



A FIELD GUIDE FROM VIBE CODER PLAYGROUND

Vibe Coder Playbook

Building Real Products with AI

Eleven practices taken from one real build,
each with the numbers that earned it.

6,700+
commits

2,272
agent runs measured

0 for 3
self-graded work that held

55
fixes tried and retired

Every practice here was used on a real product.

Vibe Coder Playground was built by one owner who is not a career engineer, directing AI coding agents. That makes it a fair test case for anyone learning to vibe code: the same tools, the same temptations, and the same kinds of mistakes. The difference is that the build kept records. Every claim of finished work, every test run, and every rule that did not hold was written down, so the practices below come with the measurement that earned them.

Each practice has three parts: **why it matters**, **what happened on this build** (in italics), and **what to do**. You do not need to adopt all of them on day one. Start with Practices 1 and 2. They are cheap, and they catch the most expensive mistakes.

A NOTE ON WHAT IS LEFT OUT

This guide only recommends what held up. Where the build fell short of accepted best practice, that practice is not recommended here just because this build did it. A short list of general standards this guide leans on is on the last page.

	Practice	The one-line version
01	Define done first	Write what the customer gets before you build it.
02	Never grade your own work	A fresh reviewer checks every claim of finished work.
03	Test your tests	A passing check is a claim, not proof.
04	Keep a tried-and-failed log	Write down what did not work, with the numbers.
05	Isolate every change	One task, one workspace, one commit that names its files.
06	Claim only what you can show	Unchecked is unknown, never clean.
07	Treat AI input as data	Secrets stay out of reach; text is never a command.
08	Make quality visible	One design system, checked contrast, one language.
09	Spend on purpose	Match the model to the job and stop re-reading.
10	Release in small steps	Ship a skeleton, then one feature at a time.
11	Write decisions down	Every finding names its next action and its owner.

PRACTICE 01

Define “done” before you build

An AI agent will happily build something. Whether it built *the* thing depends on whether you wrote down what the thing is. Keep one document that says, in customer words, what each feature gives a person, and whether it is shipped or not yet.

On this build that document is a features contract: every feature has an id, a one-line benefit, and a heading that says whether it is shipped. A check refuses a not-yet feature whose benefit line claims it is available. One task once carried seven deliverables; it collected seven reviews and six partial results naming five different leftovers, and sat unresolved for six days.

DO THIS

- Write the customer benefit first, then the tasks that deliver it. Tag each task with the feature it serves.
- Keep one deliverable per task. If a task has an “and” in it, it is probably two tasks.
- When part of a task is left over, give the leftover its own task. A remainder written in prose is never counted, so it never gets done.
- Put the order on paper: which features go in which release, decided once per feature.

PRACTICE 02

Never let the builder grade its own work

The agent that wrote the code is the worst judge of whether it works, for the same reason you are the worst proofreader of your own essay: it sees what it meant, not what it did. A second reviewer with a fresh context, one that did not write the change, catches what the first cannot.

The first real run of this rule went 0 for 3: three agents each reported “tests green, done,” and all three came back partial with real defects. Across the build, independent reviewers returned 158 partial and 3 rejected verdicts out of 854. A fresh reviewer is now required before any task is marked done, and the record refuses a “done” whose reviewer is also its builder.

DO THIS

- Review in a fresh session or a separate agent, never the one that built it.
- Size the review to the risk. Money, logins, privacy, and anything that can lose data get the full treatment. A behavior change gets its named test. A document or config value gets its check's pass or fail.
- Record the evidence the moment it happens, linked to the task: the command, the result, the reviewer. Evidence written down later is reconstruction, not proof.

→ Two review rounds is plenty. A third only for money, credentials, or data you cannot get back.

PRACTICE 03

Test your tests

A green test suite tells you the checks ran. It does not tell you the checks can fail. The strongest way to find out is mutation testing: break the code on purpose and see whether a test notices. If nothing turns red, that test is decoration.

*In one night, 117 defects were filed on this build. **55 of them were broken checks**, and only 29 were the product itself. One pattern: a search that finds nothing returns the whole file, so the assertion is true forever. Two security checks were passing over code that had been deleted. A packaging check reported 107 passes by reading the package's description; it never opened the installer. And a guard that passed 67 of 67 in a temporary folder failed 2 in the real folder, because the real path has a space in it.*

DO THIS

- For every new test, break the code it protects and confirm the test fails. Keep the count of breaks and catches.
- Test the thing, not the description of the thing. Open the real file, hit the real page, run the real installer.
- Run your checks where the product actually runs, with the real paths and settings.
- When the same kind of bug keeps coming back, fix the whole class once instead of patching each new example.

WHY THIS IS NOT JUST THIS BUILD

Google runs mutation testing on code changes as part of review for thousands of engineers, because coverage alone is misleading: a line can be run by a test without its result ever being checked. (Petrović et al., *Practical Mutation Testing at Scale*.)

PRACTICE 04

Keep a tried-and-failed log

Most process ideas sound good. Many do not hold. If you do not write down what you tried and how it went, you will try it again, and so will every AI session you start, because none of them remember.

This build keeps a log of 55 fixes that were tried, measured, and replaced. The most useful entry: a rule written three times in the instructions (“never end a turn with nothing running”) still failed in 342 of 386 cases. Writing a rule does not make it happen. What worked was a scheduled job that did the thing automatically.

DO THIS

- Before you propose a process change, read the log. A matching idea needs a reason the old measurement no longer applies.
- For each entry, write what was tried, what happened with a number, and what replaced it.
- When you write a rule, ask what would refuse a violation. If nothing would, the rule is a wish. Build the check, or say plainly it is unenforced.
- Keep your standing instructions short. This build cut its instruction file from 812 lines to about 220 and moved the history to a separate document, because every agent read the whole thing on every turn.

PRACTICE 05

Isolate every change

When several AI sessions work on one project, the most common way to lose work is not a bad change. It is two sessions touching the same files at the same time. Give every task its own workspace, and make every save say exactly which files it contains.

Every data-loss incident on this build traced back to a shared folder, a shared “stash,” or a forced cleanup, never to two clean merges. A forced cleanup destroyed 27 unsaved files belonging to another session. A test that restored files after itself wiped 256 lines of a neighbor's work. A side branch left to merge “later” fell 1,011 commits behind.

DO THIS

- Give each task its own git worktree: a separate folder on its own branch.
- Commit by naming the files (`git commit -- path/one path/two`), never “everything that changed.”
- Merge finished work back to the main branch the same day. Never let a long-lived side branch drift.

- Never use a shortcut that is shared by every session, like `git stash`. Use a second worktree as your “before” copy.
- Ship from an allow-list. Only files you named go public. This build shipped its own test files and a half-built scratch page before switching from “everything except” to “only these.”

PRACTICE 06

Claim only what you can show

The fastest way to lose a customer's trust is a claim that turns out not to be true. The promise that holds is not “no problems.” It is “confidence no wider than the evidence.”

*This build's audit tools follow four rules. Measured beats judged: whatever can be counted is counted, not estimated by a model. A category that was not read is **ungraded**, never clean. Every result expires. And a passing gate says “safe enough to ship,” while advice on making it great stays separate and never becomes a badge. Three completions were withdrawn when the last step turned out to be an account only the owner could buy: the feature did not exist for a customer, so the record could not say it did.*

DO THIS

- Mark unknowns as unknown. A check that could not run is a failure to report, not a pass.
- Put a date on every result, and treat an old result as stale.
- Do not call something done until a customer could use it. “Answered” is not “built.”
- Check your public claims the same way you check your code. This build fails its site build when a number on the page no longer matches the data behind it.

PRACTICE 07

Treat AI input as data, never as commands

An AI tool reads web pages, files, and messages. Any of them can contain text written to look like instructions. The model cannot always tell the difference, which is why prompt injection is first on OWASP's list of risks for AI applications. The defense is structural: limit what the AI can reach, and treat what it reads as data.

Apps built inside this product get an allowlisted set of settings and never the workbench's own secrets. Secrets live in the operating system's keychain, and a database backup never contains them. The audit features treat the code they read as untrusted input.

DO THIS

- Give each tool and each app the smallest set of keys it needs, and never your own.
- Keep secrets out of files, logs, backups, and URLs.

- Keep release switches separate from paid access. A switch that hides an unfinished feature must never be able to grant a paid one. Martin Fowler's guide to feature toggles recommends managing these two kinds separately.
- Publish a way for people to report a security problem privately, and say how fast you will answer.

PRACTICE 08

Make quality something people can see

Customers judge quality by what they see before they judge it by what they cannot. Keep one source for colors, type, and spacing, and check it the same way you check code.

This site keeps its design in one set of tokens. Every page uses them instead of hand-picked colors. A contrast gate reads every rule and fails any text below the WCAG AA ratio. The last fix moved a purple from 2.60:1 to 3.01:1. Motion turns off for visitors who ask for reduced motion, and a gate scrolls every page with a real mouse wheel to prove every panel can be reached.

DO THIS

- Keep one design system and never hand-style a single page.
- Check contrast automatically: 4.5:1 for normal text, 3:1 for large text and interface parts (WCAG 2.2 AA). Do not round up to pass.
- Respect reduced-motion settings, and test with real input, not just scripted scrolling.
- Pick one language and spelling and hold every page to it. Declare it in the page, and check it.

PRACTICE 09

Spend on purpose

AI work is billed by how much the model reads and writes. Long conversations that re-read themselves, and top-tier models doing routine chores, are where budgets disappear.

One measurement on this build: the session coordinating the agents was 64% of a \$10,237 bill, and 95 cents of every coordinator dollar went to re-reading its own conversation while it waited. Checking “is it done yet?” on a timer was the cause. Now the coordinator waits for a completion signal, builders run on the strongest model, reviewers on a mid-tier one, and record-keeping on the smallest.

DO THIS

- Match the model to the job: build with the strongest, review with a capable mid-tier model, file with the smallest.
- Wait for a signal instead of checking on a timer.

- Keep reports short. Every word in a report is something you pay to read and the next session pays to reload.
- When a conversation is finished and written down, clear it. When the next task builds on it, compact it.

PRACTICE 10

Release in small steps

Big releases hide which change broke what. Small, steady releases make every problem easy to trace and every rollback small. Industry research on software delivery (DORA) links small batches and merging to the main line at least daily with better speed and stability.

This product's beta ships as a skeleton, the basic tools every build needs, with every other feature already wired in but switched off. Features turn on one rung at a time from a public release ladder. After the beta, the rule is one new feature per release. In the owner's words: broken features are worse than not having them.

DO THIS

- Ship the smallest useful version first, with places ready for what comes next so nothing has to break later.
- Hide unfinished features behind a switch you can flip without a new build, and record who flipped it and when.
- Give every switch a removal date when you create it. Old switches pile up and slow everything down.
- After launch, add one feature at a time so you can see what each one does.

Write decisions down, and give every finding a next step

A decision you do not write down gets made again, often differently. A problem reported with no next step is homework for whoever reads it.

This build keeps a decisions log that every session can search. Each owner question and answer is recorded with its date. The rule for reports: every finding says what happens next and whose job it is. Either it is filed (with its id), or it is handed to someone (with the message ready to send), or there is nothing to do (said in one clause). A starred item is the one signal that comes from the owner, and it goes first.

DO THIS

- Record each decision with the date, the options, and the reason.
- End every report with the next action and its owner.
- Keep one list of outstanding work and one history of finished work. Never mix them.
- Let the owner's priorities be explicit, like a star, so nobody has to guess.

The checklist

Before you call anything finished, run down this list.

- The customer benefit is written, and this task delivers one thing toward it.
- Someone other than the builder reviewed it, in a fresh session.
- The test that proves it fails when the code is broken on purpose.
- It was checked where it actually runs, with real paths and real input.
- The evidence is recorded against the task: command, result, reviewer, date.
- It was built in its own workspace and committed by naming its files.
- It is merged into the main branch today, not “later.”
- Only files on the allow-list go public.
- Nothing on the page claims more than the evidence shows. Unknowns say unknown.
- No secret is in a file, log, backup, or URL, and AI-read text is treated as data.
- Colors come from the design system, contrast passes AA, and reduced motion is respected.
- Any unfinished part has its own task, and any decision made along the way is written down.

Standards and sources this guide leans on

- DORA, *Capabilities: Trunk-based development* and *Working in small batches* (dora.dev).
- G. Petrović, M. Ivanković et al., *Practical Mutation Testing at Scale: A view from Google*, IEEE TSE (2021).
- OWASP, *Top 10 for Large Language Model Applications*, 2025 edition (LLMo1: Prompt Injection).
- W3C, *WCAG 2.2 Understanding 1.4.3 Contrast (Minimum)* and *1.4.11 Non-text Contrast*.
- M. Fowler and P. Hodgson, *Feature Toggles (aka Feature Flags)* (martinfowler.com).
- Anthropic, *Best practices for Claude Code* (code.claude.com/docs).
- All other figures come from this build's own records, measured August and September 2026.

Everything in this playbook, already set up for you

Save Tokens, Save Time, Build Quality

Doing all eleven practices by hand takes discipline and a lot of files. Vibe Coder Playground builds the backbone of this playbook, the roadmap, into every app you start with it. When you add an app, tick **How the work gets managed** and your project comes with three plain files any Claude Code session reads before it does anything else.

Save Tokens

A fresh session reads docs/ROADMAP.md instead of rediscovering your project, so you stop paying for it to relearn what it once knew.

Finished work moves to an append-only history, so “we already did that” is something a session checks, not something it rebuilds.

Small fixes like copy, colors, and spacing you can make by hand, without spending a credit on a regeneration.

Save Time

One checkbox writes the roadmap, the history, and the rules into your project. Nothing to set up by hand.

A new session needs no briefing. It reads the rules, takes the next open item, and continues where the last one stopped.

Several sessions can work at once: agents hand back results and one step applies them, so nobody overwrites anybody.

Build Quality

An item is marked done only after a test has run and passed, and a separate check confirms it, not the agent that wrote it (Practice 02).

Every result is recorded as it happens, linked to its task (Practices 02 and 06).

Your decisions and your list of what is left live in the project, where the next session can read them (Practice 11).

Plainly: the roadmap engine runs on Python 3, and the setup tells you so before you tick the box. It keeps your plan and your proof in order; it does not run your project unattended.

FREE, AND YOURS TO USE TODAY

Vibe Coder 101	a free course from first prompt to a shipped app
Coding Guide	articles on building and shipping AI code
Pre-built code blocks	audited building blocks, free to copy
Free tools	audit tools that run in your browser, with no account and no credits
Stuck?	describe where you are stuck and get pointed to the fix

Get the Playground free, then start with the resources above.

vibecoderplayground.ai · macOS first; Windows and Linux follow.

Download free →