# How to make Vibe Coding Actually Work

Seth J. Chandler 2025

## 1. Why this matters now

Vibe coding—describing what you want to an LLM and iterating—can be a gentle on‑ramp to computing. You can copy short programs into an editor, run them, and get useful results in minutes. There is a learning curve, and some new vocabulary, but most obstacles are solvable with a few practical habits: keep examples minimal, run early and often, read the first clear error message, and iterate. The goal is not to become a software engineer; it is to acquire just enough operational skill to turn ideas into small working tools.

This guide presents a connected path from first contact with code to sustainable habits. It starts with the hidden assumptions that trip up beginners, then moves to a set of guardrails, and ends with a concrete Week‑1 plan.

## 2. What vibe coding is (and is not)

Vibe coding is not “magic code without learning.” It is a cooperative workflow: you state the outcome, the LLM proposes code, you run it, observe what fails, and refine the prompt or the code. The power comes from rapid feedback, not from skipping every prerequisite. With the right guardrails, it is realistic for absolute beginners.

## 3. Ten hidden assumptions that quietly derail beginners

The frustration most newcomers feel comes from silent prerequisites. None is advanced; together they cascade.

1. Terminal fluency is often assumed. Without cd, ls/dir, pwd, and how to run a script, nothing starts. Spend some time in study and learn mode with this curriculum. A
   1. Ask to be taught the ten most common terminal commands a vibe coder will need.
   2. Ask about relative vs. absolute paths and how failure to understand them can cause problems.
   3. Ask how to move long file names and paths into your terminal session in your operating system.
   4. Ask about how to use arrow keys to avoid excessive typing. Discuss various shells.
2. Either use one of those languages or, if you use Python try out this curriculum in study and learn mode.

Here is a concise curriculum outline for an AI learning assistant, focusing on the core concepts of Python dependency management.

### Python Dependency Management Curriculum

**1. Introduction to Dependencies**

* **Core Concept:** Dependencies are the external packages a project requires.
* **The pip Problem:** Standard pip installs packages globally or into a simple virtual environment, but it has a naive dependency resolver that can easily lead to version conflicts ("dependency hell").
* **The Solution:** Use a modern dependency manager like **Poetry** or **Pipenv**. These tools provide superior dependency resolution and create project-specific, isolated environments with a **lockfile** (poetry.lock or Pipfile.lock) to guarantee deterministic and reproducible builds.

**2. Recognizing Problems**

* **Primary Symptom:** A ModuleNotFoundError is the most common indicator of a missing or incorrect dependency.
* **Proper Diagnosis:** Avoid pip list. Instead, use poetry show --tree or pipenv graph to visualize the complete dependency tree and understand the relationships and versions required by each package.

**3. The Modern Workflow**

* **Initialization:** Start a project by creating its environment (poetry new <project> or pipenv --python <version>).
* **Adding Dependencies:** Install packages using the manager (poetry add <package> or pipenv install <package>). The tool will resolve the correct versions of all sub-dependencies and update the lockfile.
* **Execution:** Run all commands and scripts through the manager (poetry run <command> or pipenv run <command>) to ensure they execute within the correct isolated environment.

3. Project layout and imports are assumed. Files in the wrong place cause “module not found.”

4. $PATH configuration is assumed. If your shell cannot find executables, commands fail. And if you don’t even know what this means, it can make the problem even more frustrating. Use study and learn mode for a 10 minute lesson on how to deal with $PATH in your operating system.

5. Runtime selection is assumed. Language not installed, or the wrong version, derails execution.

6. Isolation is assumed (virtual environments for Python). Global installs cause conflicts and permissions errors. Use of the “Poetry” installer or “uv” for Python rather than things like pip can often help this problem.

7. Error reading is often assumed. You may be presented with a “stack trace” and have no idea what it is even is or how to read it. Stack traces have a structure; the first actionable line is usually near the bottom. Solution: give the error messages to an LLM and ask for help. Do this by cut and paste of text or a screen capture.

8. Quoting and spaces are assumed. Unquoted file paths break commands; learn to use quotes and tab‑completion. Ask study and learn mode for a brief lesson on how to get file paths into the terminal when using your operating system. Ask about drag and drop.

9. Environment boundaries are assumed. Browser JavaScript is not Node; CLI code is not a web page.

10. Version control is assumed. Without Git, experiments feel risky and recovery is painful.

Transition. Once you know these pitfalls, you can adopt a few guardrails that make vibe coding feel coherent instead of fragile.

## 4. Guardrails that keep you moving

### 4.1 Start **without** frameworks or scaffolding

A framework is a large, prebuilt set of conventions and libraries that chooses much of your project’s structure for you (Django or Flask for Python; React, Next.js, or Vue for browser apps; Ruby on Rails for full‑stack sites). Scaffolding tools are generators that create many files and settings for a new project (django‑admin startproject, npm create vite@latest, create‑react‑app, Rails generators, Yeoman). They are valuable once you understand the moving parts, but for first projects they introduce indirection and additional commands that can fail for opaque reasons. Practical rule for week one: one file, no build step. A single HTML file for browser JavaScript, a single .py for Python, or a single main.go for Go.

### 4.2 Defer Docker

If an AI answer says “just use Docker…,” pause. Docker “containers” are excellent for reproducibility and deployment, but they add images, containers, volumes, networks, and Dockerfiles—more layers, more seemingly incomprehensible vocabulary, esoteric distinctions between images and containers, more places to break. Prefer native installs at first. For Python, use a simple virtual environment or Poetry. For Go, compile or run directly. Return to Docker later when you can articulate a deployment need it uniquely solves.

### 4.3 Pick low‑friction language targets for tiny projects

When you do not plan to learn the language in depth yet, reduce setup:

• JavaScript in the browser: no installs; put a script tag in an .html file and open it.

• Go: strong standard library, fast, and compiles to a single binary; few dependency pitfalls.

• Python: excellent and ubiquitous; keep early projects dependency‑free or manage them with Poetry.

There is a trade‑off to acknowledge: JavaScript and Go may have fewer ready‑made packages for niche legal tasks, but that may well be acceptable for first projects whose purpose is confidence and momentum.

### 4.4 Python without the pain: Poetry for clean, reproducible projects

Python’s chief beginner hazard is dependency hell (conflicting versions, global installs, broken environments). Poetry mitigates this with project isolation, precise version locking, and a memorable workflow. Minimal Poetry flow: 1. pipx install poetry (or install per docs) 2. poetry new myproject (or poetry init in an existing folder) 3. poetry add requests (example package) 4. poetry run python main.py (runs inside the project’s environment) Optional: poetry shell to activate the env for multiple commands. Use Poetry the moment you add even one third‑party package, or whenever you want classmates to reproduce your setup.

### 4.5 Learn just enough Git and GitHub

On a gain‑to‑pain ratio, Git is unusually favorable. A five‑command subset changes your experience: • git init (start a repository) • git add -A and git commit -m “message” (save a snapshot) • git log (see history) • git restore –source – (recover a prior version) Optional: git branch -c feature‑x (try ideas safely). Benefits you feel immediately: easy undo, fearless experiments, off‑device backup via GitHub, and simple sharing. Spend 40 minutes learning git and Github. Great investment. Free for most plans.

### 4.6 Use Study and Learn mode to close gaps just‑in‑time

Modern study modes in large language models can turn friction into short, targeted lessons: a 90‑second PATH explainer with two commands; a readiness checklist tied to your project type (CLI Python script, browser page, small Go tool); tiny drills that build fluency and preserve momentum.

## 5. Final counsel

Keep projects small; treat line count as a risk indicator. Avoid frameworks and scaffolding until you have a couple of working wins. Defer Docker until you know why you need it. Prefer low‑friction environments (browser JS, Go, or Python with Poetry). Learn just enough Git to feel safe experimenting. Vibe coding builds confidence when you reduce operational complexity and let the LLM help with what it already does best: producing small, working programs you can actually run.