

A BEGINNER'S MANUAL

GROK EDITION

Vibe Coding

Build real apps and websites by describing them in plain English

Vibe Coding

Build real apps and websites by describing them in plain English

GROK EDITION

Concise edition, revised and fact-checked by Grok Bot in GrokAllManualsV1.

"Vibe coding" means building software by describing what you want to an AI and letting it write the code. You steer with words, try the result, and keep refining. People with no programming background are launching websites, internal tools, and even small businesses this way.

This manual tours today's tools, helps you choose one, and teaches a repeatable process: plan, build in small slices, test, save, and ship. It also covers the parts beginners usually miss, like databases, logins, payments, security, and what to do when the AI gets stuck.

HOW TO READ THIS MANUAL

Tip — a shortcut or good habit.

Warning — easy to get wrong.

\$ **command** — text you type, exactly as shown.

Note — useful background.

Danger — can lose data, money, or access.

Settings > **Privacy** — a menu path to click.

New words appear in **bold** the first time and are collected in the Glossary at the back.

Contents

01 What Is Vibe Coding?	6
1.1 A new way to build	6
1.2 What it's great for	6
1.3 Where it needs care	7
02 The Tool Landscape	8
2.1 Family 1: App builders in your browser	8
2.2 Family 2: AI code editors	8
2.3 Family 3: Coding agents	9
2.4 Which should you pick?	10
03 Your First App in a Browser Builder	11
3.1 Getting to know the screen	12
3.2 Visual editing	12
04 The Vibe Coding Workflow	13
4.1 Start with a one-page brief	13
4.2 Build in slices	14
4.3 Give the AI a rules file	15
05 Prompts That Work	16
5.1 The anatomy of a great prompt	16
5.2 Prompt templates	16
5.3 Use pictures	17
5.4 Ask for options before decisions	17
06 Data, Logins, and Payments	18
6.1 Front end and back end	18
6.2 Logins (authentication)	18
6.3 Payments	19
6.4 API keys and environment variables	19
07 Leveling Up: Editors and Agents	20
7.1 Signs you've outgrown a browser builder	20
7.2 Moving your project	20
7.3 Working in Cursor	21
7.4 Agents: the most hands-off option	21
08 Shipping Your App	22
8.1 Hosting	22
8.2 Your own domain name	23

8.3	Pre-launch checklist	23
09	When You Get Stuck	24
9.1	Break the loop	24
9.2	Reading error messages	25
10	Security for Vibe Coders	26
10.1	The big five	26
10.2	Ask the AI to audit itself	26
10.3	Privacy and the law	27
11	Costs and Credits	28
11.1	Make credits last	28
11.2	A typical starter budget	28
12	Three Complete Builds	29
12.1	Build 1: Portfolio site	29
12.2	Build 2: To-do app with logins	30
12.3	Build 3: Landing page with a waitlist	31
13	The Prompt Library	32
13.1	Planning	32
13.2	Building	33
13.3	Styling	33
13.4	Debugging	33
13.5	Refactoring and quality	34
13.6	Security	34
13.7	Shipping	35
13.8	Learning	35
14	Error Message Decoder	36
15	The Modern Web Stack, Explained	37
16	Frequently Asked Questions	38
17	Deploying, Step by Step	39
17.1	GitHub Pages (static HTML/CSS/JS)	39
17.2	Vercel or Netlify	39
17.3	Environment variables on the host	40
17.4	Custom domain	40
17.5	Rolling back	41
18	Databases and Security Rules, Explained	42
18.1	Five SQL statements	42
18.2	Row-level security in Supabase	43
19	Project: Build a Habit Tracker End to End with a Coding Agent	44
19.1	Phase 0: Set up the folder	44

19.2	Phase 1: Plan before code	45
19.3	Phase 2: Scaffold and first run	45
19.4	Phase 3: Features, one at a time	46
19.5	Phase 4: Review like an editor	46
19.6	Phase 5: Deploy	46
19.7	Where to take it next	47
20	Taking Payments Safely with Stripe Test Mode	48
20.1	The safe architecture	48
20.2	Step by step in test mode	49
20.3	Test the failure paths too	49
20.4	Going live checklist	50
21	Accessibility and Performance Checklist	51
21.1	Accessibility (a11y)	51
21.2	Performance	52
22	After Launch: Maintaining Your App	53
22.1	Weekly 20-minute routine	53
22.2	Monthly	54
22.3	When something breaks in production	54
22.4	Handing your app to someone else	54
23	Vibe Coding Prompt Patterns for Every Stage	55
23.1	Starting	55
23.2	Building	55
23.3	Debugging	56
23.4	Reviewing and shipping	56
24	Ten More App Ideas with Starter Prompts	57
	Quick Reference	59
	Glossary	60

What Is Vibe Coding?

IN THIS CHAPTER

- Where the term came from and what it really means
- What vibe coding is great at, and where it needs care
- The mindset that makes it work

1.1 A new way to build

In early 2025, AI researcher Andrej Karpathy described a style of programming where you “fully give in to the vibes”: you tell an AI what you want, accept its changes, run the result, and keep going by describing what to fix. He called it **vibe coding**, and the name stuck.

Today the phrase covers a whole spectrum:

- **Pure vibe coding:** you never look at the code. You judge only by what you see and click.
- **AI-assisted building:** you mostly describe, but you read plans, check changes, and learn as you go.
- **AI pair programming:** professionals using AI to write code faster, while reviewing every line.

This manual starts you at the first level and gently walks you toward the second. That’s where the best results, and the fewest disasters, happen.

1.2 What it’s great for

- **Prototypes:** turn an idea into something clickable in an afternoon.
- **Personal tools:** a habit tracker, a recipe organizer, a budget calculator.
- **Small-business sites and apps:** booking pages, menus, client portals, internal dashboards.
- **Automations and scripts:** “Every Monday, combine these spreadsheets and email me a summary.”

- **Learning:** asking the AI to explain what it built is one of the fastest ways to learn to code.

1.3 Where it needs care

SITUATION	WHY IT'S RISKY	WHAT TO DO
Handling money or payments	Bugs can cost real money	Use proven services (Stripe) and test thoroughly
Storing personal data	Leaks hurt people and break laws	Learn the security checklist in Chapter 10
Big, long-lived products	AI-written code can grow messy	Use Git, keep a plan, refactor regularly
Anything safety-critical	Mistakes are dangerous	Get a professional review

TIP

The mindset that works: **you are the product manager, the AI is your fast but literal developer.** Be clear about what you want, check its work, and break big jobs into small, testable pieces.

The Tool Landscape

There are dozens of vibe coding tools, but they fall into three families. Knowing which family a tool belongs to tells you most of what you need to know.

2.1 Family 1: App builders in your browser

Type a description on a website and watch a working app appear, with a live preview, hosting, and often a database built in. **Nothing to install.** Best for beginners and for web apps.

TOOL	MADE BY	KNOWN FOR	STARTING PRICE
Lovable	Lovable	Polished full-stack web apps; built-in backend ("Lovable Cloud") or Supabase; GitHub sync	Free plan; Pro from \$25/mo (100 credits)
Bolt	StackBlitz	Fast web and mobile app prototypes running in the browser	Free (1M tokens/mo); Pro \$25/mo
v0	Vercel	Beautiful interfaces with modern web components; one-click deploy to Vercel	Free (limited daily messages); Plus \$30/mo
Replit	Replit	"Agent" builds and hosts full apps in a cloud workspace	Core \$20/mo; Pro \$100/mo
Base44	Wix	All-in-one apps with database, login, and hosting included	Free; Starter \$16/mo (billed yearly)
Google AI Studio (Build)	Google	Gemini-powered app building with Firebase backends	Free for prototyping

2.2 Family 2: AI code editors

A code editor (a program for viewing and editing code files) with AI built into every corner: chat, autocomplete, and agents that edit many files at once. **Installed on your computer.** Best once your project outgrows a browser builder, or when you want to see and own the files.

TOOL	NOTES	STARTING PRICE
Cursor	The most popular AI editor, built on VS Code. Owned by SpaceX/xAI since 2026; plans now come bundled in “The X Subscription”	Free; Plus \$8/mo (Grok models only); Premium \$30/mo
VS Code + GitHub Copilot	Microsoft’s free editor plus GitHub’s AI	Copilot Free; Pro \$10/mo
Devin Desktop (formerly Windsurf)	Cognition’s AI editor	Free tier; paid plans
Kiro	Amazon’s editor, focused on writing specs before code	Free tier; paid plans
Zed	Very fast editor with AI agents built in	Free; paid AI plans

2.3 Family 3: Coding agents

Agents work on your files, run commands, test their own work, and keep going until a job is done. They can work in a terminal, an editor, a desktop app, or in the cloud while you’re away.

TOOL	NOTES	HOW YOU PAY
Claude Code	Anthropic’s agent: terminal, desktop app, VS Code, web (see the <i>Claude Code</i> manual)	Claude Pro \$20/mo and up
OpenAI Codex	OpenAI’s agent: app, terminal, IDE, cloud	Included in ChatGPT Plus (\$20/mo) and up
Gemini CLI	Google’s open-source terminal agent	Free tier with a Google account
Copilot coding agent	Assign a GitHub issue to Copilot; get a pull request	Copilot Pro and up
Devin	Cognition’s autonomous cloud engineer	Paid plans

NOTE

Prices change constantly in this market. Cursor’s plans changed in October 2026, for example. Treat these as a rough guide and check each tool’s pricing page before you subscribe.

2.4 Which should you pick?

IF YOU WANT TO...	START WITH
Build your first web app with zero setup	Lovable, Bolt, or Base44
Design a beautiful interface or landing page	v0
Build and host an app with a backend in one place	Replit or Lovable
Try for free with no credit card	Google AI Studio Build, or the free tiers of Lovable/Bolt/v0
Own your files and grow a serious project	Cursor or VS Code + Copilot, with Claude Code or Codex
Have AI work while you're away	Claude Code (cloud), Codex, or the Copilot coding agent

TIP

A great path for beginners: **start in a browser builder, connect it to GitHub early, then open the same project in Cursor or Claude Code** when you need more control. Nothing is lost along the way.

Your First App in a Browser Builder

The steps below use Lovable as the example, but Bolt, v0, Base44, and Replit work almost identically.

BUILD AND PUBLISH A FIRST APP

1

Sign up.

Go to lovable.dev (or your chosen tool) and create a free account.

2

Describe your app in the main box.

Be specific about *who* it's for and *what* they do:

PROMPT

Build a simple "Plant Pal" web app where I can add my houseplants (name, photo, how often to water), see which plants need water today, and tick them off when watered. Friendly, earthy green design. Works well on phones.

3

Watch it build.

The AI writes the code and a **preview** (a live, clickable version) appears beside the chat.

4

Try it like a user.

Click every button. Add a plant. Refresh the page. Does your data stay?

5

Ask for one change at a time.

PROMPT

When I refresh, my plants disappear. Save them permanently so they're still there tomorrow.

6

Connect GitHub.

Find the GitHub option in the project's settings or menu and link your account. Your code is now backed up, versioned, and portable.

7

Publish.

Click **Publish** (or **Deploy**). You'll get a public web address to share.

 WARNING

In step 5 the AI will probably add a **database** and maybe **login**. That's great, but it means your app now stores data. Read Chapter 7 before inviting real users.

3.1 Getting to know the screen

Most browser builders share the same layout:

- **Chat panel:** where you describe changes.
- **Preview:** your app running live. Many tools let you switch to a phone-sized view.
- **Code view:** the actual files. Peek in sometimes; it's how you learn.
- **Version history:** earlier versions you can restore if a change goes wrong.
- **Settings / Integrations:** GitHub, database, custom domain, payments.

3.2 Visual editing

Many builders let you click an element in the preview, say a heading or a button, and change its text, color, or size directly, or point at it and say “make this bigger.” This is quicker, and usually cheaper in credits, than describing the change in chat.

The Vibe Coding Workflow

Random prompting produces random apps. This five-stage loop is what experienced builders do.

THE LOOP

- 1 Plan.**
Write a short brief: who it's for, the main things they do, what “done” looks like.
- 2 Build one slice.**
Ask for the smallest useful piece: just the plant list, not the whole app.
- 3 Test it yourself.**
Click through it. Try to break it. Check on a phone.
- 4 Save a checkpoint.**
Commit to GitHub, or note the working version in your tool's history.
- 5 Repeat** with the next slice. When something breaks, roll back to the last good checkpoint rather than piling fixes on top.

4.1 Start with a one-page brief

Before you build, spend ten minutes writing this. Paste it as your first message, or save it in your project for the AI to read. Professionals call it a **PRD** (product requirements document).

BRIEF.MD

```
# Plant Pal – brief

## Who it's for
Busy people with 5–30 houseplants who forget to water them.

## Core features (version 1)
1. Add a plant: name, photo, watering interval in days.
2. "Today" list showing plants due for water.
3. Tap to mark watered; the next due date updates.

## Not in version 1
Social sharing, plant identification, reminders by email.

## Look and feel
Calm, earthy greens. Large tap targets. Works on phones first.

## Done means
I can add 3 plants on my phone, mark one watered,
close the browser, come back tomorrow, and see the right list.
```

 TIP

Ask the AI to help write the brief: “Interview me with five questions about my app idea, then write a one-page brief.” It’s a great way to discover what you actually want.

4.2 Build in slices

Each prompt should do **one thing** you can check:

1. “Create the page layout and a hard-coded sample list of three plants.”
2. “Add a form to add a new plant.”
3. “Save plants to the database so they persist.”
4. “Add the ‘Today’ view that shows only plants due for water.”
5. “Add sign-in so each person sees only their own plants.”

Small steps mean small, understandable mistakes.

4.3 Give the AI a rules file

Most tools read a project instructions file automatically. Put lasting rules there instead of repeating them in every chat:

TOOL	FILE
Claude Code	CLAUDE.md
Cursor	.cursor/rules/ folder (or AGENTS.md)
Codex, many others	AGENTS.md
Lovable, Bolt, v0	Project "knowledge" or "custom instructions" in settings

AGENTS.MD

- This is a phone-first web app. Test layouts at 375px wide.
- Use the existing color variables; don't invent new colors.
- Never delete user data without a confirmation step.
- Explain changes in plain English when you finish.

Prompts That Work

5.1 The anatomy of a great prompt

A strong prompt answers four questions:

1. **What** do you want? (“Add a search box above the plant list.”)
2. **Why?** (“I have 30 plants and scrolling is slow.”)
3. **How should it behave?** (“Filter as I type; ignore capital letters.”)
4. **How will you know it’s right?** (“Typing ‘fern’ shows only my two ferns.”)

PROMPT

Add a search box above the plant list. I have 30 plants and scrolling is slow. Filter the list as I type, ignoring capital letters. Typing "fern" should show only plants with fern in the name. Don't change anything else on the page.

That last line, “don’t change anything else,” saves a surprising amount of grief.

5.2 Prompt templates

STARTING AN APP

PROMPT

Build [app] for [who]. They need to [main job 1], [main job 2], and [main job 3]. Style: [3 adjectives], similar in feel to [an app you like]. Phone-friendly. Start with just [first slice].

ADDING A FEATURE

PROMPT

Add [feature] to [page]. When the user [action], it should [result].
Keep the current design. Don't change other features.

FIXING A BUG

PROMPT

Bug: when I [steps to reproduce], I expected [what should happen]
but instead [what actually happens]. Here's the error message: [paste].
Find the cause, explain it in one sentence, then fix it.

IMPROVING DESIGN

PROMPT

Make the [page] feel more [premium / playful / calm]. Improve spacing,
typography, and contrast. Keep all content and functionality the same.

UNDERSTANDING WHAT WAS BUILT

PROMPT

Explain how this app is organized, as if I'm new to coding.
What are the main files, and what does each one do?

5.3 Use pictures

Screenshots communicate more than paragraphs. Most tools accept images: paste a screenshot of a bug, a sketch on paper, or a website whose style you admire (“match the spacing and feel of this, but with my colors”).

5.4 Ask for options before decisions

PROMPT

I want users to be able to pay for a premium plan.
Give me three ways to do this, with pros and cons, before building anything.

Data, Logins, and Payments

Sooner or later your app needs to remember things, know who's using it, or take money. Here's what's going on behind the scenes, so you can make good decisions.

6.1 Front end and back end

- The **front end** is what runs in the user's browser: pages, buttons, layout.
- The **back end** is what runs on a server: storing data, checking passwords, sending emails, talking to other services.
- A **database** is where the back end keeps information, organized into **tables** (like spreadsheet tabs) of **rows** (records) and **columns** (fields).

Browser builders increasingly include a back end for you (Lovable Cloud, Base44, Replit, Bolt Cloud), or connect to **Supabase**, a popular hosted database with login and file storage built in.

6.2 Logins (authentication)

Authentication ("auth") is proving who someone is: email and password, a magic link, or "Sign in with Google." Always use your platform's built-in auth or a proven service. Never ask the AI to invent its own password system.

PROMPT

Add sign-in with email magic links and Google, using the built-in auth. Each user should only ever see and edit their own plants.

That last sentence matters enormously. See "Row-level security" in Chapter 10.

6.3 Payments

Use **Stripe** (or your builder's payments integration) for anything involving money. Stripe handles card details so they never touch your app.

- Build and test in Stripe's **test mode** first, with its test card numbers.
- Let Stripe host the checkout page; it's safer and simpler.
- Ask the AI to explain how a payment marks someone as "premium," and test what happens if a payment fails.

6.4 API keys and environment variables

An **API** (application programming interface) is how one program talks to another, for example your app asking OpenAI to write text or asking a weather service for a forecast. Services identify you with an **API key**, a long secret string.

Keys belong in **environment variables** (also called **secrets**): settings stored on the server, outside your code. Every builder has a **Secrets** or **Environment variables** panel.

⚠ DANGER

Never paste API keys directly into chat or code, and never put a secret key in front-end code. Anyone can read front-end code in their browser and use your key, and run up your bill.

Leveling Up: Editors and Agents

7.1 Signs you've outgrown a browser builder

- Each change breaks something else, and fixes cost more and more credits.
- You need a feature the builder can't do (a custom server task, an unusual library).
- You want to understand and own your code.

7.2 Moving your project

OPEN YOUR BUILDER PROJECT ON YOUR OWN COMPUTER

1 Make sure it's on GitHub.

Use the builder's GitHub sync (Chapter 3).

2 Install an editor or agent.

Cursor, VS Code with Copilot, or Claude Code (see the *Claude Code* and *GitHub* manuals in this series).

3 Clone the repository to your computer: `git clone <url>`, or **Clone repository** in your editor's welcome screen.

4 Ask the AI to get it running.

PROMPT

This project was built in Lovable. Install what it needs and start it locally, then explain how to run it myself next time.

5 Keep both in sync.

Push changes to GitHub; most builders pull them back automatically.

7.3 Working in Cursor

Cursor looks like VS Code, with an AI panel on the side.

FEATURE	WHAT IT DOES
Tab	Predicts your next edit as you type; press <code>Tab</code> to accept
Inline edit (<code>Ctrl</code> / <code>Cmd</code> + <code>K</code>)	Select code and describe a change
Agent / chat (<code>Ctrl</code> / <code>Cmd</code> + <code>L</code> or <code>I</code>)	Ask questions or have the agent make multi-file changes
Rules	Project instructions in <code>.cursor/rules/</code> or <code>AGENTS.md</code>
Cloud agents	Hand off a task to run in the cloud while you keep working

7.4 Agents: the most hands-off option

Coding agents like Claude Code and Codex can take a whole task, such as “add email reminders,” and work through it: read the code, plan, edit files, run the app, fix errors, and commit. Pair them with GitHub so every change arrives as a reviewable pull request.

TIP

A powerful combo: **plan in chat, build with an agent, review the pull request on GitHub.** You stay in charge of *what* gets built while the AI handles *how*.

Shipping Your App

8.1 Hosting

Hosting means running your app on a computer that's always online so others can use it. Your options:

OPTION	GOOD FOR
Your builder's Publish button	The easiest path; usually includes a free subdomain
Vercel	Next.js and React apps; great with v0
Netlify / Cloudflare Pages	Static sites and front-end apps
GitHub Pages	Simple static websites (see the <i>GitHub</i> manual)
Replit Deployments	Apps built in Replit

8.2 Your own domain name

Buy a domain (like `plantpal.app`) from a registrar such as Cloudflare, Namecheap, or Porkbun. Then, in your host's **Domains** settings, add it and follow the instructions to update the **DNS records**, the internet's address book that points your domain at your host. Changes can take from minutes to a few hours to work everywhere.

8.3 Pre-launch checklist

- ☐ Tested every feature on a phone and a computer
- ☐ Tried signing up as a brand-new user, in a private browser window
- ☐ Confirmed user A cannot see user B's data
- ☐ No secret keys in front-end code; secrets stored in environment variables
- ☐ Error messages are friendly, not technical
- ☐ A privacy policy page if you collect any personal data
- ☐ A backup or export of your database
- ☐ Spending limits set on every paid service (AI APIs, hosting, database)

TIP

Print this list and tick it off before every public launch, not just the first one. Ask your AI coding assistant to go through the list with you item by item and show you the evidence for each tick, such as where the secrets are stored or where the spending limit is set.

When You Get Stuck

Every vibe coder hits the same wall: the AI keeps “fixing” a bug and it keeps coming back, or each fix breaks something new. Here’s how to get unstuck.

9.1 Break the loop

1. **Stop prompting the same request.** If two attempts failed, a third identical one probably will too.
2. **Roll back** to the last version that worked (version history or Git), and try a different approach.
3. **Ask for a diagnosis, not a fix:**

PROMPT

Don't change any code yet. List the three most likely causes of this bug, and how we could test each one.

4. **Add visibility:** “Add logging so we can see what’s happening when I click Save.” Then paste what the logs say.
5. **Start a fresh chat** with a short summary. Long conversations confuse AIs.
6. **Simplify:** remove the broken feature, get back to green, rebuild it in smaller slices.
7. **Switch models or tools.** A different AI often spots what the first one missed.

9.2 Reading error messages

Error messages look scary but are just clues. Copy the *whole* message, including file names and line numbers, and paste it to the AI. To find errors in a web app, open your browser's **developer tools** (`F12`, or right-click and choose **Inspect**) and look at the **Console** tab.

NOTE

“It doesn’t work” gives the AI nothing to go on. “When I click Save on the Add Plant form, nothing happens, and the console shows `TypeError: cannot read properties of undefined`” gets it fixed.

Security for Vibe Coders

AI tools make it easy to build apps that *work* but aren't *safe*. A few habits prevent the most common, and most embarrassing, problems.

10.1 The big five

1. **Keep secrets secret.** API keys in environment variables, never in front-end code or GitHub.
2. **Lock down your database.** With Supabase and similar services, turn on **row-level security** (RLS): rules in the database itself that say “users can only read and change their own rows.” Without it, anyone who knows your app’s address may be able to read everything.
3. **Never trust the browser.** Anything checked only in the front end (prices, permissions, “is premium?”) can be faked. Important checks must happen on the server.
4. **Use proven building blocks** for login, payments, and file uploads, not home-made versions.
5. **Set spending limits.** Put budget caps and alerts on AI APIs, hosting, and databases. A bug or a bot can otherwise run up a large bill overnight.

10.2 Ask the AI to audit itself

PROMPT

Act as a security reviewer. Check this app for: exposed secrets, missing row-level security, actions that only check permissions in the browser, missing rate limits on forms and AI features, and personal data we store but don't need. List problems by severity, then fix the critical ones.

Many builders also include a built-in security scan. Run it before every launch.

10.3 Privacy and the law

If you collect names, emails, or anything personal, you have responsibilities: tell people what you collect (a privacy policy), collect only what you need, and delete data on request. Rules vary by country (for example GDPR in Europe and CCPA in California). For anything health- or finance-related, get professional advice.

Costs and Credits

Most vibe coding tools charge with **credits** (or **tokens**), units of AI work. Bigger requests and longer conversations use more.

11.1 Make credits last

- **Plan before building.** A clear brief avoids expensive rewrites.
- **Use visual edits** for small text and color changes instead of chat.
- **Batch small tweaks** into one clear message, but keep big features separate.
- **Roll back instead of re-prompting** when a change goes badly.
- **Use “chat” or “plan” modes** to discuss ideas; many tools charge less when the AI isn’t editing code.
- **Move heavy work to an agent subscription** (Claude Code, Codex, Cursor) when you’re doing lots of building. A flat monthly plan is often cheaper than pay-per-credit builders.

11.2 A typical starter budget

STAGE	SETUP	MONTHLY COST
Exploring	Free tiers of 2–3 builders	\$0
Building your first real app	One builder plan	≈ \$20–30
Growing a serious project	Builder or editor + Claude Pro or ChatGPT Plus	≈ \$40–60
Running in production	Plus hosting, database, domain	add ≈ \$10–50

Three Complete Builds

Follow these from start to finish. Each one uses a different family of tools from Chapter 2. They work with any builder (Lovable, Bolt, Replit, v0 and similar), any AI editor, or any coding agent. Menu names differ, but the prompts and steps carry over.

IN THIS CHAPTER

- Build 1: A personal portfolio site (static, free hosting)
- Build 2: A to-do app with logins and a database
- Build 3: A landing page with a waitlist form

12.1 Build 1: Portfolio site

1 Write the brief.

PROMPT

Build a single-page portfolio for a [your job] named [name]. Sections: hero with name and one-line pitch, About (3 sentences), Projects (3 cards with image, title, description, link), Contact (email link). Clean, lots of white space, one accent colour, mobile-friendly. Plain HTML, CSS and a little JavaScript, no frameworks.

2 Review it on desktop and phone sizes.

Most builders have a device-size toggle.

3 Iterate in slices.

“Make the project cards two columns on desktop, one on mobile.” Do one change per prompt.

4 Replace the placeholder content with your real text and images.

5 **Export or push to GitHub** (Vol. 2).

6 **Publish free on GitHub Pages:**

`Settings` > `Pages`, deploy from the `main` branch.

12.2 Build 2: To-do app with logins

This build adds the two hardest beginner topics, **authentication** and a **database**. It uses Supabase as the example back end because many builders integrate with it. Others work the same way.

1 **Brief.**

PROMPT

Build a to-do app. Users sign up and log in with email. Each user sees only their own tasks. Tasks have title, due date, done checkbox. Use Supabase for auth and the database. Add row-level security so users can only read and write their own rows. Explain each step you take.

2 **Connect Supabase** through the builder's integration, or paste the project URL and the **anon (public)** key when asked. Never paste the **service role** key into front-end code.

3 **Check the table** in Supabase's table editor: there should be a `tasks` table with a `user_id` column.

4 **Check row-level security (RLS).**

In Supabase, the table should show RLS **enabled**, with policies limiting access to `auth.uid() = user_id`.

5 **Test with two accounts.**

Sign up as two different users in two browsers. Neither should see the other's tasks.

6 **Ask for an audit.**

PROMPT

Audit this app's security: auth flows, RLS policies, exposed keys, input validation. List problems by severity and fix only the critical ones.

7 **Deploy** with the builder's publish button, or connect the GitHub repo to a host like Vercel or Netlify.

⚠ DANGER

The two-account test is the most important step in this book. Many vibe-coded apps leak every user's data because security rules were missing. Don't skip it.

12.3 Build 3: Landing page with a waitlist

1

Brief.

"A landing page for [product idea]: headline, three benefit bullets with icons, a screenshot placeholder, an email waitlist form, FAQ (4 questions), footer. On submit, save the email and show a thank-you message."

2

Store sign-ups.

Use a simple option: a form service, a Google Sheet through an automation (Vol. 6, n8n), or a database table.

3

Add basic spam protection.

Ask for a hidden "honeypot" field and server-side validation.

4

Add privacy basics.

Include a short privacy note under the form saying what you'll do with emails.

5

Measure.

Add privacy-friendly analytics if you want visitor counts.

6

Publish and share.

The Prompt Library

Fifty tested prompt patterns, grouped by moment. Copy them, then fill in the brackets.

IN THIS CHAPTER

- Planning, building, styling, debugging, refactoring, security, shipping, and learning prompts

13.1 Planning

PROMPT

Before writing code, ask me up to 8 questions that would most change how you build [idea]. Then propose a version-1 scope that fits in one weekend.

PROMPT

Turn this idea into a one-page brief with: who it's for, core features (max 5), not-in-v1, look and feel, and "done means". Idea: [idea]

PROMPT

Propose 3 different technical approaches for [feature], with pros, cons, and cost for each. Recommend one for a beginner and say why.

PROMPT

Break this brief into 8–12 small slices, each testable on its own, in build order.

13.2 Building

PROMPT

Implement slice [N] only: [description]. Don't touch unrelated files. When done, tell me exactly how to test it.

PROMPT

Add [feature] following the existing patterns in this project. Show me which files you'll change before changing them.

PROMPT

Create realistic sample data (10 items) so I can see the UI working.

PROMPT

Make this work on mobile screens 360px wide without horizontal scrolling.

13.3 Styling

PROMPT

Restyle this page to feel [calm / playful / premium / minimalist]. Keep the layout; change typography, spacing, and colours. Use one accent colour.

PROMPT

Here's a screenshot of a design I like [image]. Match its spacing and typography, not its content.

PROMPT

Check colour contrast for accessibility (WCAG AA) and fix any text that fails.

13.4 Debugging

PROMPT

Here is the exact error and what I did right before it: [error + steps]. Explain the cause in plain English, then propose the smallest fix.

PROMPT

This used to work before your last change. Compare before and after, and find what broke it.

PROMPT

Add temporary logging around [feature] so we can see what's happening, then tell me what to click and what to send you.

PROMPT

We've tried three fixes and none worked. Stop. List every assumption we've made, and which ones might be wrong.

13.5 Refactoring and quality

PROMPT

This file is getting long. Propose how to split it into smaller pieces without changing behaviour. Wait for my OK.

PROMPT

Find duplicated code and consolidate it. Behaviour must stay identical.

PROMPT

Write tests for [feature] that would catch the bug we just fixed.

13.6 Security

PROMPT

Scan the project for exposed secrets, API keys in front-end code, and missing access rules. Report, don't fix yet.

PROMPT

Explain who can read and write each table in the database, in plain English.

PROMPT

Add input validation and rate limiting to [form/endpoint].

13.7 Shipping

PROMPT

Prepare this for production: environment variables, error pages, loading states, favicon, page titles, and meta descriptions. Give me a checklist of what you changed.

PROMPT

Write a README explaining what this is, how to run it locally, and how to deploy it.

13.8 Learning

PROMPT

Explain what you just did as if I'm new to coding, using an everyday analogy. Then quiz me with 3 questions.

PROMPT

Give me a guided tour of this project's files: what each folder is for and which 5 files matter most.

Error Message Decoder

When you see one of these, here's what it usually means and what to tell the AI.

MESSAGE (OR PART OF IT)	PLAIN ENGLISH	TELL THE AI
404 Not Found	The page or API address doesn't exist	"Check the route or URL for [x]"
401 Unauthorized	Not logged in, or a missing or wrong key	"Check auth/session and the API key setup"
403 Forbidden	Logged in but not allowed (often RLS)	"Check permissions/RLS for this action"
500 Internal Server Error	The server code crashed	"Check the server logs for the real error"
CORS policy	The browser blocked a call to another domain	"Configure CORS on the API, or call it from the server"
undefined is not a function / Cannot read properties of undefined	The code used something that doesn't exist yet	"Find where [x] is undefined and guard it"
Module not found / Cannot find module	A missing package or wrong import path	"Install the package or fix the import"
Unexpected token	A syntax error, often a missing bracket	"Find the syntax error near line N"
Hydration failed	Server and browser rendered different HTML (React/Next.js)	"Fix the hydration mismatch in [component]"
ENV variable is undefined	The secret isn't set on the host	"List the env vars I need to set on [host]"
Build failed on deploy	Works locally but not on the host	"Here's the deploy log; fix the build"
429 Too Many Requests	Rate limited	"Add retry with backoff and caching"
Network Error / Failed to fetch	Server down, wrong URL, or offline	"Check the URL and whether the server is running"

The Modern Web Stack, Explained

Builders and AIs throw these names around constantly. Here's what each one is, in a sentence.

NAME	WHAT IT IS
HTML	The structure of a web page
CSS	The styling: colours, layout, fonts
JavaScript / TypeScript	The behaviour. TypeScript adds type checking
React	A popular library for building interfaces from components
Next.js	A React framework with routing, server code, and deployment conventions
Vite	A fast development server and build tool
Tailwind CSS	Styling with small utility classes, like <code>p-4 text-lg</code>
shadcn/ui	Copy-in React components that work with Tailwind
Node.js	Runs JavaScript outside the browser (servers, tools)
npm / pnpm	Install JavaScript packages
Supabase	A hosted Postgres database plus auth, storage, and APIs
Firebase	Google's app back end: database, auth, hosting
Postgres	A powerful open-source SQL database
Prisma / Drizzle	Tools for talking to databases from code
Stripe	Payments
Vercel / Netlify	Hosting for front ends and serverless functions
Cloudflare	Domain names (DNS), security, and edge hosting
GitHub	Where your code and its history live (Vol. 2)
API	A way for programs to talk to each other
Serverless function	A small piece of back-end code that runs on demand

Frequently Asked Questions

Can I really build an app without coding? You can build real, useful apps. For anything handling money, health data, or other people's private data, have an experienced developer review it.

Who owns the code? Usually you do, but read each tool's terms. Export to GitHub regularly so you're never locked in.

Why does the AI keep breaking things that worked? Long conversations lose detail. Use version control, work in small slices, and start fresh chats with a summary.

What's the cheapest way to start? A free tier of a browser builder for prototypes, then free GitHub Pages or a free hosting tier for static sites. Pricing changes often, so check each tool's current page.

When should I move from a builder to an editor or agent? When you need custom logic, you're hitting builder limits, or credits get expensive. See Chapter 7.

Deploying, Step by Step

Chapter 8 covered the ideas behind shipping. This chapter is the click-by-click version for the three most common free starting points.

IN THIS CHAPTER

- GitHub Pages for static sites
- Vercel and Netlify for modern apps
- Environment variables on a host
- Custom domains and HTTPS
- Rolling back a bad deploy

17.1 GitHub Pages (static HTML/CSS/JS)

- 1 **Push your project to GitHub** (Vol. 2). The home page must be called `index.html`.
- 2 **Open** `Settings` > `Pages` in the repository.
- 3 **Choose your source:**
Deploy from a branch, then `main` and `/ (root)`, then **Save**.
- 4 **Wait a minute**, then open the URL shown: `https://username.github.io/repo/`.
- 5 **For a framework app** (React/Vite), build it first, or use a GitHub Actions workflow that builds and deploys (Vol. 2, Actions cookbook).

17.2 Vercel or Netlify

- 1 **Sign in with GitHub** at the host's website.

- 2 **Import your repository.**
Choose *Add New* → *Project* (Vercel) or *Add new site* → *Import an existing project* (Netlify).
- 3 **Check the detected settings.**
Common build commands: `npm run build`. Output folders: `dist` (Vite), `.next` (Next.js, handled automatically on Vercel), `build` (Create React App).
- 4 **Add environment variables** (below) before the first deploy.
- 5 **Deploy.**
Every later push to `main` redeploys automatically, and pull requests get preview URLs.

17.3 Environment variables on the host

KIND	EXAMPLE	WHERE IT GOES
Public (safe in the browser)	<code>VITE_SUPABASE_URL</code> , <code>NEXT_PUBLIC_SUPABASE_ANON_KEY</code>	Host env settings. Prefixes like <code>VITE_</code> and <code>NEXT_PUBLIC_</code> expose them to the front end
Secret (server only)	<code>STRIPE_SECRET_KEY</code> , <code>SUPABASE_SERVICE_ROLE_KEY</code> , <code>OPENAI_API_KEY</code>	Host env settings, without a public prefix, used only in server functions

⚠ DANGER

Anything with a public prefix ends up inside your site's JavaScript, where anyone can read it. Secret keys must never have one.

17.4 Custom domain

- 1 **Buy a domain** from a registrar.
- 2 **Add it in your host** (Vercel/Netlify: Domains; GitHub Pages: `Settings` > `Pages` > `Custom domain`).
- 3 **Create the DNS records the host shows you**, usually an `A` record for the bare domain and a `CNAME` for `www` .
- 4 **Wait for DNS to update.**
It can take minutes to hours.
- 5 **Turn on HTTPS.**
All three hosts provide free certificates. Tick *Enforce HTTPS* where offered.

17.5 Rolling back

On Vercel and Netlify, open **Deployments**, find the last good one, and choose *Promote / Publish deploy*. On GitHub Pages, `git revert` the bad commit and push. Rollbacks take seconds, which is one more reason to deploy small changes often.

Databases and Security Rules, Explained

Most serious vibe-coding mistakes happen in the database. This chapter explains just enough SQL and row-level security to check the AI's work.

IN THIS CHAPTER

- Tables, rows, and columns
- Five SQL statements to recognise
- Row-level security (RLS) policies you can read

18.1 Five SQL statements

SQL

```
-- Make a table
create table tasks (
  id bigint generated always as identity primary key,
  user_id uuid references auth.users not null,
  title text not null,
  done boolean default false,
  created_at timestampz default now()
);

select * from tasks where done = false;      -- read
insert into tasks (user_id, title) values (auth.uid(), 'Buy milk'); -- create
update tasks set done = true where id = 42;  -- change
delete from tasks where id = 42;             -- remove
```

18.2 Row-level security in Supabase

RLS decides which rows each logged-in user can see or change. With RLS on and no policies, nobody can access the table through the public API. Each policy opens a specific door.

SQL

```
alter table tasks enable row level security;

create policy "Users read own tasks" on tasks
  for select using (auth.uid() = user_id);

create policy "Users add own tasks" on tasks
  for insert with check (auth.uid() = user_id);

create policy "Users update own tasks" on tasks
  for update using (auth.uid() = user_id);

create policy "Users delete own tasks" on tasks
  for delete using (auth.uid() = user_id);
```

CLAUSE	MEANING
<code>for select / insert / update / delete</code>	Which action the policy covers
<code>using (...)</code>	Which existing rows the user may act on
<code>with check (...)</code>	What new or changed rows must satisfy
<code>auth.uid()</code>	The ID of the logged-in user

⚠ RED FLAGS IN AI-WRITTEN POLICIES

`using (true)` lets **everyone** see every row. A table with RLS turned **off** is fully exposed to anyone holding your public key. Ask the AI to explain every policy in one plain sentence. If it can't, don't ship it.

Project: Build a Habit Tracker End to End with a Coding Agent

This chapter builds a complete app with a coding agent, such as Claude Code, Cursor's agent, Codex or Copilot agent mode. You go from an empty folder to a deployed site. You don't write code yourself. Your job is to direct the agent, check its work, and decide what to do next.

- **What you build:** a habit tracker. You add habits, tick them off each day, see streaks, and your data stays in the browser.
- **Stack:** Vite + React + TypeScript, with localStorage for data. There's no backend, so there are no secrets and nothing to bill.
- **Time:** about 2–3 hours over one or two sessions.
- **You need:** Node.js LTS, Git, a GitHub account, and one coding agent.

19.1 Phase 0: Set up the folder

- 1 Make an empty folder named `habit-tracker` and open a terminal in it.
- 2 Run `git init` so every step can be undone.
- 3 Start your agent in this folder (for example, `claude` or `codex`, or open the folder in Cursor).
- 4 Create a file called `AGENTS.md` (or `CLAUDE.md` for Claude Code) with the rules below. The agent reads it at the start of every session.

AGENTS.MD

Project rules

- Stack: Vite + React + TypeScript. No other frameworks.
- Store data in localStorage under the key "habits.v1".
- Keep components small: one component per file in src/components.
- After every change run: `npm run build && npm test`
- Never add a dependency without asking first.
- Commit after each working feature with a clear message.

19.2 Phase 1: Plan before code

Paste this prompt:

TEXT

Read AGENTS.md. Don't write code yet.
Propose a plan for a habit tracker with: add/rename/delete habits,
a daily checkbox per habit, a current-streak counter, and a 7-day history grid.
List the files you'll create, the data shape you'll store, and 5 tests you'll write.
Wait for my OK.

Check the plan for:

- a data shape you understand, e.g. `{ id, name, createdAt, doneDates: string[] }`
- dates stored as `YYYY-MM-DD` strings rather than timestamps, which avoids time-zone bugs
- tests for streak logic, which is the bug-prone part

If any of these is missing, ask the agent to change the plan. Correcting a plan costs nothing.
Correcting a codebase costs a lot.

19.3 Phase 2: Scaffold and first run

TEXT

OK, go. Scaffold with Vite (react-ts template), add Vitest,
implement only "add habit" and the list. Run the build and tests, then
tell me the command to start the dev server.

- 1 Run the dev server it gives you (usually `npm run dev`) and open the local URL.
- 2 Add two habits, then reload the page. **They must still be there.** If they disappear, say: "Habits vanish on reload; check localStorage save and load."
- 3 When it works, commit: `git add -A && git commit -m "Add habits list"`. You can also ask the agent to commit.

19.4 Phase 3: Features, one at a time

Give one prompt per feature. After each one, test it in the browser and then commit.

#	PROMPT	HOW YOU CHECK IT
1	"Add a checkbox per habit for today. Clicking toggles today's date in doneDates."	Tick it, reload, and it's still ticked
2	"Add current streak: consecutive days ending today (or yesterday if today isn't done). Write unit tests first."	Tests pass, and the streak shows 1 after ticking
3	"Add a 7-day grid showing done/not done for each of the last 7 days."	The oldest day is on the left, and today is highlighted
4	"Add rename and delete, with a confirm before delete."	Delete asks first
5	"Add export/import of all data as a JSON file."	Export, clear storage, import, and everything comes back
6	"Make it look clean on a phone: max width 480px, big tap targets."	Check with your browser's device toolbar

TIP

Test the streak at midnight without waiting for midnight. Ask: "Make the 'today' date injectable so tests can fake any date, and add tests for month boundaries and leap day."

19.5 Phase 4: Review like an editor

Before you ship, ask for a review of the agent's own work:

TEXT

Review the whole codebase as a strict senior engineer. List: bugs, accessibility problems, anything that breaks if localStorage is full or corrupted, and dead code. Don't fix yet, just list by severity.

Fix the high-severity items one prompt at a time. A common catch is corrupted JSON in localStorage crashing the app. The fix wraps `JSON.parse` in try/catch and falls back to an empty list.

19.6 Phase 5: Deploy

- 1 Create an empty GitHub repo, then ask: "Add the GitHub remote and push main."

- 2
- Deploy with any static host from *Deploying, Step by Step*. Use build command `npm run build` and output folder `dist`.
- 3
- Open the live URL on your phone and add a habit there. It's stored separately from your laptop's data, because `localStorage` belongs to each browser.

19.7 Where to take it next

Next Step	What It Teaches	Prompt Starter
Sync across devices	Databases and auth	"Replace <code>localStorage</code> with Supabase; keep an offline fallback"
Reminders	Notifications and service workers	"Make it an installable PWA with a daily reminder"
Charts	Data visualization	"Add a monthly completion chart using a small chart library; ask before installing"
Sharing	Public read-only views	"Generate a read-only share link of my streaks"

Taking Payments Safely with Stripe Test Mode

Payments are where vibe-coded apps get into the most trouble. This chapter shows the safe order: build everything in **test mode**, where no real money moves, and switch to live mode only after a checklist.

⚠ DANGER

Never paste your **secret key** (it starts with `sk_`) into browser code, a public repo, or a chat with an AI tool you don't control. It belongs only in server environment variables. Your publishable key (`pk_`) is designed to be public.

20.1 The safe architecture

PIECE	WHERE IT RUNS	WHAT IT DOES
"Buy" button	Browser	Calls <i>your</i> server endpoint. It never talks to Stripe with a secret
Create Checkout Session	Your server or serverless function	Uses the secret key to make a Stripe Checkout Session and returns its URL
Stripe Checkout page	Stripe	Collects card details, so you never handle card numbers
Webhook endpoint	Your server	Stripe tells you the payment succeeded. Only this should unlock paid features

The success page is not proof of payment. Anyone can visit `/success` by typing the URL. Unlock purchases only when your webhook receives the event and you verify its signature.

20.2 Step by step in test mode

- 1 Create a Stripe account and stay in the **test mode / sandbox** environment shown in the Dashboard.
- 2 Create a product and price in the Dashboard, and copy the price ID (it starts with `price_`).
- 3 Put your test secret key in a server-side environment variable, e.g. `STRIPE_SECRET_KEY`, in `.env.local`. Confirm `.env*` is in `.gitignore`.
- 4 Ask your agent: “Add a serverless endpoint that creates a Stripe Checkout Session in payment mode for price , with success_url and cancel_url, and returns the session URL. Use the official stripe npm package. The secret key comes only from process.env.”
- 5 Ask: “Add a webhook endpoint that verifies the Stripe signature using `STRIPE_WEBHOOK_SECRET` and, on checkout.session.completed, marks the user as paid.”
- 6 Install the Stripe CLI and forward events locally with `stripe listen --forward-to localhost:3000/api/webhook`. It prints a webhook signing secret. Put that in `STRIPE_WEBHOOK_SECRET`.
- 7 Pay with Stripe’s test card **4242 4242 4242 4242**, any future expiry date, any 3-digit CVC and any postcode.
- 8 Watch the terminal running `stripe listen`. You should see `checkout.session.completed` delivered with a 200 response.

20.3 Test the failure paths too

SCENARIO	HOW TO TEST	WHAT SHOULD HAPPEN
Card declined	Use a decline test card from Stripe's testing docs (docs.stripe.com/testing)	A friendly error, and nothing unlocked
User closes Checkout	Click back or cancel	Lands on cancel_url, and nothing unlocked
Fake success visit	Open <code>/success</code> directly without paying	Nothing unlocked
Webhook replay or forgery	Send a request without a valid signature	400 response, and nothing unlocked
Same event twice	Resend an event with <code>stripe events resend <evt_id></code>	Still only one purchase recorded (idempotent)

20.4 Going live checklist

- ☐ ☐ All tests above pass in test mode
- ☐ ☐ Live keys are set only in your host's production environment variables
- ☐ ☐ A live webhook endpoint is created in the Dashboard, and its live signing secret is set
- ☐ ☐ Prices, currency, tax and refund policy have been reviewed by you, not just the agent
- ☐ ☐ A real low-price purchase with your own card succeeds, and you refund it afterward
- ☐ ☐ Terms and privacy pages exist and are linked from checkout

Accessibility and Performance Checklist

Run this checklist before every launch. Each item is one prompt to your agent or one quick manual check. Accessibility makes your app usable by more people. Performance stops them from leaving.

21.1 Accessibility (a11y)

CHECK	HOW TO TEST	FIX PROMPT
Keyboard only	Unplug the mouse and Tab through everything. Can you reach and use every control?	"Make every interactive element reachable and usable by keyboard, with a visible focus outline"
Images have alt text	Inspect <code></code> tags	"Add meaningful alt text to content images and empty alt to decorative ones"
Form labels	Click a label. Does it focus the field?	"Give every input a visible <code><Label></code> linked with <code>htmlFor</code> "
Color contrast	Run your browser's Lighthouse accessibility audit	"Fix all contrast failures Lighthouse reports; keep the palette"
Buttons vs links	Do actions use <code><button></code> and navigation use <code><a></code> ?	"Replace clickable divs with real buttons or links"
Headings in order	One <code>h1</code> , then h2/h3 without skipping levels	"Fix heading levels so they're in logical order"
Screen reader	Try VoiceOver (Mac/iOS) or NVDA (Windows) on your main flow	"Add aria-live to status messages like 'Saved'"
Motion	Respect reduced-motion settings	"Disable animations when prefers-reduced-motion is set"

21.2 Performance

CHECK	HOW TO TEST	FIX PROMPT
Lighthouse score	Chrome DevTools → Lighthouse → Analyze (mobile)	"Here's the Lighthouse report: . Fix the top 3 opportunities"
Huge images	Network tab, sorted by size	"Serve images in WebP/AVIF at the displayed size, lazy-load below the fold"
Big JavaScript bundle	Build output sizes	"Find the largest dependencies and code-split or replace them"
Layout shift	Content jumps while loading	"Reserve space for images and embeds with width/height"
Slow API calls	Network tab timing	"Cache this response and show a skeleton while loading"
Fonts	Flash of invisible text	"Use font-display: swap and preload the main font"

TIP

Run Lighthouse in an incognito window. Browser extensions can make scores look worse than they are.

After Launch: Maintaining Your App

Shipping is the start, not the end. A small weekly routine keeps a vibe-coded app from rotting.

22.1 Weekly 20-minute routine

- 1 Check errors.**
Open your host's logs, or an error tracker if you added one. Paste anything new into your agent: "Explain and fix this production error: ..."
- 2 Check dependency alerts.**
If the repo is on GitHub, look at Dependabot alerts under the repo's **Security** tab.
- 3 Back up data.**
Export your database, or confirm automatic backups are on in your provider's dashboard.
- 4 Read feedback.**
Turn the top request into one prompt.
- 5 Ship one small improvement** on a branch, preview it, then merge.

22.2 Monthly

- ☐ ☐ Update dependencies on a branch: “Update dependencies to latest minor versions, run build and tests, report anything that broke”
- ☐ ☐ Rotate any API key that was ever shared or pasted somewhere it shouldn't have been
- ☐ ☐ Review costs: hosting, database, AI APIs, and the agent's own subscription
- ☐ ☐ Re-run the accessibility and performance checklist
- ☐ ☐ Delete test accounts and old preview deployments you no longer need

22.3 When something breaks in production

STEP	ACTION
1. Stop the bleeding	Roll back to the last good deployment. Most hosts have a one-click “promote” or “rollback” on older deploys
2. Reproduce	Get the exact error message, URL, browser and time
3. Fix on a branch	Give the agent the error and the recent changes (<code>git log -5 --stat</code>)
4. Add a test	“Add a test that would have caught this”
5. Redeploy	Merge, watch the logs for 15 minutes
6. Write it down	One line in a <code>CHANGELOG.md</code> or notes file about what happened

22.4 Handing your app to someone else

If someone else will maintain the app (a developer you hire, or future you), ask the agent to write it up:

TEXT

Write a `HANDOFF.md`: what the app does, how to run it locally, every environment variable (names only, no values), where it's deployed, how to deploy, known issues, and the 5 files someone should read first.

Vibe Coding Prompt Patterns for Every Stage

These are reusable prompt patterns, organized by stage of a project. Replace anything in angle brackets.

23.1 Starting

PATTERN	PROMPT
Interview me	"Before building, ask me up to 10 questions about to pin down users, features and must-nots. One question at a time."
MVP cut	"Here's my feature list: . Cut it to the smallest version someone would actually use. Explain each cut."
Stack pick	"Suggest the simplest stack for for a non-programmer. Prefer free tiers. Explain trade-offs in plain English."
Rules file	"Write an AGENTS.md for this project with stack, conventions, test command, and things never to do."

23.2 Building

PATTERN	PROMPT
One slice	"Implement only . Don't touch other files. Run the build and tests after."
Tests first	"Write failing tests for , show me them, then implement until they pass."
Explain as you go	"After each change, give me a 3-line summary: what changed, why, how I can check it."
Match the style	"Look at and build the same way."
Mock data	"Add realistic fake data for 20 so I can see the UI filled in."

23.3 Debugging

PATTERN	PROMPT
Full context	"Here's the exact error, what I did before it, and what I expected: <...>. Find the root cause before changing code."
Bisect	"This worked at commit . Compare to now and find what broke it."
Add logging	"Add temporary logging around so we can see what's happening, then we'll remove it."
Fresh eyes	"Forget your previous attempts. List 3 different hypotheses and how to test each."
Revert	"Revert your last change completely; we'll try a different approach."

23.4 Reviewing and shipping

PATTERN	PROMPT
Security pass	"Audit for secrets in code, missing auth checks, and unvalidated input. List by severity."
Simplify	"Find code that's more complicated than it needs to be and propose simpler versions."
Release notes	"Summarize changes since the last tag as user-facing release notes."
Readme	"Write a README for a non-technical user: what it does, how to use it, screenshot placeholders."

Ten More App Ideas with Starter Prompts

Each idea below lists its difficulty, what it teaches, and a first prompt you can paste. Start with the beginner ones.

APP	LEVEL	TEACHES	FIRST PROMPT
Tip and bill splitter	Beginner	Forms, math, mobile layout	"Build a one-page bill splitter: total, tip %, people, rounding option."
Flashcards	Beginner	localStorage, simple state	"Flashcard app: decks, flip cards, mark known/unknown, shuffle."
Recipe scaler	Beginner	Parsing text, units	"Paste a recipe, choose servings, scale every quantity correctly."
Reading list	Beginner	CRUD, filtering	"Track books: to-read/reading/done, rating, notes, search."
Pomodoro timer	Beginner	Timers, notifications	"25/5 Pomodoro with sounds, a session counter, and a settings panel."
Expense tracker	Intermediate	Charts, CSV export	"Log expenses by category, monthly chart, export CSV."
Event RSVP page	Intermediate	Database, forms, email	"Event page with RSVP form saved to a database; admin page lists guests."
Club membership site	Intermediate	Auth, roles	"Members log in to see a private page; admins approve new members."
AI writing helper	Intermediate	Calling an AI API from a server	"Server endpoint calls an AI API to rewrite text in 3 tones; key stays server-side."
Booking calendar	Advanced	Dates, conflicts, payments	"Book 1-hour slots, prevent double-booking, optional Stripe deposit in test mode."

 NOTE

For any app that stores other people's data, such as RSVPs, members or bookings, read *Security for Vibe Coders* and *Databases and Security Rules, Explained* before you share the link.

Quick Reference

The loop: Plan → build one slice → test → checkpoint → repeat

Great prompt: What + why + how it behaves + how you'll know it's right + “don't change anything else”

Stuck? Roll back → ask for causes, not fixes → add logging → fresh chat → simplify → try another model

Before launch: secrets in env vars · row-level security on · server-side checks · spending limits · test as a new user

Grow up path: Browser builder → connect GitHub → open in Cursor / Claude Code / Codex → work via pull requests

Glossary

Agent	An AI that takes actions, such as editing files and running commands, in a loop until a task is done.
API	Application programming interface: a way for one program to talk to another.
API key	A secret string that identifies you to a service, like a password for programs.
Authentication	Proving who a user is, for example with a password, magic link, or Google sign-in.
Back end	The part of an app that runs on a server: data, logins, emails, payments.
Checkpoint	A saved, working version of your project you can return to.
Code editor	A program for viewing and editing code files, such as VS Code or Cursor.
Credits	Units many AI tools use to measure and charge for usage.
Database	Organized storage for an app's information, in tables of rows and columns.
Deploy	Put your app on a server so others can use it. Also called publishing or shipping.
DNS	The internet's address book, which points a domain name at the computer hosting your site.
Environment variable	A setting stored outside your code, used for secrets like API keys.
Front end	The part of an app that runs in the user's browser.
Hosting	Running your app on an always-online computer.
PRD	Product requirements document: a brief describing what to build and why.
Preview	A live, clickable version of your app inside a builder.
Prompt	The instructions you give an AI.

Row-level security	Database rules ensuring each user can only access their own records.
Rules file	A file of lasting instructions an AI tool reads automatically, such as <code>AGENTS.md</code> or <code>CLAUDE.md</code> .
Supabase	A popular hosted database service with built-in logins and file storage.
Token	A small chunk of text used to measure AI usage, roughly three-quarters of a word.
Vibe coding	Building software mainly by describing what you want to an AI and iterating on the results.