AIVORIZE

Vibe Coder Kit

Free Edition — the real core loop, two full workflow guides, five production-ready prompts, and two project templates. No account, no signup, no catch.

VIBE_CODER.md — Aivorize Vibe Coder Playbook (Free Edition)

> How to use AI to turn an idea into a real, tested, launched product. Not “how to get AI to write code.” A full loop from idea to revenue.

Before you dive in

This isn’t a trimmed-down teaser. Everything in this free download is the real thing — the exact core loop, workflows, prompts, and templates we use ourselves. Read it, use it, and it should genuinely change how your next project goes.

What this free edition covers: the core loop, how to prompt and plan well, five essential prompts, and two templates to define and launch a project.

What it doesn’t cover, because that’s a different, deeper problem: what to do when the AI’s fix doesn’t actually fix it, whether what you built is safe to put in front of strangers, which host won’t quietly break your terms of service, and how to get your first real user once it’s live. That’s what Pro is for — more on that at the end.

If you only read one section right now, read **#1 (Start Here)** and **#8 (Verification)**.

---

1. Start Here

The basic loop:

```
IDEA → RESEARCH → SPEC → PLAN → BUILD → TEST → REVIEW → DEPLOY → GET USERS → ITERATE
```

Rules that make this whole file work:

- **Plan before you build.** Don’t ask the AI to build the whole app in one prompt.
- **Verify everything.** “AI wrote it” is not the same as “it works.”
- **Keep context lean.** Only create a file if it has a job (see #3).

---

2. Idea → Product Spec

Before writing any code, answer:

- What am I building, and who is it for?
- What problem does it solve, and why would someone use it?
- What’s the smallest MVP?
- What should NOT be built (v1 non-goals)?
- What does success look like?

Turn the answers into `PRODUCT_SPEC.md`:

```
Vision · Target user · Problem · Solution
Features · User flows · MVP · Non-goals · Success criteria
```

---

3. Context Files (only create what earns its keep)

File	When you need it
`README.md`	Always — what the project is, how to run it
`PRODUCT_SPEC.md`	Any product with real users
`CLAUDE.md` / `AGENTS.md`	Project has conventions/architecture an AI agent should follow
`ARCHITECTURE.md`	More than a couple of moving pieces (frontend+backend+db)
`TASKS.md`	You’re tracking work across sessions

Rule: every doc file must have a job. If nothing reads it, delete it. Bloated instruction files hurt agent performance and cost more tokens — don’t create 15 markdown files for a landing page.

---

4. How to Write AI Prompts

Bad: “Make me a SaaS dashboard.”

Better — give the AI:

- Role / context
- Objective
- Existing files it should look at
- Requirements + constraints
- Acceptance criteria
- Expected output format

Golden rule: never ask the AI to build the entire app in one prompt. Break it into stages.

---

5. The Planning Prompt (use before every non-trivial task)

```
Don't code yet. Analyze the project, inspect the repo,
identify risks, propose an architecture/approach,
and write an implementation plan.
```

Then: **review the plan → approve it → implement.**

---

6. Build in Small Tasks

```
1. Set up project
2. Build navbar
3. Build authentication
4. Build dashboard
5. Connect database
6. Test
7. Deploy
```

Never: “Build my entire app.”

---

7. The AI Debugging System

Don’t say: “IT DOESN’T WORK FIX IT”

Instead:

```
WHAT I EXPECTED:
WHAT ACTUALLY HAPPENED:
ERROR:
WHAT I CHANGED:
RELEVANT FILE:
```

Then ask the AI to: diagnose → explain the cause → propose the *smallest* fix → implement → verify.

---

8. Verification (the part people skip)

```
AI writes → you run it → test it → inspect it → AI fixes → test again
```

Checklist before you call anything “done”:

- ☐ Build passes
- ☐ Lint / type check passes
- ☐ Tests pass
- ☐ Manual test (happy path)
- ☐ Edge cases + error states checked
- ☐ Mobile + browser checked

---

9. Security Checklist (before every launch)

- ☐ No secrets in frontend code
- ☐ `.env` is gitignored
- ☐ API keys protected server-side
- ☐ Authentication checked
- ☐ Authorization checked (not just “logged in”, but “allowed to do this”)
- ☐ Database rules / row-level permissions checked
- ☐ User input validated
- ☐ Dependencies reviewed
- ☐ Error messages don’t leak internals
- ☐ Production environment variables checked

---

10. Git & Backups

```
Working → Commit → AI change → Test → Keep / revert
```

Commit before any risky AI change. That’s your undo button.

---

11. Architecture, Simply

```
Frontend · Backend · Database · API · Auth · Storage · Hosting · Domain
```

Static landing page? You probably just need **frontend + hosting**. No database, no auth. Don’t let the AI turn a simple project into a giant architecture.

---

12. Choosing Technology

Need	Stack
Simple site	HTML/CSS/JS → Netlify/Vercel
Interactive site	React/Vite → Vercel/Netlify
Full SaaS	Next.js → database → auth → hosting
Payments	Add a payment provider only when needed
AI features	Add the API only when it’s actually the core value

Principle: choose the simplest stack that solves the problem.

13–15. Design, Responsive, Accessibility

Don’t ask for “a beautiful modern UI” — give real constraints (spacing, type scale, colors, states).

Cover explicitly: spacing/typography/color hierarchy, empty/loading/error states, forms, mobile nav.

Test at: mobile / tablet / desktop / large desktop — nav, buttons, forms, tables, overflow, touch targets.

Accessibility basics: semantic HTML, keyboard nav, labels, alt text, contrast, focus states.

---

16. Deployment

```
Local → GitHub → Vercel/Netlify → env vars → Production → Custom domain
```

Production checklist: build succeeds · env vars set · HTTPS · mobile checked · forms tested · analytics installed · 404 page · favicon · metadata · OG image.

Full sourced 2026 comparison (Vercel vs Netlify vs Cloudflare Pages, including the commercial-use gotcha on Vercel’s free tier): `guides/deployment.md` (Pro)

17. Launching

Building ≠ launching. Before you announce anything, have: landing page, headline, screenshots/demo, pricing, FAQ, contact, privacy, terms, analytics, payment, support.

---

18. Getting Your First User

Reddit, X, GitHub, Product Hunt, Discord, Indie Hackers, niche communities, direct outreach, content.

Don’t spam. Show something useful first.

---

19. $0 → $1K

```
Tiny MVP → first user → first $1 → improve → $10 → repeat → $100 → build distribution → $1,000 → reinvest
```

---

20. Validation (before you spend months building)

- Who needs this, and what do they use now?
- What sucks about that, and would they pay to fix it?
- Can I reach them? Can I MVP this in days, not months?

---

21. Pricing

Free · one-time · subscription · usage-based · lifetime · credits.

Price on value, not hours spent coding.

Full framework (a model-per-scenario table, the “stranger test,” and a 10-minute worksheet): `guides/monetization.md` and `templates/PRICING_WORKSHEET.md` (Pro).

---

22. Digital Products (good for $0-budget builders)

UI kits, templates, boilerplates, prompt packs, scripts, plugins, components, starter projects, mini tools.

Build once, sell repeatedly.

---

23. Distribution Loop

```
Build something → show it → teach how it works → give something free
→ get attention → send people to product → collect feedback → build better → show again
```

---

24. SEO

Keyword research, landing pages, useful free pages, docs, comparison pages, metadata, sitemap, Search Console. Not just “add keywords.”

Google’s own May 2026 guidance says AEO/GEO for AI search is *not* a separate discipline from this — full breakdown, sourced: `guides/seo.md` (Pro)

---

25. Analytics — find the bottleneck

```
Visitors → landing page → signup/download → checkout → purchase → retention
```

Set up Google Search Console, Bing Webmaster Tools, and Google Analytics 4 — what each does and exactly how to set them up:

`guides/search-tools.md` (Pro)

26–27. AI Model Strategy & Cost

- Don't marry one model — different models are better at planning vs. coding vs. UI vs. review.
- Don't send the whole repo every prompt; keep context concise.
- Use smaller/free models for simple tasks, stronger ones for hard reasoning.
- Reuse `PRODUCT_SPEC.md` / `ARCHITECTURE.md` instead of re-explaining the project each time.
- Full decision framework: `guides/model-selection.md` (Pro)
- Which actual tool/product to use (Cursor, Claude Code, Lovable, Bolt, etc.), with sources: `guides/TOOLS.md` (Pro)

---

28. Common Mistakes

Build everything in one prompt · no plan · no Git · no testing · trusting AI blindly · exposing API keys · overengineering · unnecessary dependencies · constantly changing architecture · fixing symptoms not causes · no mobile testing · no error handling · no backups · no analytics · building without users · spending money before validation.

---

29. "AI Didn't Do What I Wanted" Protocol

```
STOP → read what it changed → identify the exact failure
→ give concrete evidence → ask for diagnosis → fix the smallest thing → test
```

Never just "regenerate everything."

---

30. Project Memory

At the end of a session, update (only if useful): `TASKS.md`, `CHANGELOG.md`, `DECISIONS.md`.

Rule: **every documentation file must have a job — delete it if it doesn't.**

---

31. The Aivorize Vibe Coder Workflow

```
01 IDEA         06 DESIGN      11 REVIEW
02 RESEARCH     07 BUILD       12 DEPLOY
03 VALIDATE     08 TEST        13 LAUNCH
04 SPEC         09 SECURITY    14 DISTRIBUTE
05 PLAN         10 —           15 ITERATE
```

Suggested Repo Layout

(This is the full Vibe Coder Kit — Pro structure. This free download includes `VIBE_CODER.md` plus a starter set of `prompts/` and `checklists/` — see `README.md` in this download for exactly what's included free vs. in Pro.)

```
vibe-coder/
├── VIBE_CODER.md          ← start here
├── guides/
│   ├── prompting.md
│   ├── planning.md
│   ├── debugging.md
│   ├── testing.md
│   ├── security.md
│   ├── ui-ux.md
│   ├── deployment.md
│   ├── git.md
│   ├── seo.md
│   ├── analytics.md
│   ├── monetization.md
│   └── distribution.md
├── templates/
│   ├── PRODUCT_SPEC.md
```

```
│   ├── TASKS.md
│   ├── ARCHITECTURE.md
│   ├── DECISIONS.md
│   ├── LAUNCH_CHECKLIST.md
│   └── PRICING_WORKSHEET.md
├── prompts/
│   ├── research.md
│   ├── planning.md
│   ├── build.md
│   ├── debug.md
│   ├── review.md
│   ├── security.md
│   └── launch.md
└── examples/
```

`VIBE_CODER.md` stays short and links out to `guides/`, `templates/`, and `prompts/` — it never becomes a 100-page wall of instructions.

One more thing

If this got you further than a free download usually does — that was the point. You now have the loop, the prompting/planning workflows, and two templates to actually start a project properly.

Here's where people usually get stuck next, and it's exactly what Pro picks up:

- The AI's "fix" doesn't fix it, and you're not sure if it's guessing
- You're about to launch and don't know if you've left something exposed
- You picked a host without knowing its free tier quietly bans what you're building
- It's live, and now — how does anyone actually find it?

Vibe Coder Kit — Pro is $9, one time, lifetime access. See the closing page for exactly what's in it.

PART II

Workflow Guides

Two foundational guides: how to write prompts that work the first time, and how to plan before you build.

Guide: Prompting (the part that actually moves the needle)

Most “prompting tips” content is generic filler. This isn’t that. This is the difference between a prompt that gets you working code in one pass and one that gets you three rounds of “no, not like that.”

The real reason vague prompts fail

When you say “make me a SaaS dashboard,” the AI has to *invent* answers to 20 questions you didn’t ask: what data, what layout, what stack, what auth, what happens with zero data, what happens on error. It picks defaults. Those defaults are usually wrong for your actual project. Every wrong default is a round trip you didn’t need.

A good prompt isn’t “more polite” or “more detailed” for its own sake — it’s one that pre-answers the questions the AI would otherwise have to guess at.

Before / after (a real example, not a toy one)

Before: > Add a way for users to upload a profile picture.

This looks reasonable. It will still go wrong, because the AI doesn’t know: where files are stored, size limits, what happens to the old picture, what file types are allowed, or whether this needs to work on mobile Safari (which has real quirks with file inputs).

After:

```
Context: Next.js 14 app, Supabase for auth + storage. User table has an
`avatar_url` column (nullable text). Existing upload pattern is in
/lib/storage.ts (uploadFile function) — reuse it, don't write a new one.

Objective: Let a logged-in user upload a profile picture from their
/settings page.

Requirements:
- Accept jpg/png/webp only, max 2MB
- Show a preview before upload
- On successful upload: delete the old avatar file (if one exists),
  update avatar_url, show a success toast
- On failure (wrong type, too big, upload error): show a specific
  error message, don't silently fail

Constraints: Don't add a new npm package for image handling — browser
File API is enough for this. Don't touch the storage bucket's public
access rules.
```

```
Acceptance criteria: I can upload a picture, see it update immediately,
re-upload to replace it, and get a clear error if I try a 10MB file.
```

That second prompt is longer to write. It's also very likely to produce correct code in one shot, instead of three rounds of you saying "no, also handle X" after each attempt. Five minutes writing the prompt saves twenty minutes of back-and-forth.

The prompt-sizing rule

- **Micro** (one function, one bug, one tweak) — a sentence or two is fine.
- **Task** (one feature, a handful of files) — use the full template above.
- **Anything bigger** — stop. Use the planning prompt first (see `guides/planning.md`). If a prompt would touch more than ~5 files or introduce a new concept to the codebase (new auth method, new data model, new external service), it's not a Task anymore — it's a Plan.

Three phrases that quietly cause damage

- **"Make it better"** — the AI will pick what "better" means. Usually not what you meant. Say what dimension: faster, simpler, more readable, fewer dependencies.
- **"Just get it working"** — invites shortcuts (hardcoded values, skipped validation, `any` types) that become tomorrow's bug report. If you actually want a rough spike, say so explicitly and plan to redo it properly.
- **"Add whatever you think is needed"** — this is how projects end up with three different date-formatting libraries. Be the one deciding what's needed.

A prompt template you can reuse verbatim

```
Context: [stack, and which existing files/patterns are relevant]
Objective: [the one thing this should accomplish]
Requirements: [what "correct" looks like — be specific about edge cases]
Constraints: [what NOT to touch, add, or install]
Acceptance criteria: [the exact test that proves this is done]
```

Copy this into `prompts/build.md` if you haven't already — that's exactly what it's for.

Guide: Planning (how to stop AI from silently redesigning your project)

The actual failure mode this prevents

You ask for a feature. The AI decides — without telling you — to also add a new state management library, restructure your folder layout, or change how auth works, because it seemed "cleaner" while implementing your request. You don't notice until three features later when something breaks in a way that traces back to a decision you never approved.

Planning isn't bureaucracy. It's the one checkpoint where you catch this *before* it's baked into your codebase.

The planning prompt

```
Don't code yet. Analyze the project, inspect the repository, identify
risks, propose an approach, and write a step-by-step implementation
plan. Flag any new dependencies, any changes to existing patterns, and
any part of the request that's ambiguous. Wait for my approval before
implementing anything.
```

What separates a real plan from a fake one

A fake plan is a paragraph that just restates your request back to you ("I will add the upload feature by creating an upload function and calling it from the settings page") — that's not a plan, it's a rephrase.

A real plan tells you things you didn't already know: - Which existing files/patterns this touches, specifically - What could break as a side effect - Where a decision was made that you should weigh in on - The order of steps, and why that order (so early steps are safe to test before later, riskier ones)

If the plan you get back reads like a summary instead of a decision document, ask again: *"What in this plan is a judgment call you made? What would you flag to me if I weren't reviewing this?"*

Reviewing a plan — the 4 questions that actually matter

1. **Does this match what I asked, not what's "better" in the AI's opinion?** Sometimes a suggested improvement is genuinely good — but it should be flagged as an addition, not slipped into the plan as if it were the ask.

2. **Is there a new dependency, and do I actually need it?** A one-off date formatting need doesn't require a library. A recurring, complex need might.
3. **Can I test after each step, or only at the very end?** If it's "only at the end," ask for the plan to be broken into smaller, independently testable steps. This is the single biggest lever for catching problems early instead of late.
4. **What's the riskiest step, and is it first or last?** Risky, hard-to- reverse steps (database migrations, auth changes, deleting data) should happen after the safer groundwork is verified working — not first.

After approval

Implement one step at a time. Test between steps. Don't say "implement the whole plan" and walk away — that collapses back into the one-shot problem planning was supposed to prevent.

PART III

Prompts

Five essential copy-paste prompts to get you unstuck while building.

Prompt: Research

```
I'm considering building [idea] for [target user].

Help me research this before I build anything:
1. Who already has this problem, and how are they solving it today
   (tools, workarounds, competitors)?
2. What do existing solutions cost?
3. What do people complain about with current solutions? (if you can't
   search live, tell me what to search for and what to look for)
4. Is this problem big enough / painful enough that people would pay
   to solve it?
5. What's the smallest version of this that would still be useful?

Be honest if this looks like a weak idea — don't just validate me.
```

Prompt: Planning

```
Don't code yet.

Context: [describe the project / paste relevant files or PRODUCT_SPEC.md]
Goal: [what you want built]

Please:
1. Inspect the existing project structure and conventions
2. Identify risks or ambiguous decisions and flag them
3. Propose the simplest approach that meets the goal
4. Write a step-by-step implementation plan, where each step is
   independently testable
5. List any new dependencies and why each is needed

Wait for my approval before writing any code.
```

Prompt: Build (single task)

```
Context: [stack/framework, relevant existing files]
Objective: [the one thing this should accomplish]
```

```
Requirements: [what "correct" looks like]
Constraints: [what NOT to touch/add/install]
Acceptance criteria: [how I'll know it's done]

Implement just this. Don't refactor unrelated code. If you think
something outside this scope needs to change, tell me — don't just
change it.
```

Prompt: Debug

```
WHAT I EXPECTED:
WHAT ACTUALLY HAPPENED:
ERROR (full text/stack trace):
WHAT I CHANGED RECENTLY:
RELEVANT FILE(S):

Please:
1. Diagnose the actual root cause (not just the symptom)
2. Explain briefly why this causes the symptom I'm seeing
3. Propose the smallest possible fix
4. Implement only that fix
5. Tell me how to verify it's actually fixed
```

If this doesn't resolve it — the AI is guessing, not diagnosing. Pro's `guides/debugging.md` has the full root-cause ladder and the specific red flags that tell you when to stop and ask a sharper question.

Prompt: Launch Readiness

```
Here's my product: [describe it / link the repo or live URL]

Go through this like a pre-launch reviewer:
1. Is the landing page clear about what this is and who it's for
   within 5 seconds?
2. Is the pricing clear and easy to find?
3. Are there obvious broken states (empty, error, loading)?
4. Is there a way to contact/support the user if something breaks?
5. What's the single biggest thing missing before this is launch-ready?

Be direct — I'd rather hear it now than after launch.
```

PART IV

Templates

Copy these into your own project to define what you're building and check it before launch.

Product Spec: [Product Name]

Vision

One or two sentences: what this is and why it should exist.

Target user

Who specifically is this for? Be narrow, not “everyone.”

Problem

What problem does this solve for them, today, without your product?

Solution

How does your product solve it? Keep this short — details go in Features.

Features (v1 / MVP)

- Feature 1
- Feature 2
- Feature 3

Non-goals (explicitly NOT in v1)

- Thing you’re intentionally not building yet
- Thing you’re intentionally not building yet

User flows

1. User arrives via [source] →
2. User does [action] →
3. User gets [outcome]

Success criteria

How will you know this is working? (e.g. “10 people use it twice in a week”, “$100 revenue in 30 days”)

Launch Checklist

Product

- [ ] Core flow works start to finish, tested manually
- [ ] Empty/loading/error states handled
- [ ] Mobile checked

Security

- [ ] No secrets in client code
- [ ] Auth + authorization checked
- [ ] Input validated server-side

Page

- [ ] Landing page explains what/who/why in 5 seconds
- [ ] Pricing is clear
- [ ] FAQ covers the obvious questions
- [ ] Contact/support method visible

Technical

- [ ] Custom domain + HTTPS
- [ ] Analytics installed
- [ ] 404 page, favicon, metadata, OG image set

Go

- [ ] Payment (if applicable) tested end-to-end with a real small transaction
- [ ] Posted in 1-2 relevant communities, not spammed everywhere

PART V

Checklist

A quick verification checklist before you call anything done.

Verification Checklist (before calling anything “done”)

- ☐ Build/compile passes
- ☐ Lint + type check pass
- ☐ Tests pass
- ☐ Manual happy-path test
- ☐ At least one edge case tested
- ☐ Error states show something sensible
- ☐ Checked on mobile width

WANT MORE?

Vibe Coder Kit — Pro

If this free pack worked for you, here's where people usually get stuck next — and it's exactly what Pro picks up:

• The AI's "fix" doesn't fix it, and you're not sure if it's guessing
• You're about to launch and don't know if you've left something exposed
• You picked a host without knowing its free tier quietly bans what you're building
• It's live — now how does anyone actually find it?

$9, one time, lifetime access. Adds 12 more in-depth guides (several with sourced 2026 research), the debugging system, the security workflow, the full testing workflow, complete deployment guides, growth & distribution strategy with real scripts, 4 more templates, and worked examples.

Nine dollars is less than most people spend on lunch. This is the difference between shipping something that works once on your machine, and shipping something you'd actually put your name on.