

What The F*ck is Vibe Coding?

A blunt little book about building software with AI without lying to yourself.

Author: Eryk Klopotoski

Copyright © 2026. All rights reserved.

Table of Contents

Part 0 - Opening: The attorney who says it's easy

Part 1 - Mindset: Your clear image

Part 2 - Tools: VS Code, terminal, Git, GitHub, AI

Part 3 - Skills: Reading, debugging, testing

Part 4 - Training: A simple 4-week plan

Part 5 - 15 Essential Missions (practical prompts)

Part 6 - Optional Nerd Reads

Part 7 - Closing: Back to the attorney

Glossary

Facts and sources

PART 0 - OPENING

Vibe coding is easy. (Said every confident person, five minutes before production.)

I'm having coffee with an attorney who is annoyingly up to date on technology. He's not the stereotype. He actually reads the release notes. He can explain what an API is without sweating.

He looks at my laptop, sees a chat window and a code editor, and says:

"Vibe coding is easy. You just prompt it."

I do the polite smile first. The one that says, "I respect you as a professional," while my brain is screaming, "Sir, please don't build bridges with vibes."

"Easy like what?" I ask. "Easy like it compiles? Easy like it works? Easy like it won't leak secrets and send invoices to the wrong people?"

He shrugs. "You're overthinking it. The AI does the hard part. You just tell it what you want."

This is where the conversation gets fun.

Because he's not wrong about the speed. The first draft of software is now cheap. The problem is that the first draft has always been cheap. The expensive part is everything after the first draft.

So I do what any reasonable person would do. I start a fight, politely. We go back and forth.

- He argues: prompts are enough; you can ship in a weekend.
- I argue: prompts are only the start; you can also ship a disaster in a weekend.
- He argues: the tool is smart.
- I argue: smart tools still need adults in the room.

After ten minutes, he leans back, smiles, and does the thing that makes this a book instead of a weird cafe argument.

*"Okay," he says. "Then tell me - what the f*ck is vibe coding?"*

And that's the whole point of this book.

BANANA (simple version)

Vibe coding is not 'typing a magic prompt.' It's a loop: you describe what you want, the AI generates something, you run it, you see what's wrong, you tighten the instructions, you fix the code, you test it, and you repeat until it behaves. The vibe is your intent. The coding is the discipline.

PART 1 - MINDSET

Chapter 1 - The clear image

If you want vibe coding to work, you need one thing more than prompts: a clear image of what you want out of the project.

Not a vision board. A definition of done.

If you can't explain what 'done' means, the AI will happily invent it for you. And it will sound confident. That's how you get software that looks impressive and fails the moment a real user touches it.

So here is your first tool. Not VS Code. Not Git. A one-page spec.

The Vibe Spec (one page)

Write this before you generate anything:

- **Outcome:** what exists when we're done (what a user can do).
- **User:** who uses it and why they care.
- **Inputs and outputs:** real examples (use fake data if you must).
- **Constraints:** time, budget, platform, privacy, and anything you refuse to compromise on.
- **Acceptance tests:** 'I will accept this only if...' (3 to 10 bullet points).
- **Non-goals:** what you are not building (so scope stops expanding).

BANANA (simple version)

If you don't write your destination, you can't complain about the route.

Chapter 2 - The loop

Vibe coding is not a single prompt. It's a feedback loop you can run every day without losing your mind.

- Write (or update) the Vibe Spec.
- Ask the AI for a plan and a file structure.
- Generate the smallest usable version.
- **Run it. Don't debate it. Run it.**
- Use errors and tests as evidence.
- Fix one thing at a time.
- Commit changes with Git. Repeat.

Optional Nerd Read: Why loops beat big prompts

Large prompts feel productive, but small loops produce proof. Each loop produces an artifact you can run (or a test you can fail). That evidence is what keeps you out of fantasy-land.

Chapter 2a - What AI actually does (and doesn't do)

Here's the honest picture of software development with AI:

What AI speeds up dramatically:

- **Code generation** from clear specifications
- **Boilerplate and structure** (file trees, configs, basic CRUD)
- **Pattern matching** (implementing known solutions)
- **Syntax and formatting** (writing correct code structure)

- **Documentation** (READMEs, comments, API docs)

What AI can't replace (you still own this):

- **Requirements specification** - Defining what 'done' actually means
- **Roadmapping** - Deciding what to build first, second, third
- **Testing strategy** - Determining what needs to be tested and how
- **Architecture decisions** - Choosing technologies, patterns, tradeoffs
- **Edge case identification** - Spotting the weird inputs that break things
- **Security review** - Understanding threat models and vulnerabilities
- **Production monitoring** - Watching what breaks and why

Think of AI as a senior developer who:

- **Has no context about your project** (unless you give it)
- **Forgets things constantly** (context window limits)
- **Lacks common sense** (can't judge if something is stupid)
- **Sounds confident even when wrong** (hallucinations are statistics, not lies)
- **Needs supervision** on every output

This means: You're the adult in the room. AI is the fast typist with a great memory for patterns but zero judgment.

BANANA (simple version)

AI accelerates implementation. You still own: specification, verification, and judgment.

PART 2 - TOOLS

Chapter 3 - The minimum toolset

Let's keep this simple. You need a workshop, a time machine, and a lie detector.

Workshop: Visual Studio Code

Visual Studio Code (VS Code) is a lightweight code editor with support for debugging, task running, and version control[1]. It's free for private or commercial use, under Microsoft's product license[1][2].

Two facts that confuse people:

- The open-source repository is MIT-licensed code (Code - OSS / vscode).

- The Microsoft-built Visual Studio Code download includes Microsoft-specific customizations and is released under a Microsoft product license[1][3].

Setup instructions (5 minutes):

1. **Download:** Visit <https://code.visualstudio.com/>
2. **Install:** Run installer, accept defaults
3. **Open folder:** File → Open Folder (this is your project workspace)
4. **Install extensions** (optional but useful):
 - Python (if using Python)
 - ESLint (if using JavaScript/Node)
 - GitLens (helpful Git visualizations)

BANANA (simple version)

VS Code is where you look at the mess the AI made, and turn it into something you can live with.

Terminal: where reality lives

The terminal is how you run the app, run tests, install dependencies, and see errors. Vibe coding without running the code is just creative writing.

Access terminal in VS Code: View → Terminal (or Ctrl+/ Cmd+)

Git: your time machine

Git is a free and open source distributed version control system[4]. It's released under the GNU General Public License v2.0[5]. In a distributed system, every developer keeps a full copy of the project history, not just the latest files[6].

Git lets you:

- Save checkpoints (commits).
- Compare changes (diff).
- Undo mistakes without panic.
- Experiment safely (branches).

Basic Git setup (first time):

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

Essential Git commands you'll use constantly:

```
git init # Start tracking a project
git status # See what changed
git add . # Stage all changes
git commit -m "message" # Save checkpoint
```

git log # View history
git diff # See what changed (detailed)

Why this matters for vibe coding: When AI generates bad code, you can `git diff` to see exactly what changed, and `git reset` to undo it without panic.

BANANA (simple version)

Git = undo button for your entire project. Use it obsessively.

GitHub: where your repo lives

GitHub is a hosting platform for Git repositories. It also adds collaboration tools (issues, pull requests, reviews) and automation (Actions/CI).

GitHub offers free plans with unlimited public repositories, and private repositories with feature limits depending on account type[7][8].

Quick GitHub setup:

1. Create account at <https://github.com>
2. Create new repository (click "+")
3. Connect local project:

```
git remote add origin https://github.com/yourusername/yourproject.git  
git branch -M main  
git push -u origin main
```

AI assistant: the generator

Pick one AI assistant you'll actually use daily. Your goal is not to collect tools. Your goal is to build a repeatable loop.

If you use GitHub Copilot, note that pricing is subscription-based (for example, Copilot Pro is listed at \$10/month on GitHub's pricing page)[9]. Pricing changes; check the official page before you commit a budget.

Popular options (as of 2026):

- GitHub Copilot (code completions in editor)
- ChatGPT / Claude (chat-based code generation)
- Cursor (AI-first code editor)

Chapter 4 - Costs (the honest version)

Most 'vibe coding' costs don't come from code. They come from decisions. Cost drivers you should expect:

- AI subscription(s).
- Hosting (always-on apps cost more than apps that sleep).

- Databases (size + traffic).
- CI minutes (automated tests on every change).
- Team seats (tools priced per user).

Docker note: Docker Desktop is free for personal use and some small-business use cases, but paid subscriptions can apply depending on how you use it[10][11]. Don't guess. Read the current Docker terms.

BANANA (simple version)

If you're building a hobby project, your biggest cost is usually the AI subscription. If you're building a real product, your biggest cost becomes 'keeping it alive' (hosting, monitoring, support).

PART 3 - SKILLS

Chapter 5 - Prompting is not the skill

Prompting is like asking questions. The real skill is knowing what questions matter.

Managing context: your project memory

AI assistants have a critical weakness: they forget. Every chat has a limited context window (the amount of text they can 'remember'). When you paste huge files, old messages, and five error logs, important constraints get pushed out.

Three tools to fight context rot:

1. **copilot-instructions.md** (or similar) - A file in your project root that contains:
 - Project description (2-3 sentences)
 - Tech stack (languages, frameworks)
 - Key constraints (no auth, minimal dependencies, etc.)
 - File structure overview
2. **Custom instructions** (AI assistant settings) - Platform-specific persistent instructions:
 - "Always include unit tests"
 - "Use Python 3.11+ syntax"
 - "Prefer standard libraries over dependencies"
3. **Custom agents / roles** - Pre-configured AI personas:
 - "Senior Python engineer"
 - "Security reviewer"

- "Test-driven developer"

Why this matters: You stop repeating yourself. The AI reads your project context from files instead of forgetting it after 20 messages.

Practical example: `copilot-instructions.md`

Project Context

App: Budget tracker CLI tool

Stack: Python 3.11, SQLite, no web framework

Constraints:

- No external APIs
- All secrets in environment variables
- Minimum 80% test coverage

File structure:

- `main.py` - CLI entry point
- `budget.py` - Core logic
- `db.py` - Database operations
- `tests/` - All tests

Place this file in your project root. Reference it in prompts: "See [copilot-instructions.md](#) for project context."

Context rot: why you should start fresh chats

When a chat gets long (50+ messages), the AI becomes dumber. Why? Old messages, errors, and dead-end attempts clog the context window. The AI can't tell what's current vs. abandoned.

Solution: Start a new chat for new features. Copy forward ONLY:

- The project context file
- Current working code (not all the failed attempts)
- The specific thing you're building now

A useful prompt structure

A useful coding prompt usually includes:

- **Goal:** what you want to build.
- **Context reference:** "See [copilot-instructions.md](#)" or paste 2-3 sentences.
- **Constraints:** language, framework, libraries, environment.
- **Output format:** files, file tree, tests, commands.

- **Examples:** sample input/output.
- **Definition of done:** tests that must pass, behaviors that must exist.

Template prompt (use Git if relevant):

Build a minimal working version of [feature].

Context: [See [copilot-instructions.md](#) or paste project summary]

Constraints:

- Language: [Python / Node / etc.]
- No external services
- Include unit tests
- Use Git: commit after implementation with message "[feature]: implement X"

Deliverables:

1. File tree
2. Complete code
3. Tests (at least 5)
4. Commands to run and test
5. Git commands to commit

Definition of done:

- All tests pass
- Handles edge cases: [list specific ones]
- Code committed to Git

Note: The role description ("You are a senior engineer") has been removed because coding agents already have that in their system prompt.

BANANA (simple version)

If your prompt doesn't say what 'good' looks like, you're outsourcing taste to a machine.

Chapter 6 - Reading code beats writing code (at first)

Early on, your job is not to produce genius code. Your job is to spot nonsense. Practice these:

- Follow the data: where does input enter, and where does it leave?
- Trace the happy path, then the failure path.
- Ask: what happens when input is empty, huge, or weird?
- Check if secrets are hardcoded (they shouldn't be).

- Look for TODO comments (AI loves leaving these).
- Use `git diff` like a microscope - see exactly what AI changed.

BANANA (simple version)

You don't need to write perfect code. You need to recognize broken code before you ship it.

Chapter 7 - Debugging is a discipline

Debugging is not yelling at the AI. Debugging is reducing uncertainty with evidence.

- Reproduce the bug.
- Write a failing test (if possible).
- Change one thing.
- Run again.
- Commit when it's fixed (with Git).

Optional Nerd Read: Why 'one change at a time' matters

AI-generated code can fail for multiple reasons at once. If you change three things, you can't learn which change mattered. Small changes turn debugging into science instead of folklore.

Chapter 8 - Tests are your anti-hallucination layer

AI can sound right and still be wrong. Tests don't care how confident the code feels.

Start with three kinds of tests:

- One happy-path unit test.
- One edge-case test.
- One 'bad input' test.

Example (Python):

```
def test_happy_path():
    result = add_expense("Groceries", 50.0)
    assert result.success == True

def test_edge_case():
    result = add_expense("", 0.0) # Empty name, zero amount
    assert result.success == False

def test_bad_input():
    result = add_expense("Food", -10.0) # Negative amount
    assert result.success == False
```

BANANA (simple version)

If you don't test it, you don't know it.

PART 4 - TRAINING

Chapter 9 - A simple 4-week plan

You can learn vibe coding the same way you learn anything: by shipping small things repeatedly.

Week 1 - Foundation

Goals:

- Install tools (VS Code, Git, Python/Node).
- Build a tiny CLI tool.
- Write 10 tests.
- Push to GitHub.

Recommended missions: 1, 3, 7

Week 2 - Small web app

Goals:

- One data model.
- Validation.
- Useful errors.
- Basic README.

Recommended missions: 4, 5, 8

Week 3 - Integration + deploy

Goals:

- Use one external API.
- Deploy somewhere simple.
- Handle secrets correctly.

Recommended missions: 9, 12, 13

Week 4 - Real project

Goals:

- Write a Vibe Spec.
- Ship v1.
- Add logging/monitoring.
- Write a postmortem.

Recommended missions: 2, 10, 11, 14, 15

BANANA (simple version)

The goal is not to build the perfect app. The goal is to build the habit of turning vibes into proof.

PART 5 - 15 ESSENTIAL MISSIONS

Copy/paste prompts you can use immediately. Each mission focuses on a specific skill. No artificial one-page limit - quality over constraint.

Note: Role descriptions have been removed from prompts. Coding agents already have role context in their system prompts.

Mission 1: Weekend App Ideas

Skill: Idea generation with tight constraints.

Prompt

Task: Pick a weekend app idea.

Context: [See [copilot-instructions.md](#), or: I want to build something small and ship it fast]

Constraints:

- Shippable in a weekend
- No payments, no login (unless strongly recommended)
- Minimal database
- Must include Git workflow in plan

Output:

1. 5-10 app options
2. For each: who it's for, core workflow, definition of done

3. Suggested Git commit strategy

Keep it blunt and practical.

What good looks like

- Constraints respected (no surprise features).
- Each idea has a clear user and workflow.
- Each idea has a definition of done.
- Git workflow included in plan.

Common traps

- Ideas too big for timeline.
 - Features instead of workflows.
 - No acceptance tests.
-

Mission 2: Write a Vibe Spec

Skill: Turning vague ideas into concrete specifications.

Prompt

Task: Write a one-page Vibe Spec.

Context:

- App idea: [describe in 2 sentences]
- Target user: [who]
- Platform: [web / mobile / CLI]

Deliverable:

A one-page spec with:

- Outcome (what users can do when done)
- Workflow (step-by-step user actions)
- Inputs/outputs (with examples)
- Constraints (technical and business)
- Acceptance tests (pass/fail criteria)
- Non-goals (what we're NOT building)

Write it so a human can build it, not as a vibe poem.

What good looks like

- Spec is one page and concrete.
- Inputs/outputs include real examples.
- Non-goals are explicit.
- Someone else could build from this spec.

Common traps

- Spec is fluffy and abstract.
 - No edge cases.
 - No acceptance tests.
-

Mission 3: Scaffold a Python Project

Skill: Project structure, tests, Git setup.

Prompt

Task: Scaffold a complete Python project with Git.

Context:

- App: [describe in one paragraph]
- Python 3.11+
- Local development only

Deliverables:

1. File tree with purpose of each file
2. Runnable code
3. Unit tests (minimum 5, covering happy path + edge cases)
4. README with setup and run commands
5. Git initialization commands
6. .gitignore for Python projects

Constraints:

- Minimal dependencies (prefer standard library)
- Environment variables for any config
- All secrets in .env (never hardcoded)
- Git: include initial commit strategy

Quality bar:

- Project runs with one command

- All tests pass
- Git repo initialized and clean

What good looks like

- Project runs locally: `python main.py`
- Tests pass: `pytest`
- Git initialized with clean initial commit.
- README includes setup steps.
- No hardcoded secrets.

Common traps

- Missing run instructions.
 - No tests or tests that don't run.
 - Over-engineered dependencies.
 - Secrets hardcoded.
-

Mission 4: Data Model Design

Skill: Thinking through data structures and relationships.

Prompt

Task: Design a data model with examples.

Context:

- App: [describe]
- Database: [SQLite / PostgreSQL / in-memory]
- Expected scale: [small / medium / production]

Deliverables:

1. Entity list with attributes
2. Relationships between entities
3. Sample data (at least 3 realistic examples per entity)
4. Edge cases to handle
5. What data we explicitly do NOT store (privacy/security)

Output format: Tables or objects with examples, not abstract schemas.

What good looks like

- Concrete examples, not abstract types.
- Edge cases identified (empty strings, null values, duplicates).
- Privacy considerations explicit.

Common traps

- Abstract schemas without examples.
 - Missing edge case handling.
 - No privacy considerations.
-

Mission 5: Build a CLI Tool

Skill: Command-line interfaces, argument parsing, error handling.

Prompt

Task: Build a working CLI tool.

Context:

- Language: [Python / Node]
- Function: [what the tool does]
- Users: [technical / non-technical]

Requirements:

1. File tree
2. Complete implementation
3. Help text and usage examples
4. Unit tests (minimum 5)
5. Error messages that explain what went wrong and how to fix
6. README with installation and usage
7. Git: commit with message "feat: implement [tool name] CLI"

Constraints:

- Use standard library for argument parsing (argparse / commander)
- All errors caught and explained
- Include `--help` flag
- Git: include `.gitignore`

Definition of done:

- `tool --help` shows clear usage

- All commands work with valid input
- Invalid input produces helpful errors
- Tests pass

What good looks like

- Clear help text.
- Errors explain what's wrong and how to fix.
- Tests cover happy path and error cases.
- README with examples.

Common traps

- Cryptic error messages.
 - No help text.
 - Doesn't handle invalid input.
-

Mission 6: Debugging with Failing Tests

Skill: Test-driven debugging, minimal reproduction.

Prompt

Task: Turn a bug report into a test, then fix it.

Evidence:

- What I expected: [describe]
- What happened: [describe]
- Error output / logs: [paste]

Your job:

1. Explain likely root cause (2-3 sentences)
2. Write a minimal failing test that reproduces the bug
3. Propose a minimal fix
4. Verify: test fails before fix, passes after
5. Git: commit with message "fix: [brief description]"

Constraints:

- Change as little as possible
- Test must be minimal and clear
- No "maybe" fixes - verify with test

What good looks like

- Fix is minimal and targeted.
- Test demonstrates bug and verifies fix.
- Explanation ties to evidence.
- Git commit clear and atomic.

Common traps

- Multiple changes at once.
 - No reproduction test.
 - Fix "seems right" but isn't verified.
-

Mission 7: Add External API Integration

Skill: API calls, secrets management, error handling, testing with mocks.

Prompt

Task: Integrate one external API safely.

Context:

- API: [which API - weather, currency, maps, etc.]
- Language: [Python / Node]
- Usage: [what data you need and why]

Deliverables:

1. API client module (separate from business logic)
2. Configuration via environment variables
3. Error handling for: network errors, rate limits, bad responses
4. Unit tests using mocks (no real API calls in tests)
5. Example .env file (with fake/example values)
6. README section on API setup
7. Git: commit with message "feat: integrate [API name]"

Constraints:

- API keys ONLY in environment variables
- Include .env in .gitignore
- Graceful failure if API unavailable
- Tests don't require real API access

Quality bar:

- Works with valid API key
- Clear errors when API key missing or invalid
- Tests pass without network access

What good looks like

- API key never in code.
- Clear errors for common failures.
- Tests use mocks, don't hit real API.
- README explains setup.

Common traps

- Hardcoded secrets.
 - No error handling for network failures.
 - Tests require real API credentials.
-

Mission 8: Add Structured Logging

Skill: Production-ready observability, debugging in production.

Prompt

Task: Add structured logging you can search and filter.

Context:

- App: [describe]
- Language: [Python / Node]
- Environment: [local / deployed]

Requirements:

1. Use proper logging library (not print statements)
 - Python: `logging` module with JSON formatter
 - Node: `winston` or `pino`
2. Log levels: `DEBUG`, `INFO`, `WARNING`, `ERROR`
3. Structured format (JSON) with fields:
 - `timestamp`
 - `level`
 - `message`

- context (user_id, request_id, etc.)
- 4. Different log levels for different environments
- 5. Examples of what to log:
 - Requests/responses (excluding secrets)
 - Errors with stack traces
 - Performance milestones
- 6. Git: commit with message "feat: add structured logging"

Constraints:

- Never log secrets or PII
- Logs must be searchable (JSON format)
- Configure via environment variable (LOG_LEVEL)

Definition of done:

- Can filter logs by level
- Can search logs by context fields
- No secrets in logs

What good looks like

- JSON-formatted logs.
- Clear context in every log entry.
- No secrets logged.
- Easy to search and filter.

Common traps

- Using `print()` instead of logging.
- Logging secrets or sensitive data.
- No structure (can't search effectively).

Mission 9: Deployment Checklist

Skill: Production readiness, operational thinking.

Prompt

Task: Create a deployment checklist for this app.

Context:

- App: [what it does]

- Hosting target: [local / cloud platform / VPS]
- Scale: [hobby / small business / production]

Deliverable: A practical checklist covering:

1. **Pre-deployment:**

- All tests passing
- Secrets in environment variables (not code)
- .env.example provided (no real secrets)
- Dependencies documented
- Git: clean working directory, tagged release

2. **Deployment:**

- Health check endpoint
- Startup commands
- Port configuration
- Environment variable setup

3. **Post-deployment:**

- Health check returns 200 OK
- Logs are flowing
- Basic smoke test (can it handle one request?)
- Monitoring configured

4. **Rollback plan:**

- How to revert to previous version
- Database migration rollback (if applicable)
- Data backup verification

5. **Operational:**

- Where to check logs
- How to restart service
- Who to contact if things break
- Git: tag with version number

Keep it lightweight but complete.

What good looks like

- Checklist is actionable (yes/no items).
- Covers secrets, health checks, rollback.
- Someone else could deploy from this.

Common traps

- No health check.
 - No rollback plan.
 - Keys in code or repo.
-

Mission 10: Refactor Without Breaking

Skill: Code cleanup, behavior preservation, testing discipline.

Prompt

Task: Refactor messy code while proving behavior stays the same.

Context:

- Code: [paste the messy code]
- Language: [Python / Node]

Constraints:

- Do NOT change behavior
- Keep inputs/outputs identical
- Improve readability and structure

Deliverables:

1. Tests BEFORE refactoring (prove current behavior)
2. Refactored code
3. Tests AFTER refactoring (prove behavior unchanged)
4. Explanation of changes (2-3 sentences)
5. Git: two commits
 - "test: add tests for [function] before refactor"
 - "refactor: improve [function] readability"

Quality bar:

- All tests pass before and after
- Code is more readable
- No new bugs introduced

What good looks like

- Behavior unchanged (proven by tests).
- Names clearer.

- Smaller functions or modules.
- Git history shows test-first approach.

Common traps

- Changing behavior silently.
 - No tests to verify equivalence.
 - Adding features during refactor.
-

Mission 11: Security Review Checklist

Skill: Threat modeling, vulnerability assessment.

Prompt

Task: Review this app for common security issues.

Context:

- App: [describe]
- Stack: [languages, frameworks, database]
- Users: [internal / public]
- Data handled: [describe sensitive data]

Checklist to review:

1. **Secrets and credentials:**
 - No hardcoded secrets
 - Environment variables used
 - .env in .gitignore
 - No secrets in Git history
2. **Input validation:**
 - All user inputs validated
 - SQL injection prevented (parameterized queries)
 - Path traversal prevented
 - File upload restrictions (if applicable)
3. **Authentication/Authorization:**
 - Passwords hashed (if applicable)
 - Session management secure
 - Authorization checks on sensitive actions

4. **Dependencies:**

- No known vulnerabilities (run `npm audit / pip check`)
- Minimal dependencies
- Dependencies from trusted sources

5. **Data exposure:**

- No sensitive data in logs
- No stack traces in production
- Error messages don't leak internal details

6. **Infrastructure:**

- HTTPS in production (not HTTP)
- Rate limiting on APIs
- Secure headers configured

Output:

- Issues found (with severity: high/medium/low)
- Fix recommendations
- Commands to verify fixes

What good looks like

- Concrete issues identified.
- Severity assigned.
- Actionable fixes.

Common traps

- Generic advice without context.
- No severity ranking.
- Missing practical fixes.

Mission 12: Add Health Checks and Monitoring

Skill: Operational visibility, production readiness.

Prompt

Task: Add health checks and basic monitoring.

Context:

- App: [describe]
- Platform: [web app / API / CLI]
- Deployment: [local / cloud]

Requirements:

1. **Health check endpoint** (if web/API):
 - Route: /health or /healthz
 - Returns: JSON with status
 - Checks:
 - App is running
 - Database connection (if applicable)
 - External dependencies reachable
 - Response: 200 OK if healthy, 503 if not
2. **Startup checks:**
 - Verify environment variables present
 - Test database connection
 - Fail fast with clear error if not ready
3. **Basic metrics to log:**
 - Request count
 - Error count
 - Response time (P50, P95, P99 if possible)
4. **Alerting strategy** (describe, don't implement):
 - What should wake someone up at 2am?
 - What can wait until morning?
5. Git: commit with message "feat: add health checks and monitoring"

Definition of done:

- Health endpoint returns status
- Clear errors if dependencies unavailable
- Logs include key metrics

What good looks like

- Health check returns clear status.
- Fails fast on startup if misconfigured.
- Metrics logged in structured format.

Common traps

- Health check always returns 200 (even when broken).
 - No dependency checks.
 - Metrics not actionable.
-

Mission 13: Write a Runbook

Skill: Operational documentation, incident response.

Prompt

Task: Write a runbook for operating this app.

Context:

- App: [describe]
- Team: [solo / small team / company]
- On-call: [yes / no]

Sections:

1. **Overview:**
 - What this app does (2 sentences)
 - Who uses it
 - Critical vs. non-critical
2. **How to deploy:**
 - Commands to deploy
 - How to verify deployment succeeded
 - Rollback procedure
3. **Common issues and fixes:**
 - Issue 1: [symptom] → [diagnosis] → [fix]
 - Issue 2: [symptom] → [diagnosis] → [fix]
 - Issue 3: [symptom] → [diagnosis] → [fix]
4. **Where to look:**
 - Logs location
 - Metrics dashboard (if any)
 - Health check URL
5. **Emergency contacts:**

- Who to contact for what

6. **Git workflow:**

- Branching strategy
- When to create a release tag
- How to hotfix production

Keep it practical. Write for someone operating this at 2am.

What good looks like

- Actionable steps for common problems.
- Clear log locations.
- Emergency contacts listed.
- Written for stress and fatigue.

Common traps

- Too abstract (no specific commands).
 - No common issues documented.
 - No emergency contacts.
-

Mission 14: Build a Complete CRUD App

Skill: Full-stack implementation, end-to-end workflow.

Prompt

Task: Build a complete CRUD (Create, Read, Update, Delete) app.

Context:

- App: [e.g., "expense tracker", "todo list", "inventory manager"]
- Stack: [Python + Flask / Node + Express / etc.]
- Database: [SQLite / PostgreSQL]
- Interface: [web UI / CLI]

Deliverables:

1. **Complete implementation:**

- Create: Add new item
- Read: List all items, view single item
- Update: Edit existing item
- Delete: Remove item

- Validation on all inputs
- 2. **File structure:**
 - Clear separation: routes, models, views
 - Config in environment variables
 - Static files organized (if web)
- 3. **Tests:**
 - Unit tests for models (10+)
 - Integration tests for full workflows (5+)
 - Edge cases covered
- 4. **Documentation:**
 - README with setup, run, test commands
 - API documentation (if API)
 - Example usage
- 5. **Git workflow:**
 - Initial commit: project scaffold
 - Feature commits: each CRUD operation
 - Test commits: tests added/updated
 - Final commit: documentation

Constraints:

- Minimal dependencies
- All secrets in environment variables
- Graceful error handling
- Input validation everywhere

Quality bar:

- Runs with one command
- All tests pass
- Handles invalid input gracefully
- Git history is clean and logical

What good looks like

- All CRUD operations work.
- Tests cover happy path and edge cases.
- Clear error messages.
- README sufficient for another developer.

- Git commits tell the story of development.

Common traps

- Missing validation.
 - No error handling.
 - Tests don't cover edge cases.
 - Git history is chaotic.
-

Mission 15: Postmortem Template

Skill: Learning from failures, blameless retrospective.

Prompt

Task: Create a postmortem template for when things break.

Context:

- Team size: [solo / small / large]
- App: [describe]

Template sections:

1. **Incident summary:**

- What broke?
- When did it break?
- When was it detected?
- When was it resolved?
- Impact: [users affected, duration, data lost]

2. **Timeline:**

- [Time] - [Event]
- [Time] - [Action taken]
- [Time] - [Result]

3. **Root cause:**

- What was the underlying cause? (technical detail)
- Why did it happen? (process failure)

4. **Contributing factors:**

- What made it worse?
- What delayed detection or resolution?

5. **What went well:**

- What helped fix it faster?
- What prevented worse damage?

6. **Action items:**

- [] Action 1 [owner] [due date]
- [] Action 2 [owner] [due date]
- [] Action 3 [owner] [due date]

7. **What we learned:**

- Key takeaways (3-5 bullets)

8. **Git forensics:**

- Commits related to incident
- Code changes that introduced issue
- How to prevent in CI/tests

Rules:

- Blameless (no "who screwed up")
- Specific (no vague "improve monitoring")
- Actionable (items have owners and dates)

Output: Template in markdown format.

What good looks like

- Blameless tone.
- Specific action items with owners.
- Timeline helps reconstruct events.
- Focuses on process, not people.

Common traps

- Blaming individuals.
 - Vague action items.
 - No timeline.
 - No follow-through plan.
-

PART 6 - OPTIONAL NERD READS

If you like the simple version, stop here. You're not missing anything required to ship.

If you want to understand why the AI behaves the way it does (especially when it is confidently wrong), keep reading.

Nerd Read 1 - Why the model 'hallucinates' code

Large language models predict the next token. They do not 'know' your codebase unless you give it to them. If a function name feels plausible, the model may invent it. That is not malice. That is statistics wearing a confident face.

BANANA (simple version)

The model is a guesser. A very good guesser. Still a guesser.

Nerd Read 2 - Context windows (why it forgets)

Models have a limited context window. If you paste a huge file tree and five error logs, something will get pushed out. When the model forgets a constraint, it's usually because you didn't restate it or you overloaded the context.

Context window sizes (as of 2026):

- GPT-4: ~128K tokens (~300 pages)
- Claude 3: ~200K tokens (~500 pages)
- Gemini: ~1M tokens (~2000+ pages)

But practical limits are lower - models get "distracted" long before hitting limits.

Solution: Use `copilot-instructions.md` and start fresh chats regularly.

Nerd Read 3 - Why tests work so well with AI

Tests are executable truth. They turn a vague request into a measurable outcome. In vibe coding, tests are your contract with the machine.

Tests prevent:

- Confident nonsense (test fails)
- Silent behavior changes (test fails)
- Scope creep (test remains focused)

Nerd Read 4 - Guardrails you'll see in real teams

- **Formatting/linting** so code stays readable.
- **CI (automated tests)** on every pull request.
- **Secret scanning** to prevent credential leaks.
- **Dependency scanning** to reduce supply-chain risk.
- **Logging and monitoring** so production failures are visible.
- **Pre-commit hooks** to run checks before commit.
- **Code review** (human or AI-assisted) before merge.

Nerd Read 5 - Awesome Copilot Resources

GitHub maintains a community-driven repository of prompts, custom instructions, agents, and skills for working more efficiently with AI coding assistants:

Resource: <https://github.com/github/awesome-copilot>

What's in there:

- Prompt libraries for common tasks
- Custom instruction templates
- Agent configurations for specific roles
- Skills and patterns for advanced usage

Worth exploring if you're building production systems with AI.

Nerd Read 6 - Recommended video

"Working with AI in 2024" by [Creator] -
<https://www.youtube.com/watch?v=oXoXNG65rfg>

Key insights from this video:

- Using `copilot-instructions.md` for persistent context
- Custom instructions and agents
- Keeping project requirements in separate files

- Reducing hallucinations with explicit context
-

PART 7 - CLOSING

Chapter 10 - Back to the attorney

I meet the attorney again a few weeks later. He has a new confidence. The quieter kind.

"Okay," he says, "I get it now. Prompting is easy. Shipping is not."

I nod. Because this is the only honest conclusion.

Vibe coding is powerful. It's also dangerous in the boring ways: missing tests, unclear requirements, hidden security holes, and 'works on my machine' energy.

He asks one more question.

"So what's the definition? The one-liner I can tell my team."

Here it is:

Vibe coding is turning intent into working software through fast AI generation and strict human steering.

The vibe is your vision. The steering is your discipline. You need both.

BANANA (simple version)

If you can't explain what you want, the AI will explain something else. And it will do it confidently.

GLOSSARY

AI assistant - A tool that generates text and code based on your prompts. Useful. Also capable of confident nonsense.

Branch - A parallel line of work in Git. It lets you experiment without breaking main.

CI (Continuous Integration) - Automation that runs tests and checks on changes before they merge.

Commit - A saved checkpoint in Git with a message describing what changed.

Context window - The amount of text an AI model can "remember" in a single conversation. When exceeded, early messages get forgotten.

Dependency - A library/package your project relies on. Helpful, but each dependency is also risk.

Diff - A view of what changed between two versions of code.

Docker - A way to package software so it runs the same everywhere. Great for 'it works on my machine' problems.

Environment variable - A configuration value stored outside the code (often used for secrets).

Git - A distributed version control system[4][5].

GitHub - A hosting service for Git repositories plus collaboration and automation tools[7][8].

Hallucination - When an AI invents plausible-sounding but incorrect information (code, function names, APIs that don't exist).

Issue - A tracked task/bug/request in a repo.

Lint - Automated checks that enforce code style and catch simple bugs.

PR (Pull request) - A proposed change set that can be reviewed before merging.

Repo (repository) - The folder (plus history) that contains your project.

Runtime - The environment that executes your code (Python, Node.js, etc.).

Secret - Credentials or keys that must not be committed or pasted into chats.

Test - Executable checks that verify behavior. Your best defense against confident mistakes.

VS Code - A code editor used to read/edit code and run/debug tasks[1].

FACTS AND SOURCES

This book avoids exact pricing wherever it might change quickly. When a number is included, it's tied to an official source and should be verified before you budget.

1. Visual Studio Code FAQ (description of VS Code, licensing notes, and 'Is VS Code free?').
<https://code.visualstudio.com/Docs/supporting/FAQ>
2. Visual Studio Code license (Microsoft product license for the distribution).
<https://code.visualstudio.com/license>
3. 'Differences between the repository and Visual Studio Code' (explains Code - OSS vs Microsoft distribution).
<https://github.com/microsoft/vscode/wiki/Differences-between-the-repository-and-Visual-Studio-Code>
4. Git home page (description: free/open source DVCS).
<https://git-scm.com/>

5. Git 'About' page (license: GPLv2).
<https://git-scm.com/about>
 6. GitHub Docs: About Git (distributed model explanation).
<https://docs.github.com/en/get-started/using-git/about-git>
 7. GitHub Docs: GitHub's plans (free plans overview).
<https://docs.github.com/en/get-started/learning-about-github/githubs-plans>
 8. GitHub pricing page (plan comparison; pricing and features can change).
<https://github.com/pricing>
 9. GitHub Copilot plans & pricing (example: Copilot Pro listed at \$10/month).
<https://github.com/features/copilot/plans>
 10. Docker pricing page (plan overview; licensing can change).
<https://www.docker.com/pricing/>
 11. Docker Personal page (free use cases and limits).
<https://www.docker.com/products/personal/>
 12. Python license (PSF license text).
<https://docs.python.org/3/license.html>
 13. Node.js license (MIT license text in repo).
<https://github.com/nodejs/node/blob/main/LICENSE>
-

References

This revised manuscript incorporates feedback from initial review, including:

- Removal of role descriptions from prompts (already in AI system prompts)
- Reduced mission count from 81 to 15 focused, high-quality missions
- Clear linkage between missions and 4-week training plan
- Added Git workflow throughout missions and prompts
- Expanded content on:
 - Software development process (requirements, testing, roadmapping)
 - Context management ([copilot-instructions.md](#), custom instructions)
 - Context rot and chat management strategies
 - Tool setup with specific instructions
 - AI as "senior developer with no context" analogy
- Improved targeting for non-technical audience while adding depth