctrl+F (Linux/Windows). You can collapse blocks you're done with.

When you're diagnosing a problem, this matters. You're going to run a diagnostic command, read the output, run a fix command, and check the result. In a traditional terminal, those four outputs are interleaved. In Warp, each one is its own unit. You can see exactly what each command produced.

— AI Command Search

This is the feature that changes how you use the terminal.

At the prompt, type #. Don't press Enter yet — just #. The prompt will shift into a description field.

Now type what you want in plain English. Not a command — a description of what you're trying to do, or a problem you're looking at. Warp processes your description and generates the command. You review it. Then you run it.

A few things this means:

You are not memorizing syntax. The syntax for checking which process is using a port is `lsof -i :3000`. You don't need to know that. You describe the situation, and Warp produces the command. The next time you need it, you describe it again.

You review before you run. Warp generates the command and shows it to you. You can read it, understand what it does, and decide whether to execute it. This is not a "just trust the AI" workflow. You own the final decision on every command, especially commands that modify system files.

Specificity matters. "Something is broken" is too vague for Warp to do much with. "Port 3000 is already in use and I need to find what's using it" is specific enough. The more accurately you describe the situation, the more useful the generated command will be. You'll develop a feel for this after a few uses.

Pattern 2: Port Conflict

Here's how this plays out in practice, starting from the wall.

You tried to start the frontend dev server and got:

```
Error: listen EADDRINUSE: address already in use :::3000
```

Open Warp. At the prompt, type # and then describe what you need:

```
# what is on port 3000
```

Warp generates:

```
lsof -i :3000
```

`lsof` lists open files and network connections. The `-i :3000` flag filters to connections on port 3000. You don't need to remember the flag. You just need to know that this command will tell you what's there.

Run it. The output looks something like this:

```
COMMAND   PID   USER   FD   TYPE             DEVICE SIZE/OFF NODE NAME
node    12345   you   24u  IPv6 0x...         0t0  TCP *:3000 (LISTEN)
```

There it is. A Node process, PID 12345, listening on port 3000. This is the previous dev server session — it didn't stop cleanly when you closed

that terminal window. The process is still running. The port is still occupied.

Now type # again:

```
# kill that process
```

Or if you want to be more specific:

```
# kill PID 12345
```

Warp generates:

```
kill -9 12345
```

`kill` sends a signal to a process. `-9` is the SIGKILL signal — it terminates the process immediately, no cleanup. The `12345` is the PID from the output you just read.

> **Watch For:** After running `kill -9 12345`, run `lsof -i :3000` again. If the output is empty, the port is clear. If the same process is still there, you may need to run the kill command again.

Now start your dev server. The `EADDRINUSE` error is gone.

> **Don't Do This:** Don't run `kill -9` without checking the PID first. The number in your output will be different from the one in this example — it changes every time a process starts. Read the `lsof` output, find the PID column, and use that specific number. Running `kill -9` on the wrong PID terminates a different process.

Pattern 3: PATH Problem

The other wall from Chapter 12. You (or the AI) just installed `gh` and now:

```
zsh: command not found: gh
```

Open Warp. Type:

```
# gh command not found but I just installed it
```

Warp will suggest checking where the binary is and whether that location is on your PATH:

```
which gh
```

`which` searches your PATH and prints the location of the command if it finds it. If `gh` isn't on your PATH at all, `which gh` returns nothing — or a message like `gh not found`. If it finds it, it returns the path, like `/usr/local/bin/gh`.

Run `which gh`. Two possibilities:

It returns a path: The binary exists and the shell can find it — which means something else is going on. You might be running the wrong shell, or there might be a version conflict. Describe what you see to Warp and continue diagnosing.

It returns nothing: The binary is installed somewhere that isn't on your PATH. Now you need to figure out where it actually is and add that location to PATH.

Type # again:

```
# add /usr/local/bin to my PATH permanently
```

Warp generates:

```
echo 'export PATH="/usr/local/bin:$PATH"' >> ~/.zshrc && source ~/.zshr
```

This is worth reading before you run it.

`echo 'export PATH="/usr/local/bin:$PATH"'` produces a line of text. `>> ~/.zshrc` appends that line to the end of your `~/.zshrc` file — your shell configuration file, which runs every time a new terminal session starts. `&&` means "if the first command succeeded, run the next one." `source ~/.zshrc` reloads the configuration file in the current session, so the change takes effect immediately without you needing to close and reopen the terminal.

> **Don't Do This:** Do not use `>` instead of `>>`. A single `>` overwrites the file. A double `>>` appends to it. If you accidentally overwrite your `~/.zshrc`, you'll wipe out all your existing shell configuration. Warp generates the correct `>>` — this is just something to understand when you read the command before running it.

After running it, confirm:

```
gh --version
```

If it prints a version number, the PATH fix worked. If it still says `command not found`, the binary isn't in `/usr/local/bin` — it's somewhere else. Type # again and describe what you're seeing; Warp will help narrow it down.

> **Watch For:** `gh --version` should return something like `gh version 2.x.x`. If you see that, you're done.

Describing Problems Well

The quality of what Warp generates depends on how specifically you describe what you're looking at. Some examples of the difference:

The pattern: name the error or symptom you actually see, not just the outcome you want. Warp needs to understand what the current state is before it can suggest a path forward.

You'll get faster at this. After the first few times, you start to know instinctively how much detail Warp needs.

What You Own

Warp generates the command. You run it. That division is straightforward for read-only commands like `lsof` — there's no risk in checking what's on a port. It matters more for commands that change things.

Two categories to be careful about:

Commands that modify shell configuration files. When Warp generates a command that writes to `~/.zshrc`, read it before running it. Check whether it's appending (`>>`) or overwriting (`>`). Check that the value being written is what you intend. These changes persist across every future terminal session.

Commands that kill processes. Confirm the PID is what you expect before running `kill -9`. The `lsof` output tells you the command name (`node`, `python`, etc.) — that's a sanity check. If the PID in your output belongs to something other than a dev server, stop and look more carefully.

Warp is a tool for working faster, not for running commands without reading them. The AI generates; you decide.

These two patterns — killing a stuck process and fixing a broken PATH — cover the most common machine-state problems you'll hit while running `neighborhood-meals`, but they're just the beginning of what Warp can do when you have a few more worked examples.

Chapter 14: Warp Workflows for Developers

You've used # to kill a stuck process and to fix a broken PATH. The pattern is familiar now: describe the problem, get the command, review it, run it.

This chapter adds two more patterns and introduces something that happens naturally once you're comfortable with those patterns.

Pattern 4: Environment Variables That Disappear

You're connecting the `neighborhood-meals` backend to an external API. The AI coding tool told you to set an API key as an environment variable, so you did:

```
export OPENAI_API_KEY=abc123
```

The backend can read it. Everything works. You stop for the day, close the terminal, and come back the next morning.

New terminal. You check the value:

```
echo $OPENAI_API_KEY
```

Nothing. The variable is gone.

This is the persistence wall from Chapter 12 in practice. `export` sets a variable for the current shell session. When the session ends — when you close the terminal — the variable disappears. The OS doesn't write it anywhere permanent. The next terminal session starts fresh.

If you want an environment variable to be there every time you open a terminal, it has to live in a shell configuration file. On macOS and Linux with zsh (the default shell on modern macOS), that file is `~/.zshrc`. Every time a new terminal session starts, your shell reads `~/.zshrc` and runs everything in it. If your `export` statement is in there, the variable gets set automatically on every startup.

Open Warp and describe what you need:

```
# set OPENAI_API_KEY permanently so it survives terminal restarts
```

Warp generates:

```
echo 'export OPENAI_API_KEY=abc123' >> ~/.zshrc && source ~/.zshrc
```

The logic is the same as the PATH fix from Chapter 13: append the export statement to `~/.zshrc`, then reload the file in the current session so the change takes effect immediately.

> **Don't Do This:** Do not put the actual value of a sensitive API key in a public repository. The command above writes the key to `~/.zshrc`, which is a local file on your machine and is not tracked by git (as long as you haven't explicitly added it to the project). But if you ever commit `~/.zshrc` itself, or copy its contents somewhere version-controlled, the key is exposed. Keep `~/.zshrc` off of git.

Run the command. To confirm it worked, open a new terminal — not just a new tab, a genuinely new session — and check:

```
echo $OPENAI_API_KEY
```

> **Watch For:** The value you set should print to the terminal. If it returns an empty line, the command didn't write to `~/.zshrc`

correctly — check whether you're running zsh (the default shell on modern macOS) or bash. If you're running bash, the file you want is `~/.bash_profile` or `~/.bashrc`, not `~/.zshrc`. Describe this to Warp: `# I'm using bash and I need to set an environment variable permanently` — Warp will generate the right command for your shell.

Pattern 5: Check Before You Start

The port conflict in Chapter 13 happened after the fact: you tried to start the server, it failed, you diagnosed. There's a faster approach.

Before starting the dev server, check whether the port is already in use:

```
# is anything on port 3000 right now
```

Warp generates:

```
lsof -i :3000
```

You've seen this command before. You don't need to memorize what `lsof` or `-i` mean — but you're starting to recognize it. Run it.

If the output is empty: Nothing is on port 3000. Safe to start the server.

If you see a process: Something is already there. You know what to do: `# kill that process`, get the PID from the output, run `kill -9 <PID>`. Then check again before starting.

The lesson here is about sequence, not commands. Starting a dev server without checking first means you might get an `EADDRINUSE` error, go through the diagnosis, kill the process, and then start the server. That's three steps. Checking first collapses it to one.

There's a deeper pattern here too: Warp is often most useful before the failure, not just after it. Once you know the common walls on your machine, you can ask preventative questions. Is the port free? Is the binary on PATH? Is the variable still set? These are boring checks, and boring checks are good. They turn "debugging" into "verifying the machine state before you depend on it."

Why `lsof` Keeps Appearing

You've now seen `lsof -i :3000` come up twice — once when diagnosing a port conflict, and once as a pre-check. This is not a coincidence, and it's not asking you to memorize anything.

What's happening is recognition. You described two different situations to Warp ("what is on port 3000" and "is anything on port 3000 right now"), and both times it gave you the same command. That's because `lsof -i :3000` is the right tool for both questions — it checks which processes have a network connection on that port.

After seeing it twice, you know what it does. You don't need the flags memorized. You don't need to know how to spell it. But the next time you see `lsof` in Warp output, you'll recognize it as "the command that tells me what's using a port." That recognition is the point. It's the difference between output that means something and output that's just noise.

The same thing will happen with other commands you encounter repeatedly. Each time Warp generates the same tool for similar situations, the tool becomes slightly more legible.

Warp Drive

Once you've been using # for a while, you'll start to notice a pattern: some descriptions come up over and over. "What is on port 3000." "What is on port 8000." "Is anything on port 3000 right now." You're asking the same question in slightly different forms.

Warp Drive is a library of saved, shareable commands. A **workflow** is a command with named blanks — like a template. You might save a workflow called "Check port" that runs `lsof -i :{PORT}`, where `PORT` is a placeholder. When you use the workflow, Warp asks you for the port number and fills it in.

For a solo developer, Warp Drive is a personal cookbook. You save the patterns you use often so you don't have to retype the description from

scratch each time.

For a team, it's a shared resource. If everyone on the team saves the same diagnostic workflows, you're all using the same commands when you're debugging the same server. The senior developer who knows what `lsof` is can save it once, and everyone else can use it without needing to know the syntax.

That matters more than it might seem. A shared workflow doesn't just save time. It standardizes what "check the port" means on your team. Instead of one person using `lsof`, another using `netstat`, and a third searching the web every time, everyone reaches for the same known-good diagnostic. For a new developer, that consistency removes a lot of ambient doubt.

You don't need Warp Drive to use the patterns in this book. The # workflow gets you there every time. But if you find yourself describing the same situation repeatedly, Warp Drive is where you turn that repetition into a saved shortcut.

To get to it: look for the Warp Drive icon in the Warp sidebar, or search for it in Warp's command palette. You can browse workflows others have published, save your own, and share them with a team.

All Five Patterns Together

At this point you have five diagnostic patterns:

None of these require memorizing commands. Each one starts with describing a situation. Warp handles the syntax.

What you're building is not a list of commands — it's a habit of reaching for the right tool when you see a particular shape of problem. Port occupied: Warp. Command not found: Warp. Variable disappeared overnight: Warp. Syntax error in `backend/app.py`: AI coding tool.

And by this point, another habit has started to form too: you review the generated command before you run it. That's important. Warp is helping with translation, not replacing judgment. You still glance at the

command, make sure it matches what you asked for, then run it. That's the same pattern you've been building with the AI coding tool throughout the book: let the tool do the syntax, keep ownership of the intent.

> **Key Takeaway:** The # pattern in Warp handles machine-state problems the same way every time: describe what you see, review the generated command, run it, confirm it worked. Warp Drive extends this with saved, shareable workflows — useful once you're running the same patterns often enough to want shortcuts.

You now have a working toolkit for the gap between code problems and machine problems — the last chapter draws the line between them.

Chapter 15: Dividing Responsibilities

Something broke. You're staring at an error message. The first question is: do I ask my AI coding tool, or do I open Warp?

You face this constantly once you're building real things. And for the first few weeks, you'll probably pick wrong sometimes. Not disastrously — the wrong tool will just tell you it can't help, and you'll switch — but there's friction in the detour. The goal of this chapter is to make the right call on the first try.

The Decision Model

Two questions. Ask them in order.

Is the problem in a file?

If you can open a file and see the problem — a syntax error, a missing import, a misconfigured value — it's a file problem. File problems go to the AI coding tool. It can read the file, trace the issue, and write the fix. That's its terrain.

Is the problem in the machine's running state?

If the problem is about what's executing right now, what the shell can find, what's listening on a port, or what an environment variable is set to — that's machine state. Machine state lives outside the project files. It lives in the OS. It goes to Warp.

The heuristic in one sentence: if the AI coding tool can read it in a file, it owns the problem. If it lives in the machine's running state, Warp owns it.

Walking Through the Decision

Abstractions are only useful when they connect to real errors. Here are the errors from this book, with the right call for each one.

```
ModuleNotFoundError: No module named 'flask'
```

AI coding tool.

This error means Python can't find the Flask package in the current environment. The fix is installing the package — `pip install flask` inside the active virtual environment, or better, making sure the project's dependencies are installed with `pip install -r requirements.txt` or `uv sync`.

That's a project-level operation. The AI coding tool knows the project structure, knows where `requirements.txt` lives, and knows how to install into the right environment. This is what it's for.

It's tempting to call this a "machine-state problem" because it involves an environment. But the virtual environment is part of the project — it's described in project files, managed by project tools, and the AI coding tool knows how to work with it. The problem is in the project layer, not in the raw OS.

```
Error: listen EADDRINUSE: address already in use :::3000
```

Warp.

A process is occupying port 3000. That process exists in the OS's list of running processes. It's not in any file. The AI coding tool can tell you what kind of problem this is and what the fix probably looks like, but it can't check which process is there or kill it.

Open Warp: `# what is on port 3000`. Get the PID. `# kill that process`. Check again. Start the server.

`zsh: command not found: node`

Depends.

This one straddles the line, and it's worth being precise about why.

If Node isn't installed on your machine at all, that's a machine-state problem. Warp can help you install it or diagnose why it's not being found.

If nvm is set up and you just need to switch to the right Node version, that's a project-level operation. The AI coding tool handles version management through nvm — it reads `.nvmrc`, runs `nvm use`, and gets the right version active. It knows the project's requirements. That's not machine state; that's project configuration.

The signal: does the AI coding tool have what it needs to fix this? If nvm is in place and the version is just wrong, yes. If the runtime itself is missing, no — Warp.

`zsh: command not found: gh` **(after installing)**

Warp.

The `gh` binary is on disk, but the shell can't find it because the directory it's in isn't on PATH. The AI coding tool can explain PATH, but it can't look at your current PATH, find the missing directory, and write the fix to your `~/.zshrc` from inside your project files.

This is a shell configuration problem. Open Warp: `# gh command not found but I just installed it`. Follow the diagnostic. If the PATH is missing a directory, `# add [directory] to my`

`PATH permanently`.

`export OPENAI_API_KEY=abc123` disappears after terminal restart

Warp.

Shell configuration — not code. The value needs to live in `~/.zshrc` (or the equivalent for your shell), not just in the current session. The AI coding tool can tell you what to write to the file, but Warp is already in a live terminal session and can write it directly.

`# set OPENAI_API_KEY permanently so it survives terminal restarts`. Done.

A syntax error in `backend/app.py`

AI coding tool.

It's in a file. The AI coding tool can see it, explain it, and fix it. This is the simplest case.

When You Pick the Wrong Tool

Usually nothing bad happens. You just slow down.

If you paste `EADDRINUSE` into the AI coding tool, it will explain what it means, probably suggest a command, and note that it can't run the command itself. That's a couple of exchanges before you end up in Warp anyway.

If you describe a syntax error to Warp, it will generate a command to open the file or search it — something like `grep -n "def " backend/app.py`. Not useless, but also not what your AI coding tool would do with the same information.

The boundary isn't a trap. It's a guide. Both tools will usually point you toward the right place if you're using the wrong one. The cost is a few extra steps.

The Grey Zone

Some problems genuinely straddle the line. Version management is the main one.

nvm and uv are project-aware tools — they read project files to determine which version to use, and the AI coding tool is designed to work with them. So when nvm needs to switch Node versions for `neighborhood-meals`, that's an AI coding tool operation. When the Python virtual environment needs to be created or activated, that's the AI coding tool.

But when nvm itself is broken — misconfigured, not installed, or invisible to the shell — that's a Warp problem. When uv can't be found because it isn't on PATH, that's a Warp problem. The tool that manages your version managers is a machine-state concern.

The pattern: if the problem is about the state of a project tool, the AI coding tool handles it. If the problem is about whether the tool exists on your machine and your shell can find it, Warp handles it.

Agent Mode

One more thing worth knowing before you close this part.

Warp rebranded as an "Agentic Development Environment" in June 2025 and added a feature called Agent Mode, activated with `Cmd+I` on macOS or `Ctrl+I` on Linux and Windows.

Where # generates a single command from a description, Agent Mode takes a multi-step task. You describe the goal — "find what's on port 3000 and kill it" — and Warp runs a sequence of commands to get there, showing you each step.

For the patterns you've built in this book, # is the right entry point. Each problem has a clear first command, and you want to see and understand each step. Agent Mode is useful when the task is genuinely multi-step and you trust Warp to sequence it — or when you want to hand it a task rather than a question.

This is the same distinction you saw with your AI coding tool: there's a difference between asking for a specific change and asking for a goal. As you get more experienced, you'll know when to hand Warp a task rather than a command. # gets you there first.

A Final Word on Whether You Need This

Warp is optional. The author uses it and finds it worth the subscription — not for coding, but as an on-demand systems administrator for the things an AI coding tool wasn't designed to touch. When something is wrong at the machine level, Warp is faster than switching to a browser, searching for the right command, and then coming back to the terminal.

If you're already comfortable at the command line and don't mind looking things up the old way, you don't need Warp to work effectively with an AI coding tool. The decision model in this chapter — code layer vs. machine layer — applies regardless of which terminal you're using.

What you do need is the mental model. That's what this part of the book was actually about.

The Map

You started this book without a map. You had code that ran sometimes, error messages that didn't mean anything, and an AI tool that was confidently guessing about things it couldn't see.

Here's what the map looks like now.

Git and GitHub are version control and remote hosting. Git tracks snapshots. GitHub hosts them. Authentication problems at the GitHub layer — not git problems, not code problems — belong to the auth tooling.

Python environments are isolated containers of packages, one per project. When the environment isn't active or the wrong version is active, you get import errors that aren't import problems. The AI coding tool manages the environment. You know what "activate" means.

Node and npm follow the same logic as Python, with version management via nvm. The `.nvmrc` file declares which version the project expects. The AI reads it. Version mismatches are not broken code.

Machine state is the layer underneath all of it: running processes, PATH, shell configuration, environment variables. The AI coding tool works in the file layer. Warp works in the machine layer. Both are open. You switch based on what you're looking at.

The lines will blur as both tools evolve. Warp will get better at understanding code. AI coding tools will get better at understanding machines. But the underlying distinction — code versus environment — isn't going away. It's not a Warp feature or a Claude feature. It's how operating systems work.

> **Key Takeaway:** Code problems go to the AI coding tool. Machine-state problems go to Warp. When you're not sure which you have, ask: is this in a file, or in the running state of the machine?

You don't need to know every command — you need to recognize what kind of problem you're looking at, and now you do.

Conclusion

Both servers are running. The `neighborhood-meals` frontend is available at `localhost:3000` and the backend is responding on `localhost:8000`. You got here by working through every wall the project threw at you — not by memorizing fixes, but by understanding what you were looking at.

That's what you proved to yourself: you don't need to be a sysadmin. You need to recognize what you're looking at.

That claim was made in the introduction as a thesis. It's yours now. You ran into `fatal: not a git repository` and knew immediately

it was a directory problem, not a git failure. You saw your AI create a virtual environment and you knew why — not because you'd memorized the explanation, but because the pattern was legible. When the Node version was wrong, you recognized the mismatch as a version problem before the error message finished scrolling. When a port was already in use, you knew to go to Warp and not to keep asking the AI to guess.

That recognition is the whole game.

What Actually Changed

Before this book, when your AI touched the terminal, you were watching a stranger do something you couldn't follow. You could tell it worked or didn't work, but you couldn't tell why, and you couldn't tell when to trust the result.

That gap is closed now — not completely, but enough. When your AI coding tool runs `git stash` before switching branches, you know what it's protecting and why. When it runs `source .venv/bin/activate`, you know what state just changed. When it reaches for `nvm use` and you see the Node version shift, you know the project asked for that version and the environment is now complying.

You can follow what it's doing. You can tell whether the fix worked. You can avoid undoing the fix — which is the thing that sends developers into a second, longer spiral.

That's not a small thing. Most of the time the AI is right. But the times it isn't — the times it guesses wrong about your machine state or confidently suggests a command that conflicts with something it can't see — those are the moments where your understanding keeps you from making things worse. You know enough now to notice when something feels off, and you know which tool to reach for when it does.

What This Book Didn't Cover

These are specific gaps, not hedges.

Docker. Once a project is containerized, the environment problem is mostly solved — the container carries its own runtime, its own dependencies, its own configuration. But getting there requires understanding containers: images, volumes, networking, how Docker interacts with your machine's ports and filesystem. That's a different layer of knowledge, and it builds on what you have now.

CI/CD. Continuous integration and continuous deployment are what happen when your code leaves your machine and runs somewhere else automatically — on GitHub Actions, on a build server, in a deployment pipeline. The environment problems are similar to the ones in this book, but they're happening on a machine you can't see, triggered by events in a git repository. Warp can't help you there. The mental model can.

Cloud infrastructure. Running a backend on a VPS or a cloud provider introduces a new layer: the server itself, its operating system, its firewall rules, its logging. The same code-versus-environment distinction applies, but the machine you're diagnosing isn't your laptop.

Advanced Python packaging. This book covered virtual environments, `requirements.txt`, and `pyproject.toml` at the level you need to work with existing projects. Building and distributing a Python package — managing version constraints, building wheels, publishing to PyPI — is a separate subject that assumes you're producing software other people will install.

Windows-specific tooling. Warp runs on macOS and Linux. On Windows, the equivalent landscape includes Windows Terminal, WSL2, and PowerShell — each with its own behavior around PATH, environment variables, and process management. The concepts in this book transfer. The specific commands and configuration files often don't.

These aren't walls you'll hit on day one. But they exist, and when you hit them, knowing they're named and documented is useful. You won't be staring at something that has no explanation.

When You Hit the Next Wall

You will hit walls this book didn't cover. That's not a failure of the book — there's no way to cover everything, and trying would make this three times as long and half as useful.

The pattern, though, is the same every time.

When something breaks and you don't recognize it: describe the error to your AI coding tool and ask what class of problem it is. Not "fix this" — "what kind of problem is this?" Is it a code problem? An environment problem? A configuration problem? A permissions problem? Once you know the class, you know roughly where to look.

Then ask whether the problem lives in a file or in the machine's running state. If it's a file, the AI coding tool owns it. If it's machine state — a process, a PATH entry, a missing binary, a persistent environment variable — Warp owns it.

That two-step has gotten you through fifteen chapters. It will get you through the next fifteen problems.

What changes after this book is not that you stop needing help. It's that your questions get better.

Before, the question was often just "why doesn't this work?" Now it can be "is this a Node version mismatch or a missing package?" or "is this a shell problem or a project problem?" or "did the AI change code, or did it change machine state?" Those are dramatically better questions. They produce dramatically better help — from an AI, from documentation, or from another developer.

That's what understanding buys you. Not independence from tools. Better leverage with them.

What You Can Do Now

You can clone a project, let your AI try to set it up, and stay oriented while it works.

You can see a branch appear and know that `main` is being protected.

You can see a `.venv` directory appear and know that the project just got its own Python room.

You can see `npm ERR! EBADENGINE` and know to ask about Node versions before touching application code.

You can see `EADDRINUSE` and know you're no longer debugging code at all.

Those are small sentences, but they add up to a very different experience of using AI tools on real projects. The terminal stops feeling like a slot machine. The AI stops feeling like a magician. You can follow what happened, which means you can notice when something is off, which means you can course-correct earlier.

For a new developer, that's a real threshold. It is the difference between "I hope this works" and "I see what layer this failure belongs to."

How to Carry This Forward

The next project will not fail in exactly the same way this one did. It might use FastAPI instead of Flask. It might use pnpm instead of npm. It might run in Docker from day one. It might have a `.env` file instead of exporting variables directly in the shell.

That's fine. The surface details can change without invalidating the map.

What you take with you is the habit of asking the right first question. What layer am I in? Is this git state, project dependencies, runtime versioning, or machine state? Once you can sort the problem into the right layer, you are no longer reacting blindly. You have a direction.

That is what makes this kind of knowledge reusable. You are not carrying around a bag of project-specific tricks for `neighborhood-meals`. You are carrying a way to look at setup failures that applies to the next repo, and the one after that.

And that matters because real confidence usually does not feel like certainty. It feels like orientation. You may still need the AI to suggest the exact command, or documentation to confirm a detail, or another developer to sanity-check an unfamiliar tool. But you are no longer lost at the start of the problem.

The Distinction That Doesn't Change

Warp will change. The AI coding tools will change. The specific commands, the UI, the feature names — all of it will drift. Some of what you read in this book may already be slightly out of date by the time you're reading it.

The distinction between the code layer and the environment layer is not going to change. It's not a Warp feature. It's not a Claude feature. It's how operating systems work. Code is text in files. Environments are the runtime state the code executes inside. Those two things have been separate since before any of these tools existed, and the separation is structural, not historical.

Your mental model is built on that structure. Not on specific commands, not on specific tools — on the thing that explains why the commands exist. That's the durable part.

When the next project doesn't run on the first try — and it won't — you'll know what you're looking at.