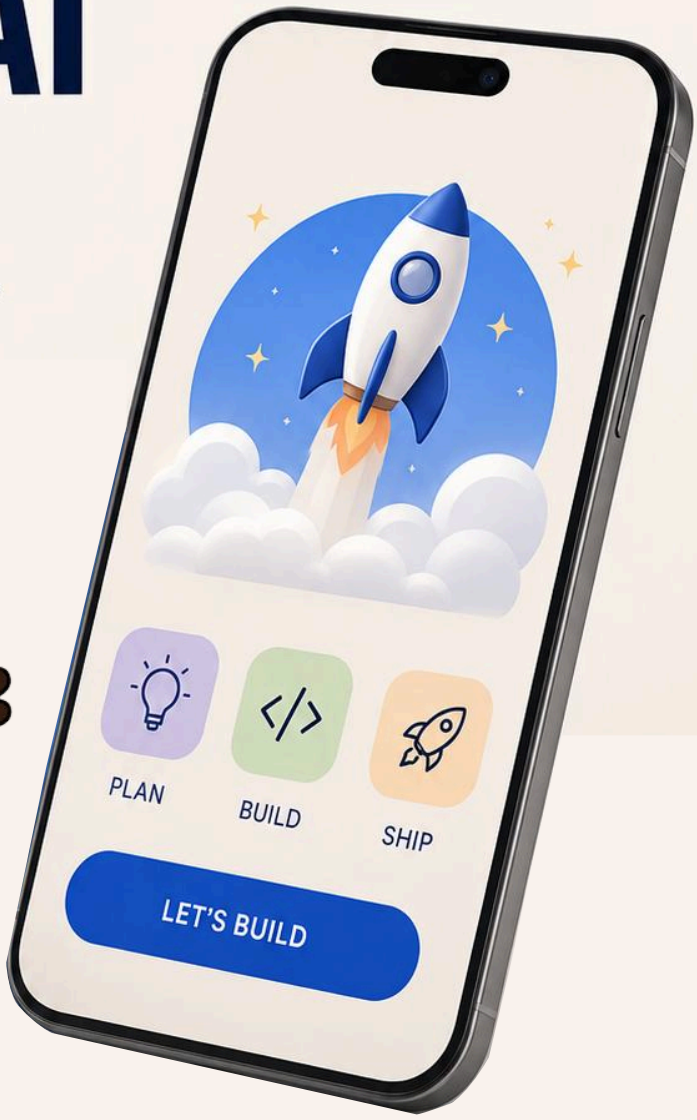
PRACTICAL VIBE CODING WITH AI

{JS} MASTERY

A complete workflow for building mobile apps with AI, from idea to release



Practical Vibe Coding for Mobile Apps

You don't need to be a senior engineer to build great mobile apps with AI.

You need a repeatable workflow.

This guide walks through the complete process of building mobile apps with AI, from idea to release.

Not with random prompts or endless planning, but with a practical system you can actually follow and reuse.

This isn't something you read once. It's a reference you come back to while building. Skim ahead, use what fits where you are, and return when you start the next lesson.

One feature at a time. One prompt at a time. One working build at a time.

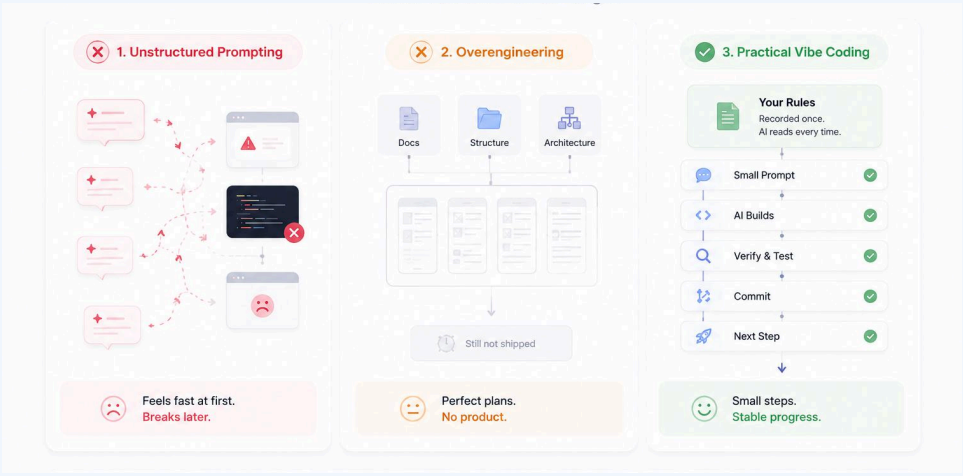
What Practical Vibe Coding is

There are two ways people build with AI, and both break eventually.

The first is unstructured prompting, or “vibe coding.” You ask for big changes, take whatever comes back, and keep going.

It feels fast at first, but the codebase slowly becomes inconsistent and starts breaking in ways the AI can’t reliably fix.

The second is the opposite. You spend days planning architecture and folder structures before writing any features. Everything stays theoretical, and the app never really gets built.



Practical Vibe Coding is what you do instead.

The AI still writes most of the code, but it works inside a clear set of rules you record once at the start of the project. Every prompt has the same structure.

Every step is small enough that you can verify it before moving to the next one. That's the entire approach.

> ***A prompt is an instruction with a defined scope.***

The curriculum

There are seven lessons in this guide. They're sequential, so follow them in order. Each one assumes the previous is done.

1. Idea and design research
2. Generate the visuals
3. Pick your stack
4. Project setup
5. Write your AGENTS.md
6. Build feature by feature
7. Test, polish, ship

Each lesson gives you a checklist, prompt templates where they help, and a "done looks like" line so you know when to move to the next one.

Lesson 1: Idea and design research

Before you write any code, you need to know what the app looks like.

The visual direction you pick here shapes your component choices, the assets you'll generate next, and the layout of every screen.

If you skip this and figure it out later, you'll end up rebuilding screens once you finally settle on a style.

Checklist

- [ ] Write the app idea in one sentence
- [ ] List the core screens. Five to eight is typical for v1: onboarding, auth, home, the primary feature screens, profile
- [ ] Open Pinterest. Search [your app type] mobile UI. Save at least 15 references.
- [ ] Open Dribbble. Same search. Save another 10.
- [ ] Open Mobbin. Pull complete real app flows that match yours.
- [ ] Pick one visual direction
- [ ] Describe that direction in one sentence (example: "playful, rounded cards, soft shadows, mascot driven").

Where to look

- Pinterest, for broad mood and inspiration
- Dribbble, for individual screen polish
- Mobbin, for full flows from real apps
- Behance, for case studies
- Real apps you already use. Screenshot the flows you like.

Boundaries

If you want to use a designer's published work as is, ask permission first or credit them.

If you're using it as a reference for your own version, you don't need to. Cloning pixel for pixel and shipping it isn't acceptable either way.

Done

A folder of references and a one sentence description of the visual direction.

Lesson 2: Generate the visuals

Generate your assets before you start building screens.

When you build a component around a real asset instead of a placeholder, your spacing, sizing, and layout come out right the first time, and you don't have to revisit them once the real image lands.

Checklist

- [ ] Generate the brand element first like mascot, hero illustration, or logo. Whatever style you lock in here becomes the reference for everything else.
- [ ] Generate every other screen illustration in the same style, using the brand element as a reference.
- [ ] Generate empty states, success states, and error states.
- [ ] Export at the resolutions your platform needs (2x and 3x for mobile).
- [ ] Save into assets/images/ with descriptive names like onboarding_illustration.png, mascot_happy.png, empty_lessons.png.

Tools

- ChatGPT Image 2 for Stylized illustrations and character work.
- Nano banana through Gemini. Strong character consistency across multiple generations.
- Midjourney for Cinematic imagery.
- Photopea, Adobe Pixelmator for Quick edits, background removal, resizing.

Prompt Template

```
Create a [style description] illustration for [feature or screen].

Reference attached.

Match the reference exactly on color palette, line weight, character proportions, lighting.

Transparent background. Square. Mobile resolution.

[attach 2 to 3 references]
```

Done

Every screen on your list has its assets generated and saved before you write the screen.

Lesson 3: Pick your stack

When you pick libraries, go with the ones that have stable, current documentation.

AI tools produce better code for established libraries than for newer or more niche ones, because there's more training data and the patterns are more consistent across versions.

The boring, well documented choice is usually the one that gives you fewer bugs.

Recommendation

Layer	Library
Framework	Expo with React Native
Language	TypeScript
Styling	NativeWind
State	Zustand
Persistence	Async Storage
Database	PostgreSQL
Auth	Clerk
Analytics	PostHog
Code review	CodeRabbit

Checklist

- [ ] Lock the framework. Use Expo if you're starting fresh.
- [ ] Pick one styling system. NativeWind gives you tailwind syntax in React Native.
- [ ] Pick one state library. Zustand keeps the API minimal.
- [ ] Pick one auth provider. Clerk handles email, social login, verification, and user management for you.
- [ ] Pick one analytics tool.
- [ ] Skip the database for v1 unless you have a real reason to add one. Hardcoded JSON or TypeScript files are usually enough for content driven apps.
- [ ] Pick one database

Prompt for stack advice

```
I am building a [type] mobile app with these features:
[list].

Solo developer. Target ship date: [date].

Recommend a Expo and React Native stack. Prioritize:
  1. AI tooling support
  2. Documentation quality
  3. Time to first feature

For each pick, state why and what alternative you
considered.

Keep total dependencies under 10 if possible.
```

Rule of Thumb

If the AI gives you three different alternatives for one decision, your requirement is probably too custom. Pick the most established option and keep going.

Lesson 4: Project setup

You'll only run through this lesson once per project. After this, you spend the rest of your build time in Lesson 6.

Checklist

- [ ] Create the project following framework instructions
- [ ] Initialize git. First commit message, init
- [ ] Add .env and .env.example. Gitignore the local one.
- [ ] Install and configure NativeWind (or any other styles). One prompt.
- [ ] Drop the assets you generated in Lesson 2 into assets/images/.
- [ ] Create constants/images.ts with centralized image imports.
- [ ] Wire up lint and typecheck commands. Confirm they pass.
- [ ] Create AGENTS.md (Lesson 5).
- [ ] Push to GitHub.

Lesson 5: Write your AGENTS.md

AGENTS.md is a single markdown file you put at the root of your project.

It records everything the AI needs to know about your project in one place: the role it's playing, the stack, the conventions, the architecture, the constraints. The AI reads it before every prompt.

Without it, the AI has to guess your conventions every time, and its guesses come out different on different days.

Those inconsistencies build up across the codebase as you ship features, and after a while it becomes hard to remember which file does what or why a pattern was chosen.

Required sections

- [ ] Role. What kind of engineer the AI is acting as. Example: "Senior Expo and React Native engineer."
- [ ] Project overview. What you're building, in three to five lines.
- [ ] Tech stack. Exact libraries, with versions if they matter.
- [ ] Development philosophy. Feature by feature, simplest version first, no overengineering.
- [ ] Architecture. Folder structure with one line of purpose per folder.
- [ ] UI rules. Match attached designs exactly. What not to approximate.
- [ ] Styling rules. NativeWind first. When to fall back to StyleSheet. Exception list (SafeAreaView, Modal, Animated views, etc).

Required sections

- [ ] State rules. When to use global state, when local, when to persist.
- [ ] TypeScript rules. Strict mode. No `any`. Simple types.
- [ ] Asset rules. Centralized image imports. Naming conventions.
- [ ] Secret rules. Never expose keys in client code. Tokens come from server routes.
- [ ] Decision rules. Ask before installing new libraries. Ask before changing existing UI.
- [ ] Final reminder. State that the file should be read before every feature.

Tip

Don't write yours from scratch. Copy the structure from a public reference (this repo's `AGENTS.md` is one) and adapt it to your project.

And you don’t have to define everything in one go. Update modify as you go and develop features.

Starter template

Copy the block below into a new `AGENTS.md` at the root of your project. There are five things to fill in, all marked with `[SQUARE_BRACKETS]`. Search for `[` in your editor to find them all.

The five fields

1. [APP_NAME]. What your app is called.
2. [ONE_LINE_DESCRIPTION]. A single sentence on what the app does.
3. [FEATURE_LIST]. Three to seven core features, as a bulleted list.
4. [EXAMPLE_COMPONENT_NAMES]. Two or three real components from your app.
5. [EXAMPLE_STATE_FIELDS]. Two or three pieces of state your app needs.

Everything else can stay as is and works for any mobile app built on the recommended stack.

```
You are an expert React Native and Expo engineer helping me build [APP_NAME].

Write clean, simple, maintainable code. Prioritize clarity over unnecessary abstraction.

Think like a senior mobile developer.

---

## Project Overview

We are building [APP_NAME], [ONE_LINE_DESCRIPTION].

The app includes:

[FEATURE_LIST]

Keep the implementation simple and readable.

---

## Tech Stack

- Expo
- React Native
- TypeScript
- Expo Router
- NativeWind
- Zustand
- AsyncStorage
- Clerk for authentication

Do not introduce new major libraries unless there is a strong reason. Ask before installing anything new.

---
```

Development Philosophy

Build feature by feature.

For every feature:

1. Read this file first.
2. Keep the implementation simple.
3. Avoid overengineering.
4. Prefer readable code over clever code.
5. Build the smallest useful version first.
6. Refactor only when repetition appears.

Decision Making

If something is unclear or could be improved, suggest a better approach. If a new library would significantly help, recommend it, explain why, and ask before adding it.

Do not install new libraries without approval.

Architecture

Use this folder structure:

```
app/
  (auth)/
  (tabs)/
components/
constants/
```

````
constants/
data/
hooks/
lib/
store/
types/
assets/
```

**app/** is for routes and screens only. Screens compose components and call hooks or stores. They should not contain large reusable UI blocks or business logic.

**components/** is for reusable UI. Create a component when it is reused in multiple places, when it makes a screen easier to read, or when it represents a clear UI concept. Examples for this app: [EXAMPLE_COMPONENT_NAMES]. Do not create components too early.

**data/** holds hardcoded content. Keep it typed.

**store/** holds Zustand stores. Examples of state to keep here: [EXAMPLE_STATE_FIELDS]. Persist with AsyncStorage when needed.

**lib/** holds external service helpers (clerk.ts, api.ts, cn.ts). Never expose secret keys here.

---

## UI Rules

For any UI task:

- Replicate the provided design exactly.
````
````

- Match layout, spacing, padding, font sizes, font hierarchy, colors, border radius, shadows, alignment, and proportions.
- Do not approximate. Do not simplify unless explicitly asked.

Styling Rules

Use NativeWind classes. Do not use StyleSheet unless it is not possible to style with className.

Use the NativeWind version installed in this project. Check package.json. Do not upgrade without approval.

Reuse class patterns through utilities in global.css.

Style Exception List

Use StyleSheet or inline styles for:

- SafeAreaView (className not supported)
- KeyboardAvoidingView (behavior props)
- Modal (visible, transparent props)
- Animated.View (animated style values)
- Dynamic styles calculated at runtime
- Platform specific styles
- Pressable or TouchableOpacity pressed states
- Shadows (different per platform)

Everywhere else, use NativeWind.

Image Rule

Use centralized image imports.

1. Check if constants/images.ts exists.
2. If not, create it.
3. Import all app images there.
4. Use them through the centralized object.

```ts
import mascot from "@/assets/images/mascot.png";

export const images = {
  mascot,
};
```

```tsx
<Image source={images.mascot} />
```

Do not import image assets directly inside screens or components.

State Management

- Zustand for global client state.
- Local state for temporary UI state.
- AsyncStorage for persistence.

TypeScript

- Strict mode.
- No `any`.
- Keep types simple and readable.

```
---

## Feature Implementation

When building a feature:

1. Read this file first.
2. Identify the files to change.
3. Keep changes focused.
4. Do not rewrite unrelated code.
5. Follow existing patterns.
6. Make sure the feature works end to end.
7. Fix lint and type errors before finishing.

---

## Secrets

- Never expose secret keys in client code.
- Use server routes for tokens, AI calls, and any external API access.

---

## Authentication

Use Clerk. Do not build custom auth.

---

## Communication

Be concise. Explain what changed and how to test it.

---

## Final Reminder
```

Before every feature:

- Read this file.
- Follow it strictly.
- Build clean, simple code.
- Replicate UI exactly when designs are provided.

Lesson 6: Build feature by feature

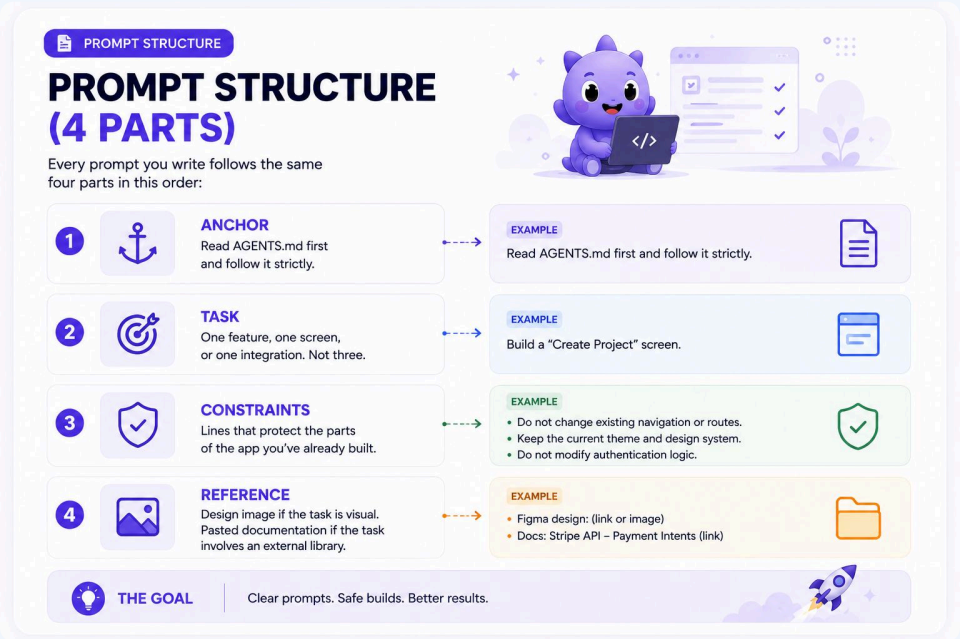
This is where you'll spend most of your time.

Every feature you add follows the same loop: same prompt structure, same verification step, same kind of commit at the end. Once you've done it a few times, you stop thinking about the loop and just run it.

Prompt structure (4 parts)

Every prompt you write follows the same four parts in this order:

1. Anchor. Read AGENTS.md first and follow it strictly.
2. Task. One feature, one screen, or one integration. Not three.
3. Constraints. Lines that protect the parts of the app you've already built.
4. Reference. Design image if the task is visual. Pasted documentation if the task involves an external library.



The build loop

For every feature,

- [ ] Write the prompt using all four parts.
- [ ] Send it.
- [ ] Read the diff.
- [ ] Run the app. Test the new feature.
- [ ] Test the previously built features. Confirm nothing regressed.
- [ ] Commit if everything works.
- [ ] If something broke, write one targeted fix prompt.

Prompt Templates

Building a UI screen

```
Read AGENTS.md first and follow it strictly.

Implement the [screen name] screen as shown in the
attached design exactly, using assets from the assets
folder.

[Optional navigation requirement, e.g.,"Add a navigation
link from the home route to open this screen."]

Do not change [thing to preserve].

[attached design image]
```

Adding application state

```
Read AGENTS.md first and follow it strictly.

Integrate [feature] state. Store [data] using Zustand
with @react-native-async-storage/async-storage.

[Behavior rule, e.g.,"If an authenticated user has no
selected language, route them to the language selection
screen."]

Preserve the existing UI exactly.

[Optional development utility, e.g.,"Add a button on the
home screen to clear async storage for testing this
flow."]
```

Integrating a library or external service

```
Read AGENTS.md first and follow it strictly.

Study the existing [related code or flow], then [task] by
following the [library] documentation provided below.

Keep the existing UI and navigation flow intact.

Do not change the screen design.

Do not expose any secrets in the client app.

[paste current documentation]
```

Fixing a specific issue

```
Read AGENTS.md first and follow it strictly.

The [thing] is [actual behavior]. It should [correct
behavior].

Do not change any other behavior or layout.
```

Constraint library

These are the lines you drop into the constraints section of any prompt. Pick whichever fit the task in front of you.

- "Do not change the screen design."
- "Preserve the existing UI exactly."
- "Keep the existing [feature] flow intact."
- "Do not expose any secrets in the client app."
- "Do not introduce new libraries without asking."
- "If a change is needed to make this work, ask first."
- "Match spacing, typography, and color exactly to the attached design."
- "Do not modify files outside [folder]."
- "Do not refactor existing code."
- "Do not add features that were not requested."

Common mistakes

These are the patterns that will quietly degrade your codebase if you let them.

- Bundling multiple features into one prompt.
- Asking for full app generation in one prompt.
- Repeating project context in every prompt instead of relying on AGENTS.md.
- Asking the AI to "improve" or "clean up" working code. You'll get back a large rewrite you didn't want.
- Accepting AI output without running it first.
- Describing UI in words when a design image is available.

Done

Each feature has one prompt, one verification, and one commit. The diff for any single commit is small enough that you can review it at a glance.

Lesson 7: Test, polish, ship

Before you submit anything to TestFlight or Google Play, walk through this checklist.

The production build on a real device behaves differently than what you've been running during development, and this lesson is where you catch the gaps before users do.

Checklist

- [ ] Run the full primary flow on a real device, not the simulator.
- [ ] Test edge cases: empty states, long input, no internet, slow internet, denied permissions.
- [ ] Test on both iOS and Android if you're shipping cross platform.
- [ ] Run lint and typecheck. Resolve all errors. No any in TypeScript.
- [ ] Code review pass with CodeRabbit or equivalent for any feature the AI wrote in full.
- [ ] Remove development utilities: test buttons, console logs, mock data, storage clearers.
- [ ] Add analytics events on the flows that matter for retention.
- [ ] Verify no secrets in the client bundle or git history. Run a secret scanner.
- [ ] Build a production binary with EAS Build. Test it on a real device before submission.

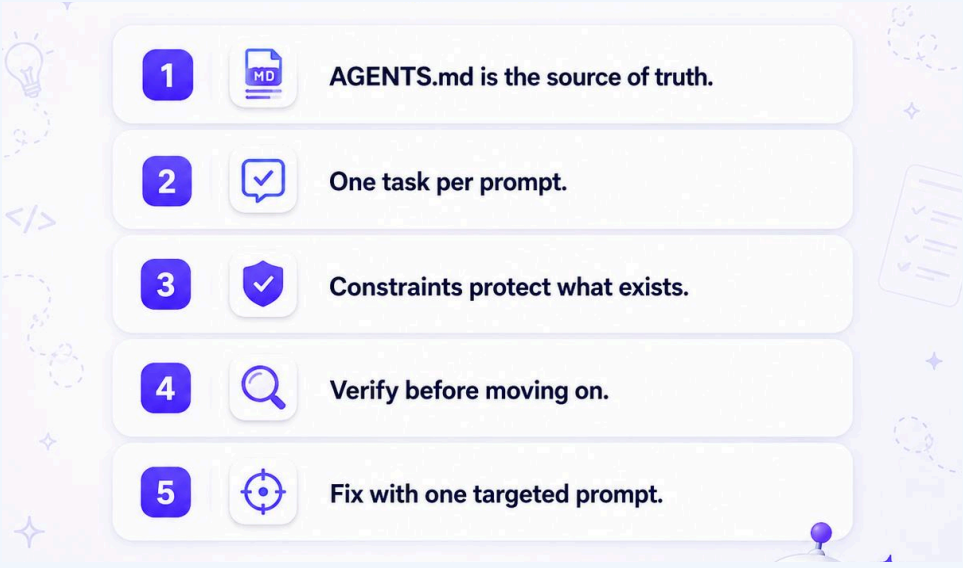
- [ ] Submit to TestFlight (iOS) and Google Play internal testing (Android). Get feedback from real users before public release.

Mental model card

Practical Vibe Coding,

1. AGENTS.md is the source of truth.
2. One task per prompt.
3. Constraints protect what's already built.
4. Verify each step before moving on.
5. When something breaks, write one targeted fix

> ***A prompt is not a random instruction. A prompt is an instruction with a defined scope.***



Resource list

Design references

- Pinterest, mood and inspiration
- Dribbble, single screen polish
- Mobbin, full app flows from real apps
- Behance, case studies
- Lapa Ninja, landing page references

Image generation

- ChatGPT Image 2, stylized illustrations
- nano banana via Gemini, character consistency
- Midjourney, cinematic imagery
- Photopea, Adobe, Pixelmator, quick edits

AI coding tools

- Claude
- Codex
- Cursor
- Gemini

One page cheatsheet

`IDEA → DESIGN → GENERATE → STACK → SETUP → AGENTS.md → BUILD → SHIP`

Every prompt:

1. "Read AGENTS.md first and follow it strictly."
2. ONE task.
3. Constraints (what not to break).
4. Reference (design image or pasted docs).

Every feature loop:

`Prompt → Read diff → Run → Verify old features → Commit → Next.`

If something breaks:

ONE targeted fix prompt.

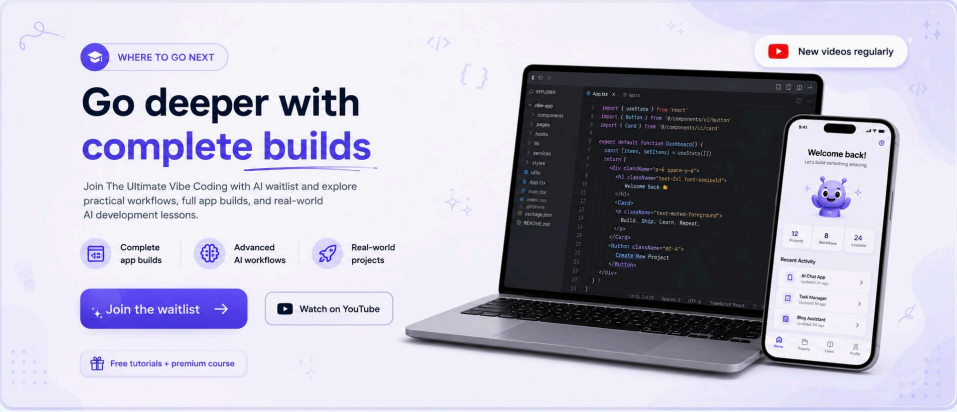
Avoid:

- Multiple features in one prompt.
- "Build me the whole app."
- Repeating context that belongs in AGENTS.md.
- Submitting a build without testing the production binary.

Always:

- One sentence describing the app before you start.
- One visual direction before generating assets.
- One commit per working feature.

These lessons work for any mobile app you decide to build. The app changes every time. The way you build it doesn't. Pick an idea and start at Lesson 1.



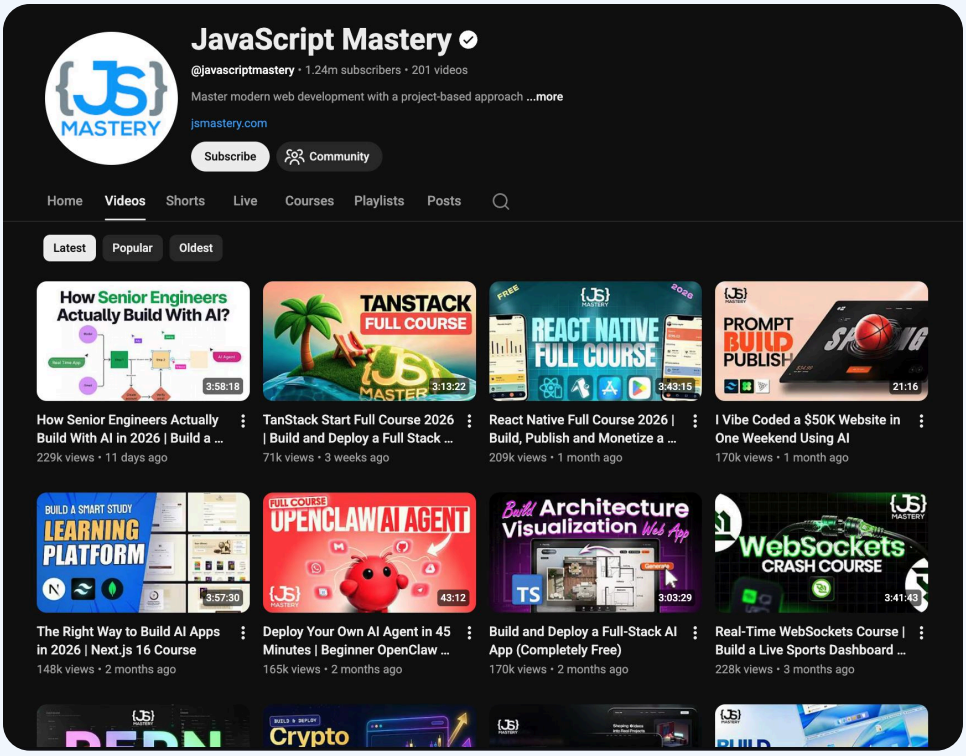
You've got the workflow. What next?

If you want to go deeper than what you learned here, I'm putting together a dedicated course called The Ultimate Vibe Coding with AI.

It covers a lot more than what's in this guide from bigger concepts, complete builds for both websites and mobile apps using the same workflow you just read, and the kind of depth a one page reference can't carry.

Join the waitlist so you don't miss it when it opens.

And if you want shorter content in the meantime, my YouTube channel is where I post new videos regularly. Different stacks, different kinds of apps, different ways to build with AI. All free, all the time.



Happy building with AI.