

A BEGINNER'S MANUAL

CLAUDE EDITION

Vibe Coding

Build real apps, websites, extensions and tools by
describing them in plain English

Vibe Coding

Build real apps, websites, extensions and tools by describing them in plain English

CLAUDE EDITION Expanded edition written by Claude (Anthropic's AI) in Claude Code and published in GuidesForEverything PR #4.

"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 honestly, explains how web apps work in plain English, and teaches a repeatable process: plan, build in small slices, test, save, and ship. Seven complete project builds take you from a portfolio site to a mobile app and an AI chatbot. It also covers the parts beginners usually miss: databases, logins, payments, email, security, costs, debugging, 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?	10
1.1 A new way to build	10
1.2 What it's great for	11
1.3 Where it needs care	11
1.4 What a session actually looks like	11
1.5 Words you'll meet straight away	12
02 The Vibe-Coding Mindset	13
2.1 Your three jobs	13
2.2 Realistic expectations	14
2.3 Seven habits of successful vibe coders	14
2.4 Myths and realities	15
2.5 When to bring in a professional	15
03 The Tool Landscape	16
3.1 Family 1: App builders in your browser	16
3.2 Family 2: AI code editors	17
3.3 Family 3: Coding agents	17
3.4 Which should you pick?	18
04 App Builders Up Close	19
4.1 How builders charge	19
4.2 Lovable {{Freemium}}	20
4.3 Bolt {{Freemium}}	20
4.4 v0 {{Freemium}}	21
4.5 Replit {{Freemium}}	21
4.6 Base44 {{Freemium}}	22
4.7 Google AI Studio (Build mode) {{Free}}	22
4.8 Figma Make	23
4.9 Side by side	23
05 AI Editors and Agents Up Close	24
5.1 Why move to an editor or agent?	24
5.2 Cursor {{Freemium}}	25
5.3 VS Code + GitHub Copilot {{Freemium}}	25
5.4 Devin Desktop (formerly Windsurf) {{Freemium}}	25
5.5 Kiro {{Freemium}}	26
5.6 Zed {{Freemium}}	26
5.7 Claude Code {{Paid}}	26

5.8	OpenAI Codex {{Freemium}}	27
5.9	Gemini CLI {{Free}}	27
5.10	Devin {{Paid}}	27
5.11	How to choose	27
06	How Web Apps Work	28
6.1	The restaurant model	28
6.2	APIs: how programs talk	29
6.3	Hosting and domains	29
6.4	What happens when someone opens your app	29
6.5	Static sites and dynamic apps	30
6.6	Tech words, translated	30
6.7	A tour of a project folder	31
07	Your First App in a Browser Builder	32
7.1	Build it	32
7.2	Getting to know the screen	33
7.3	Three ways to talk to a builder	34
7.4	Visual editing	34
7.5	Giving the builder lasting instructions	34
7.6	When a change goes wrong	35
7.7	Test on your phone, then share	35
7.8	First-build troubleshooting	35
08	The Vibe Coding Workflow	36
8.1	Start with a one-page brief	37
8.2	Build in slices	37
8.3	Order your slices wisely	38
8.4	Write a test script	38
8.5	Give the AI a rules file	39
8.6	A rhythm for each session	39
8.7	When to start a fresh chat	40
09	Prompts That Work	41
9.1	The anatomy of a great prompt	41
9.2	Prompt templates	42
9.3	Specs: say exactly what "done" means	43
9.4	Use pictures	43
9.5	A vocabulary for design	44
9.6	Giving feedback the AI can act on	44
9.7	Ask for options before decisions	44
9.8	Let the AI write the prompt	45
9.9	Build your own prompt library	45
10	Version Control Safety Nets	46

10.1	Three layers of safety	46
10.2	Git without jargon	47
10.3	Connect your builder to GitHub	47
10.4	Saving versions yourself	48
10.5	Keep secrets out of Git	48
10.6	Branches for experiments	49
10.7	Recovering from a bad change	49
11	Project 1: A Portfolio Site	51
11.1	Step 1: Gather your content	51
11.2	Step 2: Describe the whole site, build the first slice	52
11.3	Step 3: Pour in your real content	52
11.4	Step 4: Polish in small passes	53
11.5	Step 5: Make it shareable and findable	53
11.6	Step 6: Add a contact option	53
11.7	Step 7: Publish	53
11.8	Step 8: Add your own domain (optional)	54
11.9	If something goes wrong	54
12	Project 2: A Landing Page with a Sign-Up Form	55
12.1	Step 1: Plan the page	56
12.2	Step 2: Build the page with a placeholder form	56
12.3	Step 3: Save sign-ups to a database	56
12.4	Step 4: Stop spam	57
12.5	Step 5: Handle personal data properly	57
12.6	Step 6: Send a confirmation email (optional)	57
12.7	Alternative: Netlify Forms, no database needed	58
12.8	Step 7: Launch and measure	58
13	Project 3: A To-Do App with a Database and Logins	60
13.1	Step 1: Design the data	61
13.2	Step 2: Create a Supabase project	61
13.3	Step 3: Connect your builder	62
13.4	Step 4: Add sign-in	62
13.5	Step 5: Lock down the data	62
13.6	Step 6: Prove it with two users	63
13.7	Step 7: Publish and fix redirects	63
13.8	If something goes wrong	64
14	Project 4: A Chrome Extension	65
14.1	How an extension is put together	65
14.2	Step 1: Make a folder and open your agent	66
14.3	Step 2: Describe the extension	66
14.4	Step 3: Load it into Chrome	67
14.5	Step 4: Iterate	67

14.6	Finding errors in an extension	68
14.7	Step 5: Icons and polish	68
14.8	Step 6: Publish to the Chrome Web Store (optional)	68
15	Project 5: A Simple Mobile App	70
15.1	Route A: In a browser builder	71
15.2	Route B: With an agent on your computer	71
15.3	Step 1: Install Expo Go on your phone	71
15.4	Step 2: Describe the app	71
15.5	Step 3: Test on your phone	72
15.6	Step 4: Build the rest in slices	72
15.7	Step 5: Publishing (when you're ready)	72
16	Project 6: An AI Chatbot App	74
16.1	How it's wired	74
16.2	Step 1: Choose an AI provider and get a key	75
16.3	Step 2: Build the chat	75
16.4	Step 3: Give it a personality and boundaries	76
16.5	Step 4: Add limits	76
16.6	Step 5: Test, publish, watch	76
17	Project 7: An Internal Tool or Dashboard	78
17.1	Step 1: Start from your real spreadsheet	79
17.2	Step 2: Describe the tool	79
17.3	Step 3: Roles and permissions	79
17.4	Step 4: Test without risking real data	80
17.5	Step 5: Make it useful daily	80
18	Databases and Supabase	81
18.1	Front end, back end, database	81
18.2	Relationships between tables	82
18.3	Supabase in brief	82
18.4	The two kinds of key	82
18.5	Row-level security, again	83
18.6	Looking after your data	83
19	Logins, Payments, and Email	84
19.1	Logins (authentication)	84
19.2	Payments with Stripe	85
19.3	Sending email	86
19.4	API keys and environment variables	86
20	Leveling Up: Editors and Agents	88
20.1	Signs you've outgrown a browser builder	88
20.2	Moving your project	88
20.3	Working in Cursor	89

20.4	Agents: the most hands-off option	89
20.5	Working through pull requests	90
21	Deployment and Domains	91
21.1	Hosting options	91
21.2	Deploying from GitHub	92
21.3	Environment variables in production	92
21.4	Your own domain name	92
21.5	Pre-launch checklist	93
22	When You Get Stuck	94
22.1	Break the loop	94
22.2	Where errors hide	95
22.3	Error-message playbook	95
22.4	Diagnosis prompts	96
23	Security for Vibe Coders	97
23.1	The big five	97
23.2	Common leaks in vibe-coded apps	98
23.3	Use the scanners	98
23.4	Ask the AI to audit itself	98
23.5	If a key leaks	99
23.6	Privacy and the law	99
24	Costs and Credits	100
24.1	Where the money goes	100
24.2	Make credits last	101
24.3	Free tiers worth knowing	101
24.4	A typical starter budget	101
24.5	Avoiding surprise bills	102
25	Maintaining and Handing Off	103
25.1	A light maintenance routine	103
25.2	Updating libraries safely	104
25.3	Write a README	104
25.4	Handing off to a developer	104
26	Troubleshooting Table	106
26.1	Building	106
26.2	Data and logins	106
26.3	Payments and email	107
26.4	Deploying and domains	107
26.5	Costs and limits	107
27	Three Complete Builds	108
27.1	Build 1: Portfolio site	108

27.2	Build 2: To-do app with logins	109
27.3	Build 3: Landing page with a waitlist	110
28	The Prompt Library	111
28.1	Planning	111
28.2	Building	112
28.3	Styling	112
28.4	Debugging	112
28.5	Refactoring and quality	113
28.6	Security	113
28.7	Shipping	114
28.8	Learning	114
29	Error Message Decoder	115
30	The Modern Web Stack, Explained	116
31	Frequently Asked Questions	117
32	Deploying, Step by Step	118
32.1	GitHub Pages (static HTML/CSS/JS)	118
32.2	Vercel or Netlify	118
32.3	Environment variables on the host	119
32.4	Custom domain	119
32.5	Rolling back	120
33	Databases and Security Rules, Explained	121
33.1	Five SQL statements	121
33.2	Row-level security in Supabase	122
34	Project: Build a Habit Tracker End to End with a Coding Agent	123
34.1	Phase 0: Set up the folder	123
34.2	Phase 1: Plan before code	124
34.3	Phase 2: Scaffold and first run	124
34.4	Phase 3: Features, one at a time	125
34.5	Phase 4: Review like an editor	125
34.6	Phase 5: Deploy	125
34.7	Where to take it next	126
35	Taking Payments Safely with Stripe Test Mode	127
35.1	The safe architecture	127
35.2	Step by step in test mode	128
35.3	Test the failure paths too	128
35.4	Going live checklist	129
36	Accessibility and Performance Checklist	130
36.1	Accessibility (a11y)	130
36.2	Performance	131

37	After Launch: Maintaining Your App	132
37.1	Weekly 20-minute routine	132
37.2	Monthly	133
37.3	When something breaks in production	133
37.4	Handing your app to someone else	133
38	Vibe Coding Prompt Patterns for Every Stage	134
38.1	Starting	134
38.2	Building	134
38.3	Debugging	135
38.4	Reviewing and shipping	135
39	Ten More App Ideas with Starter Prompts	136
	Quick Reference	138
	Glossary	139

What Is Vibe Coding?

Vibe coding is the friendliest way into software that has ever existed. You describe what you want in everyday words, an AI writes the code, and you judge the result by using it. This chapter explains where the idea came from, what it is good at, and where you need to be careful.

IN THIS CHAPTER

- Where the term came from and what it really means
- The three levels of vibe coding, and which one to aim for
- What vibe coding is great at, and where it needs care
- What a real vibe-coding session looks like

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.

Before this, building software meant learning a **programming language** (a precise written language computers can follow, such as JavaScript or Python) and typing every instruction yourself. Now the AI does the typing. Your job moves from *writing code* to *deciding what to build and checking that it works*.

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.”
- **Browser extensions and small mobile apps:** little helpers that fix one annoyance in your day.
- **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	Follow the security chapter's checklist
Big, long-lived products	AI-written code can grow messy	Use Git, keep a plan, tidy up 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.

1.4 What a session actually looks like

Here is a realistic first hour with a browser app builder, so you know what to expect.

TIME	WHAT YOU DO	WHAT THE AI DOES
0–10 min	Write a short description of your idea	Asks a question or two, then builds a first version
10–25 min	Click around the preview and note problems	Fixes one problem per message
25–40 min	Ask for the next feature	Adds it, sometimes breaking something small
40–50 min	Spot the breakage and describe it precisely	Repairs it
50–60 min	Save a checkpoint and publish	Gives you a public web address

Notice that most of your time goes into **trying the app and describing what you see**. That is the core skill, and you already have it: you use apps every day.

1.5 Words you'll meet straight away

You do not need to learn to program, but a few words come up constantly. Each is explained again where it matters, and all of them are in the glossary.

WORD	PLAIN MEANING
Code	The written instructions that make an app work
Prompt	The message you type to the AI
Preview	A live, clickable version of your app while you build
Deploy / publish	Put your app online so others can use it
Database	Where the app stores information so it isn't lost
Bug	Something that doesn't work the way it should
Repository (repo)	A project folder whose history is tracked, usually on GitHub

- ☐ I know vibe coding means describing, testing, and refining
- ☐ I'm aiming for "AI-assisted building," not blind acceptance
- ☐ I know which kinds of projects need extra care
- ☐ I expect to spend most of my time testing and describing

The Vibe-Coding Mindset

The tools in this manual change every few months. The habits in this chapter don't. People who succeed with vibe coding are rarely the most technical; they are the clearest thinkers and the most patient testers.

IN THIS CHAPTER

- Your role: product manager, tester, and editor
- Realistic expectations, including the “last 20%” wall
- Seven habits that separate finished apps from abandoned ones
- Common myths, and the truth behind them

2.1 Your three jobs

When you vibe code, the AI handles the typing. You keep three jobs that no AI can do for you:

1. **Product manager.** Decide what to build, for whom, and what “done” means. Say no to features that can wait.
2. **Tester.** Use the app like a real person would, including a confused or careless one. Find what's broken.
3. **Editor.** Read the AI's plans and summaries, ask “why?”, and stop it when it heads the wrong way.

NOTE

You will also become a little bit of a developer over time, whether you mean to or not. That's a good thing. Each concept you pick up, like “front end” or “environment variable,” makes your prompts sharper and your apps safer.

2.2 Realistic expectations

AI builders are astonishing at the first 80% of an app. In an hour you can have something that looks finished. The last 20%, the edge cases, the logins, the “what if two people do this at once,” takes longer. Experienced builders call this **the last-20% wall**.

STAGE	HOW IT FEELS	WHAT HELPS
First version	Magical: it works in minutes	Enjoy it, then save a checkpoint
Adding features	Mostly smooth, occasional breakage	Small slices, one feature per message
Real users and data	Slower; details matter	Security checklist, testing as a new user
Long-term upkeep	Changes ripple in surprising ways	Git, a rules file, regular tidy-ups

None of this is a reason to stop. It's a reason to plan for it, so that a slow week doesn't feel like failure.

2.3 Seven habits of successful vibe coders

THE HABITS

- 1 Start smaller than you want.**
Build the one feature that proves the idea. Everything else waits for version 2.
- 2 Write it down first.**
A one-page brief beats an hour of improvised prompting.
- 3 One change per message.**
Small requests produce small, understandable mistakes.
- 4 Test after every change.**
Click the thing you just changed, and one thing you didn't.
- 5 Save working versions.**
A checkpoint or Git commit after each success makes every mistake reversible.
- 6 Read what the AI tells you.**
Its summary often admits “I also changed X” or “I couldn't do Y.”
- 7 Ask “explain this” often.**
Understanding grows quietly, and it pays off when things break.

2.4 Myths and realities

MYTH	REALITY
"I'll never need to understand anything."	You'll need concepts, not code: what a database is, where secrets live
"If it looks done, it is done."	Looks are the easy part; test data, logins, and errors
"More detailed prompts are always better."	Clear beats long. One focused request wins
"The AI knows best."	It is confident, not always right. You decide
"I picked the wrong tool and must start over."	Code moves between tools through GitHub
"Vibe-coded apps can't be real businesses."	Many are, once they pass security and testing basics

2.5 When to bring in a professional

Vibe coding takes you a long way on your own. Consider paying a developer for a review when:

- you are about to handle payments, health data, or children's data;
- your app has hundreds of real users and changes keep breaking it;
- you need to meet a legal or industry standard;
- you have been stuck on the same problem for more than a few days.

A few hours of expert review costs far less than a data leak. The "Maintaining and Handing Off" chapter shows how to prepare your project so that review is quick.

TIP

Keep a **build diary**: a simple note with the date, what you asked for, what worked, and what broke. It costs a minute a day and it is the best debugging tool you'll ever have.

- ☐ I understand my three jobs: product manager, tester, editor
- ☐ I expect the last 20% to take longer, and I'm fine with that
- ☐ I've picked one small first feature
- ☐ I have somewhere to keep a build diary

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: where it runs, who it suits, and how far it can take you.

IN THIS CHAPTER

- The three families: app builders, AI editors, and coding agents
- The main tools in each family, with starting prices
- A quick way to pick your first tool

3.1 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, including mobile apps, in a cloud workspace	Free Starter; 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
Figma Make	Figma	Turns prompts and Figma designs into working prototypes and sites	Part of Figma's plans

3.2 Family 2: AI code editors

A **code editor** is a program for viewing and editing code files. AI editors have 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; Pro \$20/mo
Kiro	Amazon's editor, focused on writing specs before code	Free; Pro \$20/mo
Zed	Very fast editor with AI agents built in	Free; Pro \$10/mo

3.3 Family 3: Coding agents

An **agent** is an AI that takes actions in a loop until a job is done. Coding agents work on your files, run commands, test their own work, and keep going. They can work in a **terminal** (a text window for typing commands), 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 plans, from Free to Plus (\$20/mo) and Pro
Gemini CLI	Google's open-source terminal agent	Free tier with a Google account
Copilot cloud agent	Assign a GitHub issue to Copilot; get a pull request	Copilot Pro and up
Devin	Cognition's autonomous cloud engineer	Devin Pro (\$20/mo) and up

 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.

3.4 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 or Figma Make
Build and host an app with a backend in one place	Replit or Lovable
Build a phone app without installing anything	Bolt or Replit (both use Expo)
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 cloud 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.

The next two chapters look at each family up close, with honest strengths and weaknesses.

App Builders Up Close

Browser app builders are where most people start, and many never need to leave. They look alike at first glance, but each has a personality. This chapter gives an honest profile of each, so you can pick one with confidence.

IN THIS CHAPTER

- Lovable, Bolt, v0, Replit, Base44, Google AI Studio, and Figma Make
- What each is best at, where each struggles, and what it costs
- How credits and tokens work in each
- A side-by-side comparison

4.1 How builders charge

Every builder measures AI work in some unit: **credits**, **tokens** (small chunks of text, roughly three-quarters of a word), or dollars of usage. A tiny change like “make the button gray” costs little; “add login and a database” costs more. Large projects cost more per message, because the AI has to read more of your code each time.

TIP

Most builders now have a **planning or chat mode** that discusses changes without editing code. It is usually cheaper than building, and it prevents expensive wrong turns.

4.2 Lovable FREEMIUM

Lovable turns a description into a polished full-stack web app: pages, a database, logins, and hosting.

Best for	Good-looking web apps with accounts and data
Backend	Built-in “Lovable Cloud” (database, auth, storage, functions) or your own Supabase
Modes	Build (makes changes), Plan (writes a plan you approve), Chat (discuss only)
Price	Free: 5 credits a day (up to 30 a month). Pro from \$25/mo (100 credits)

Strengths. Attractive designs out of the box; two-way sync with GitHub, GitLab, or Bitbucket; **Knowledge** settings that hold lasting instructions; version history with bookmarks; built-in payments (Stripe or Paddle); a Quick security scan that runs every time you publish, plus an optional Deep scan.

Watch out for. Credits can go quickly on big projects. Custom domains need a paid plan. Monthly credits expire if unused (two months after issue on monthly plans), so don’t stockpile.

i NOTE

Lovable says you own the code, apps, and data you create. A Plan mode message costs one credit, plus any research the AI runs while planning.

4.3 Bolt FREEMIUM

Bolt, from StackBlitz, runs a full development environment inside your browser tab. It is fast, flexible, and handles mobile apps.

Best for	Quick prototypes, mobile apps (with Expo), importing Figma designs
Backend	Bolt Cloud: hosting, domains, database, sign-in, file storage, payments; or Supabase
Modes	Build, plus Plan Mode to agree on an approach first
Price	Free: 300K tokens a day, 1M a month. Pro \$25/mo (from 10M tokens). Teams \$30/member

Strengths. Very fast iteration; starts from a prompt, uploaded files, a Figma frame, or a Google Stitch design; GitHub sync; a database security check that flags issues such as a missing access rule; paid tokens roll over for one extra month.

Watch out for. Token use grows with project size, because Bolt syncs your whole project to the AI with each message. Keep projects focused.

4.4 v0 FREEMIUM

v0 is Vercel's builder. It shines at interfaces: clean layouts using modern web components, ready to deploy on Vercel.

Best for	Landing pages, dashboards, portfolio sites, polished interfaces
Backend	Connects to databases through Vercel's marketplace (Supabase, Neon, and others)
Editing	Design Mode for visual tweaks without prompting
Price	Free: 7 messages a day. Plus \$30/user/mo (\$30 of credits plus \$2 daily on login)

Strengths. Excellent visual quality; one-click deploy to Vercel; GitHub sync; produces standard Next.js code that any developer recognizes.

Watch out for. The free plan's daily message limit is tight. Back-end features take more setup than in Lovable or Base44.

4.5 Replit FREEMIUM

Replit is a complete cloud workspace with an AI **Agent** that plans, builds, tests, and publishes apps, including native mobile apps.

Best for	Apps with real back ends, scripts, bots, mobile apps, dashboards from data
Agent modes	Free Mode (no-cost everyday work within an allowance), Power , and Max
Safety net	Checkpoints after each request, with one-click rollback
Price	Starter free (daily Free Mode cap; one published app for 30 days). Core \$20/mo. Pro \$100/mo

Strengths. Handles almost any kind of app or language; built-in database, secrets, and publishing; mobile apps you can test with Expo Go and send to the App Store; Plan and Build modes; it asks for confirmation before a paid action starts.

Watch out for. Paid work uses **effort-based pricing**: complex requests cost more. Watch the usage meter in your first weeks.

4.6 Base44 FREEMIUM

Base44, owned by Wix, builds complete apps with the database, logins, and hosting already wired together. It is very beginner-friendly.

Best for	Internal tools, client portals, simple business apps
Backend	Built in: data tables, user sign-in, data permissions, workflows, email
Extras	Plan mode, a separate test database, one-click security scan, Stripe setup
Price	Free: 25 message credits a month. Paid plans from Starter (\$16/mo billed yearly)

Strengths. The least setup of any builder; data permissions you can set in plain English; paid plans include a free domain for a year; two-way GitHub sync exports your full source code.

Watch out for. On the Free plan you can view your code but not edit it. Yearly billing saves 20%, but try monthly first.

4.7 Google AI Studio (Build mode) FREE

Google AI Studio's Build mode creates web apps powered by Google's Gemini models.

Best for	Free experiments and AI-powered apps (image, text, voice)
Backend	Can set up Firebase: Firestore database and "Sign in with Google"
Extras	Secrets management, GitHub sync, download as ZIP
Deploy	Share, or deploy to Google Cloud Run (Cloud Run charges may apply)

Strengths. Free for prototyping; your Gemini API key stays on the server side; great for apps that use AI features.

Watch out for. Deploying to Cloud Run uses your Google Cloud account and its billing. Set a budget alert there.

4.8 Figma Make

Figma Make turns prompts, and designs you already have in Figma, into working prototypes and responsive websites you can publish. It is a natural choice if you or your designer already live in Figma. Check Figma's pricing page for how its AI credits work on your plan.

4.9 Side by side

BUILDER	EASIEST FOR	MOBILE APPS	CODE OWNERSHIP
Lovable	Polished web apps with logins	Web only (phone-friendly)	Two-way Git sync
Bolt	Fast prototypes, Figma imports	Yes, via Expo	GitHub sync, export
v0	Interfaces and landing pages	Web only	GitHub sync
Replit	Anything with a real back end	Yes, via Expo	Full workspace access
Base44	Internal tools, least setup	Web only	GitHub sync (paid plans)
AI Studio	Free AI-powered experiments	Web only	GitHub sync, ZIP

☆ IMPORTANT

Whatever you choose, **connect GitHub in your first session**. It backs up your work, and it means you can move the project to any other tool later.

- ☐ I've chosen one builder to start with
- ☐ I know how it charges (credits, tokens, or usage)
- ☐ I've found its plan or chat mode
- ☐ I know where its version history lives

AI Editors and Agents Up Close

Editors and agents work on real files on your computer (or in a cloud copy of your project). They are more powerful than browser builders and less hand-holding. You don't need them on day one, but it helps to know what each is for.

IN THIS CHAPTER

- What an editor gives you that a builder doesn't
- Profiles of Cursor, Copilot, Devin Desktop, Kiro, and Zed
- Profiles of Claude Code, Codex, Gemini CLI, and Devin
- How to choose, and how they combine

5.1 Why move to an editor or agent?

- **You own the files.** Your project lives in a normal folder, backed up to GitHub.
- **No ceiling.** Any language, any library, any hosting service.
- **Flat-rate plans.** A monthly subscription is often cheaper than builder credits for heavy building.
- **You learn faster.** You see every file the AI touches.

The trade-off: you set up things yourself, like installing tools and running the app locally. The AI can do most of that for you if you ask.

5.2 Cursor FREEMIUM

Cursor looks like VS Code (the most popular free code editor) with AI woven through it. It is owned by SpaceX/xAI, and its plans are part of “The X Subscription,” which also covers Grok.

PLAN	PRICE	NOTES
Free	\$0	Limited usage
Plus	\$8/mo	Grok models only
Premium	\$30/mo	Most popular; more usage, latest models
Super / Ultra	\$100 / \$200 per month	For heavy daily agent use

Usage comes from one pool shared across Cursor and Grok, and the allowance **refreshes weekly**, so one heavy day doesn’t wipe out your month.

Best for: people who want the most polished all-round AI editor. **Watch out for:** the cheapest paid plan limits you to Grok models.

5.3 VS Code + GitHub Copilot FREEMIUM

VS Code is free from Microsoft; **GitHub Copilot** adds AI chat, autocomplete, an agent mode, and a **cloud agent** you can assign GitHub issues to.

PLAN	PRICE	NOTES
Copilot Free	\$0	Up to 2,000 completions a month; automatic model choice
Copilot Student	\$0	For verified students
Copilot Pro	\$10/mo	More AI credits; cloud agent included
Copilot Pro+ / Max	\$39 / \$100 per month	Larger allowances of AI credits

Best for: anyone already on GitHub, and anyone on a tight budget. **Watch out for:** Copilot Free doesn’t include the cloud agent.

5.4 Devin Desktop (formerly Windsurf) FREEMIUM

Cognition renamed its Windsurf editor **Devin Desktop**. It sits alongside the Devin CLI and **Devin Cloud** agents under one account.

- **Free:** a light quota, unlimited Tab completions and inline edits.
- **Pro, \$20/mo:** frontier models from several AI companies and access to cloud agents.
- **Max, \$200/mo:** much higher quotas.

Best for: people who want an editor and cloud agents from one company.

5.5 Kiro FREEMIUM

Amazon's Kiro is built around **specs**: before writing code, it helps you write requirements, a design, and a task list. That makes it a natural fit for the planning habits in this manual.

PLAN	PRICE	CREDITS PER MONTH
Free	\$0	50
Pro	\$20/mo	1,000
Pro+	\$40/mo	2,000
Pro Max / Power	\$100 / \$200 per month	5,000 / 10,000

Extra credits cost \$0.04 each on paid plans.

5.6 Zed FREEMIUM

Zed is a very fast editor with AI agents built in. The editor is free forever; **Pro** is \$10/mo with \$5 of AI usage included and usage-based billing beyond that. Best for people who like a lightweight, speedy tool.

5.7 Claude Code PAID

Anthropic's **Claude Code** is an agent you talk to in plain English. It reads your project, plans, edits files, runs commands, and checks its own work. It runs in a terminal, a desktop app, VS Code and other editors, and on the web. The *Claude Code* manual in this series covers it in depth.

PLAN	PRICE
Claude Pro	\$20/mo (\$17/mo billed yearly)
Claude Max	From \$100/mo (5x Pro usage) or \$200/mo (20x)

Best for: larger tasks you want done end to end, and working while you're away (cloud sessions).

5.8 OpenAI Codex FREEMIUM

Codex is OpenAI's coding agent, with an app, a terminal tool, editor extensions, and cloud tasks. It is included in ChatGPT plans: Free, Go (\$8/mo), Plus (\$20/mo), and Pro (from \$100/mo), with bigger plans giving more usage.

5.9 Gemini CLI FREE

Google's open-source terminal agent. Sign in with a Google account for a free tier. If you have Node.js installed, you can try it without installing anything:

TERMINAL

```
npx @google/gemini-cli
```

5.10 Devin PAID

Devin is Cognition's autonomous cloud engineer: you hand it a task, it works in its own cloud workspace, and returns a pull request. Cloud agents are included from the Devin Pro plan.

5.11 How to choose

YOU ARE...	TRY
New to editors and want a guided feel	Cursor or VS Code + Copilot
On a tight budget	VS Code + Copilot Free, Gemini CLI, Zed
A planner who likes specs	Kiro
Wanting big tasks done end to end	Claude Code or Codex
Wanting work done while you sleep	Claude Code on the web, Codex cloud, Copilot cloud agent, Devin

TIP

Editors and agents combine well. A common setup is **Cursor or VS Code for looking around**, plus **Claude Code or Codex for the heavy lifting**. They all read the same project files, and most read an `AGENTS.md` rules file.

- ☐ I know the difference between an editor and an agent
- ☐ I've picked one to try when I outgrow my builder
- ☐ I've noted its price and what the free plan allows

How Web Apps Work

You can vibe code without knowing how the web works, but a little understanding goes a long way. When you know the names of the parts, you can tell the AI exactly where a problem is, and you'll spot risky decisions before they bite.

IN THIS CHAPTER

- The restaurant model: front end, back end, database
- What APIs, hosting, and domains really are
- What happens, step by step, when someone opens your app
- The tech words AI tools use, translated
- A guided tour of a typical project folder

6.1 The restaurant model

Think of a web app as a restaurant.

RESTAURANT	WEB APP	WHAT IT DOES
Dining room, menu, waiter	Front end	What people see and click in their browser
Kitchen	Back end	Does the work out of sight: checks, calculations, emails
Pantry and recipe book	Database	Stores information so it's there tomorrow
Order slips between rooms	API	Structured messages between the parts
The building and its address	Hosting and domain	Where it all runs, and how people find it

The **front end** runs in the visitor's browser, on their phone or computer. Anything in the front end can be seen and changed by a curious visitor, which is why secrets and important checks belong in the back end.

The **back end** runs on a **server**, a computer that is always on and connected to the internet. It stores data, checks passwords, takes payments, and talks to other services.

The **database** is organized storage, usually in **tables** (like spreadsheet tabs) made of **rows** (one record each, such as one plant) and **columns** (fields, such as “name” or “watering interval”).

6.2 APIs: how programs talk

An **API** (application programming interface) is a menu of requests one program accepts from another. Your front end asks your back end, “please save this plant.” Your back end asks Stripe, “please charge this card,” or asks an AI service, “please write a reply.”

Requests and replies are usually written in **JSON**, a simple text format of labels and values:

PLANT.JSON

```
{
  "name": "Monstera",
  "water_every_days": 7,
  "last_watered": "2026-10-14"
}
```

When a service gives you an **API key**, it is a password that lets your app use that menu. Keys belong on the back end, never in front-end code.

6.3 Hosting and domains

Hosting is renting space on an always-on computer so your app is reachable. Builders host for you when you click Publish. Services like Vercel, Netlify, and Cloudflare host apps from a GitHub repository.

A **domain** is a human-friendly address like `plantpal.app`. Behind the scenes, computers find each other by numeric addresses. **DNS** (the domain name system) is the internet’s address book that turns your domain into your host’s address. **HTTPS** is the padlock in the browser: it encrypts traffic, and every modern host sets it up for free.

6.4 What happens when someone opens your app

THE JOURNEY OF ONE VISIT

- 1 The visitor types your address.**
Their browser asks DNS where `plantpal.app` lives.
- 2 DNS answers.**
It points to your host, for example Vercel.

- 3 The host sends the front end.**
HTML (structure), CSS (style), and JavaScript (behavior) arrive in the browser.
- 4 The page asks for data.**
The front end calls your back end's API: "show this person's plants."
- 5 The back end checks who's asking.**
It confirms the visitor is signed in and allowed to see the data.
- 6 The database answers.**
Only that person's rows come back.
- 7 The page fills in.**
The browser shows the list. Total time: usually under a second.

Every bug lives somewhere on this journey. "The page is blank" is usually step 3. "My data doesn't show" is usually step 4, 5, or 6. Telling the AI which step seems broken speeds up every fix.

6.5 Static sites and dynamic apps

- A **static site** is the same for every visitor: a portfolio, a menu, a brochure. It needs no back end and hosts almost anywhere for free.
- A **dynamic app** changes per visitor: accounts, saved data, payments. It needs a back end and a database.

Many modern back ends are **serverless functions** (Lovable and Supabase call them **edge functions**): small pieces of back-end code that run only when called. You don't manage a server; you pay little or nothing when nobody is using the app.

6.6 Tech words, translated

AI tools will mention these constantly. You don't need to master them; just recognize them.

WORD	WHAT IT IS
HTML	The structure of a page: headings, paragraphs, buttons
CSS	The styling: colors, fonts, spacing, layout
JavaScript	The language that makes pages interactive
TypeScript	JavaScript with extra checks that catch mistakes early
React	A popular way to build interfaces from reusable pieces ("components")
Next.js	A React framework with built-in back-end features; v0's default
Vite	A fast tool for building React apps; common in Lovable and Bolt

Tailwind CSS	A styling approach using short class names like <code>text-lg</code>
Node.js	Lets JavaScript run on a computer or server, not just in browsers
npm	The tool that installs code libraries ("packages") for Node.js
PostgreSQL	A powerful, popular database; Supabase is built on it
Expo / React Native	Tools for building phone apps with React

6.7 A tour of a project folder

Open the code view in any builder, or a project in an editor, and you'll see something like this:

PROJECT FOLDER	
plant-pal/	
├─ src/	your app's code (pages, components)
├─ public/	images and icons served as-is
├─ package.json	the project's name and list of libraries
├─ node_modules/	installed libraries (never edit; can be huge)
├─ .env	secret settings (must stay private)
├─ .gitignore	files Git should never save
└─ README.md	instructions for humans

The file `package.json` also lists **scripts**, short names for common commands. The most common is `npm run dev`, which starts your app on your own computer at an address like `http://localhost:5173`. **Localhost** means "this computer": only you can see it.

TIP

Ask your builder: "Give me a guided tour of this project's folders, as if I'm new to coding. Which files would I change to edit the home page text?" You'll be surprised how quickly the structure makes sense.

- ☐ I can name the front end, back end, database, and API in my app
- ☐ I know why secrets and important checks belong on the back end
- ☐ I can describe which step of a visit seems broken
- ☐ I've looked inside my project's folder at least once

Your First App in a Browser Builder

Time to build something. The steps below use Lovable as the example, but Bolt, v0, Base44, and Replit work almost identically. Allow about an hour, and expect it to be fun.

IN THIS CHAPTER

- Build and publish a small working app, start to finish
- Find your way around a builder's screen
- Use plan, chat, and visual editing to save credits
- Restore an earlier version when a change goes wrong

7.1 Build it

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 the "Databases and Supabase" and "Security for Vibe Coders" chapters before inviting real users.

7.2 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, secrets.

7.3 Three ways to talk to a builder

Most builders now offer modes. Using the right one saves both credits and frustration.

MODE	WHAT IT DOES	USE IT WHEN
Build (or Agent)	Changes your code	You know what you want
Plan	Investigates and writes a plan you approve	The change is big or risky
Chat (or Discuss)	Talks without touching code	You're exploring or confused

In Lovable these appear as **Build**, **Plan**, and **Chat** in the chat box. Bolt and Replit have a **Plan Mode**; Base44 has a plan mode too.

PROMPT

(Plan mode) I want to add reminders so people get an email when a plant needs water. Investigate the project and propose a plan. List any new services or settings I'll need. Don't build yet.

7.4 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. In v0 it's called **Design Mode**; in Lovable you **edit from the preview**: select an element, then edit, annotate, or comment on it.

7.5 Giving the builder lasting instructions

Builders let you store instructions they read before every message. Lovable calls this **Knowledge** (for a project or a whole workspace); others call it custom instructions or project settings.

PROJECT KNOWLEDGE

- Phone-first. Check every screen at 375px wide.
- Keep the earthy green palette. No new colors without asking.
- Never delete data without a confirmation step.
- After each change, summarize what you changed in plain English.

7.6 When a change goes wrong

Every builder keeps a **version history**. In Lovable you can preview any earlier version, **revert** to it, and **bookmark** versions you know are good. Replit calls its saved points **checkpoints**, with one-click **rollback**; Bolt has version history and backups.

UNDO A BAD CHANGE

- 1 **Stop prompting.**
Don't try to fix a bad change with more changes.
- 2 **Open version history.**
Look for a clock icon or "History."
- 3 **Preview the last good version.**
Click around to confirm it works.
- 4 **Restore it.**
Then try again with a smaller, clearer request.

7.7 Test on your phone, then share

1. **Open the published address on your phone.** Phone screens reveal cramped buttons and text that's too small.
2. **Send it to two friends.** Ask them to do one specific task, like "add a plant and mark it watered," without help from you.
3. **Watch or ask what confused them.** Their confusion is your next prompt.

7.8 First-build troubleshooting

PROBLEM	TRY THIS
Preview is blank or shows an error	Ask: "The preview shows an error. Read it and fix the cause." Or restore the last version
It built something different from what I meant	Restore, then add specifics: who, what, and an example
Data disappears on refresh	Ask it to save data in a database, not in the page
It looks bad on a phone	"Make every page work well at 375px wide. Larger tap targets."

- ☐ I built a first app and clicked every button
- ☐ I connected GitHub, published, and opened it on my phone
- ☐ I know where version history is, and I've tried restoring a version

The Vibe Coding Workflow

Random prompting produces random apps. This five-stage loop is what experienced builders do, whatever tool they use. It feels slower for the first day and much faster from then on.

IN THIS CHAPTER

- The plan → build → test → save → repeat loop
- Writing a one-page brief (a PRD)
- Breaking work into slices you can check
- A simple test script, and a rules file for the AI
- A rhythm for each building session

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.

8.1 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.

8.2 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.

i NOTE

A good slice is **visible**: when it's done, you can see or click something new. "Refactor the data layer" is not a slice you can test; "the plant list now loads from the database" is.

8.3 Order your slices wisely

ORDER	SLICE TYPE	WHY
1	Layout with sample data	Fast feedback on the look, before anything is wired up
2	The main action (add, book, buy)	Proves the core idea works
3	Saving data	Makes it real
4	Accounts and permissions	Easier once the data shape is settled
5	Polish, emails, extras	Only after the core is solid

8.4 Write a test script

A **test script** here simply means a written list of things to try after each change. It stops you forgetting to check the boring parts.

TEST-SCRIPT.MD

1. Open the app in a private window. Sign up as a new user.
2. Add a plant with a photo. Add one without a photo.
3. Mark a plant watered. Does the next date make sense?
4. Refresh. Is everything still there?
5. Sign out, sign in as a second user. Can you see user 1's plants?
(You must NOT be able to.)
6. Try it at phone width.

Run it before every publish. It takes three minutes and catches most embarrassing bugs.

8.5 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.

8.6 A rhythm for each session

ONE BUILDING SESSION

- 1 Re-read your brief and diary.**
Where did you leave off?
- 2 Pick one or two slices.**
Not five.
- 3 Plan, then build.**
Use plan mode for anything bigger than a tweak.
- 4 Run your test script.**
Fix only what's broken.
- 5 Save and note.**
Commit or bookmark, and write one line in your build diary.

8.7 When to start a fresh chat

AI conversations get worse as they get longer: the AI loses track of early details and starts contradicting itself. Start a new chat when you begin a new feature, or when the AI seems confused. Paste a short summary to bring it up to speed:

PROMPT

Context: Plant Pal is a phone-first plant-watering app (see brief.md).
Done so far: plant list, add form, database, sign-in.
Today's goal: a "Today" view showing plants due for water.
Please read the project first, then propose a plan.

- ☐ I've written a one-page brief with a "done means" section
- ☐ My next five slices are written down, in a sensible order
- ☐ I have a test script I run before each publish
- ☐ My tool has a rules file or knowledge entry
- ☐ I start fresh chats for new features

Prompts That Work

A **prompt** is the message you type to the AI. You don't need magic words, but a few patterns make a dramatic difference. This chapter collects the ones that work across every tool.

IN THIS CHAPTER

- The four questions every good prompt answers
- Copy-paste templates for common jobs
- Specs, screenshots, and “make it look like this”
- How to give feedback that the AI can act on
- A personal prompt library

9.1 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.

9.2 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?

9.3 Specs: say exactly what “done” means

For anything bigger than a tweak, write a mini **spec** (specification): a short, structured description of behavior. Specs are where most of the quality in vibe coding comes from.

PROMPT

Feature: weekly summary email.

- Who: every signed-in user who has at least one plant.
 - When: Mondays at 8am in the user's time zone.
 - Contains: plants due this week, with photos; a link to the app.
 - Not included: plants the user archived.
 - Edge cases: users with no plants get no email; an unsubscribe link turns the email off.
 - Done when: I can trigger a test send to myself and see it arrive.
- Plan this first. Tell me which email service you'd use and why.

TIP

The **edge cases** line is the most valuable one. An edge case is an unusual situation: an empty list, a very long name, two people editing at once, no internet. Ask the AI: “What edge cases should this feature handle?” before you build.

9.4 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.

- **For bugs:** screenshot the problem and circle it (any phone photo editor works).
- **For layouts:** sketch boxes on paper, photograph it, and say “build this layout.”
- **For style:** screenshot two or three sites you like.

PROMPT

Here's a screenshot of a site I like. Match its spacing, font sizes, and overall calm feel for my home page, but keep my green colors and my own content. Don't copy their text, logo, or images.

WARNING

“Make it look like” means borrowing a *feel*, not copying. Never copy another company's logo, text, photos, or distinctive branding. Besides being unfair, it can break copyright and trademark law.

9.5 A vocabulary for design

If you can name it, the AI can build it. A few useful words:

SAY THIS	TO GET
More white space	Calmer, less crowded layouts
Larger type scale	Bigger headings relative to body text
Higher contrast	Text that's easier to read
Card layout	Items in tidy boxes with rounded corners
Sticky header	A top bar that stays while you scroll
Hero section	The big first section of a landing page
Empty state	What a page shows when there's no data yet
Loading state	What shows while data is on its way
Mobile-first	Designed for phones, then stretched for desktop

9.6 Giving feedback the AI can act on

Vague feedback (“it’s not right”) leads to random changes. Use **see / expect / where**:

PROMPT

On the Today page (where), the "Watered" button is grey and tiny (see). It should be a large green button the full width of the card, easy to tap with a thumb (expect).

When the AI gets it partly right, say what to *keep* as well as what to change: “The new colors are perfect. Keep them. Only fix the button size.”

9.7 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.

9.8 Let the AI write the prompt

When you're unsure how to ask, ask the AI to help:

PROMPT

I want to add a way for users to share a plant with a friend.
Before building, ask me up to five questions to clarify what I want, then write a clear spec for my approval.

9.9 Build your own prompt library

Keep a note of prompts that worked well for you. Over time, you'll collect a personal toolkit: your best "fix this bug" wording, your favorite design prompt, your security check. The Quick Reference at the end of this manual is a starting point.

- ☐ My prompts say what, why, how it behaves, and how I'll know
- ☐ I write a mini spec, with edge cases, for bigger features
- ☐ I use screenshots for bugs and style
- ☐ I give feedback as see / expect / where
- ☐ I've started a prompt library

Version Control Safety Nets

The single most important habit in vibe coding is saving working versions. AI makes changes quickly and confidently, and sometimes those changes are wrong. With good safety nets, any mistake is one click away from undone.

IN THIS CHAPTER

- The three layers of safety: builder history, Git, and GitHub
- Git explained without jargon
- Using GitHub Desktop, the command line, or your AI to save versions
- Branches for experiments
- How to recover from a bad change

10.1 Three layers of safety

LAYER	WHAT IT IS	PROTECTS YOU FROM
Builder history	Versions or checkpoints inside Lovable, Bolt, Replit	A bad prompt in the last hour
Git	A history of your project's files, saved at moments you choose	Bad changes from any tool, on any day
GitHub	An online home for your Git history	A lost laptop, a closed account, a builder you leave

Use all three. Builder history is automatic. Git and GitHub take a minute to set up and repay that minute many times over.

10.2 Git without jargon

Git is a program that records snapshots of your project. Each snapshot is a **commit**, with a short message describing what changed. You can look back at any commit, compare it with today, or return to it.

GitHub is a website that stores your Git history online, called a **repository** (or **repo**). Builders that “sync with GitHub” are making commits for you every time the AI changes something.

A few more words:

- **Push:** send your new commits up to GitHub.
- **Pull:** bring down commits made elsewhere (by a builder, a teammate, or a cloud agent).
- **Clone:** download a full copy of a repo to your computer.
- **Branch:** a separate line of work, so you can experiment without touching the working version.
- **Pull request (PR):** a proposal to merge a branch into the main version, with the changes laid out for review.

NOTE

The *GitHub* manual in this series covers all of this in depth, with screenshots of every step. This chapter gives you just enough to stay safe.

10.3 Connect your builder to GitHub

TURN ON GITHUB SYNC

- 1 **Create a free GitHub account** at github.com, if you don't have one.
- 2 **Open your builder's GitHub setting.**
It's usually in the project's settings or an integrations menu.
- 3 **Authorize the builder.**
GitHub asks which repositories it may access. Allow just the ones it creates, if offered.
- 4 **Create the repository.**
The builder makes a new repo and pushes your code.
- 5 **Check it on github.com.**
You should see your files and a list of commits.

From now on, every change the builder makes becomes a commit. Lovable, Bolt, and Base44 offer two-way sync, so changes pushed to GitHub from elsewhere flow back into the builder too.

10.4 Saving versions yourself

When you work in an editor or with an agent on your computer, you decide when to commit. Three ways to do it:

GitHub Desktop (easiest). A free app for Windows and Mac. Changed files appear in a list; type a short summary and click **Commit to main**, then **Push origin**.

The command line. In a terminal, inside your project folder:

TERMINAL

```
git status          # what's changed?
git add -A          # include every change
git commit -m "Add Today view" # save a snapshot
git push            # send it to GitHub
```

Ask your AI agent. Agents like Claude Code and Codex can commit for you:

PROMPT

The Today view works now. Commit everything with a clear message, then push to GitHub.

TIP

Commit every time something works, not just at the end of the day. Small commits with clear messages (“Add plant photo upload”) make it easy to find exactly where something broke.

10.5 Keep secrets out of Git

A file called `.gitignore` lists files Git must never save. Your `.env` file, which holds secret keys, must be in it. Most project templates include one already; check:

.GITIGNORE

```
node_modules/
.env
.env.local
.DS_Store
```

GitHub also has **push protection**, on by default for public repositories, which blocks pushes that contain recognizable secrets such as API keys. Treat it as a seatbelt, not a reason to be careless.

10.6 Branches for experiments

When you want to try something risky, like a redesign or a new payment flow, do it on a branch.

TERMINAL

```
git switch -c try-new-design    # make and enter a new branch
# ...build and test the experiment...
git switch main                # go back to the safe version
```

If the experiment works, merge it through a pull request on GitHub. If not, delete the branch and nothing is lost. In GitHub Desktop, use **Current branch > New branch**.

10.7 Recovering from a bad change

SITUATION	WHAT TO DO
The last prompt broke things	Use the builder's history or checkpoint rollback
A file got mangled, not yet committed	<code>git restore filename</code> puts back the last committed version
A commit broke things	<code>git revert <commit-id></code> makes a new commit that undoes it
You're lost and want yesterday's version	Ask your agent: "Show me recent commits and help me return to the last one where X worked"

`git revert` is safe because it doesn't erase history; it adds an "undo" on top. Avoid commands with `--force` or `reset --hard` unless you understand exactly what they'll delete.

TERMINAL

```
git log --oneline -10          # show the last 10 commits
```

⚠ DANGER

If an AI suggests `git push --force` or `git reset --hard`, stop and ask why. These commands can permanently throw away work. There is almost always a safer way.

- ☐ My project is connected to GitHub
- ☐ I can see my commits on github.com
- ☐ `.env` is listed in `.gitignore`
- ☐ I commit after each working slice
- ☐ I know how to restore a version in my builder and with Git

Project 1: A Portfolio Site

Your first complete build is a personal portfolio: a few pages that show who you are and what you do. It needs no database and no logins, so you can focus on the workflow. We'll use **v0**, which excels at good-looking pages and deploys to Vercel in one click. Lovable, Bolt, and Figma Make work just as well.

IN THIS PROJECT

- Gather content before you prompt
- Build a multi-section site in slices
- Add page titles, descriptions, and a sharing image
- Publish and connect a custom domain

Time	1–2 hours
Cost	Free; a custom domain is an optional yearly fee
Tools	v0 (free plan), Vercel (free Hobby plan for personal sites)
You'll learn	Prompting for design, iterating, deploying, domains

NOTE

Vercel's free **Hobby** plan is for personal, non-commercial use. A personal portfolio is fine. For a business site, use the Pro plan or another host such as Netlify or Cloudflare.

11.1 Step 1: Gather your content

The AI can invent placeholder text, but your site will feel generic until you give it the real thing. Collect these in a document first:

- Your name and a one-line description
- A short “about” paragraph (3–5 sentences)
- 3–6 projects or work samples
- A photo of each project
- A photo of you (optional)
- How to contact you
- Links: LinkedIn, GitHub, Instagram
- Three adjectives for the feel

11.2 Step 2: Describe the whole site, build the first slice

PROMPT

I'm building a personal portfolio site. I'm Sam Rivera, a freelance illustrator who draws for children's books and small brands. Sections: hero with my name and tagline, selected work (a grid of 6 projects), about me, and contact. Feel: warm, playful, lots of white space, hand-drawn accents. Mobile-first. Start with just the hero section and the work grid, using placeholder images. I'll give you real content next.

Look at the preview on desktop and in the phone view. Don't fix everything at once: pick the biggest problem first.

11.3 Step 3: Pour in your real content

PROMPT

Replace the placeholder text with my real content below. Keep the layout. For each project card show the title, a one-line description, and the image. Clicking a card opens a larger view with the full text.

[paste your about text, project titles and descriptions here]

Upload your images in the chat (most builders accept image uploads) and ask the AI to use them in the right cards.

11.4 Step 4: Polish in small passes

Do one pass at a time, checking the preview after each:

1. **Typography:** “Make headings bolder and larger; increase line spacing in paragraphs.”
2. **Color:** “Use a warm cream background with a coral accent for links and buttons.”
3. **Motion:** “Add a subtle fade-in as sections scroll into view. Keep it gentle.”
4. **Phone check:** “Check every section at 375px wide and fix anything cramped.”

Use v0’s **Design Mode** for small text and color tweaks; it’s faster than prompting.

11.5 Step 5: Make it shareable and findable

When someone shares your link, apps show a title, a description, and an image. Search engines use the same information.

PROMPT

Add proper page metadata: a page title "Sam Rivera — Illustrator", a 150-character description, and a social sharing image using my hero illustration. Add descriptive alt text to every image for screen readers. Add a favicon using my initials.

Alt text is a short written description of an image. Screen readers (software that reads pages aloud for blind and low-vision users) speak it, and search engines read it too.

11.6 Step 6: Add a contact option

The simplest option is a link that opens the visitor’s email app:

PROMPT

In the contact section, add a large "Email me" button that opens a new email to hello@samrivera.art with the subject "Project inquiry". Also add icon links to my Instagram and LinkedIn.

If you’d rather have a form, see Project 2.

11.7 Step 7: Publish

DEPLOY FROM V0

- 1 Click **Publish** (or **Deploy**) in v0.

- 2 **Connect your Vercel account** when asked. v0 creates a Vercel project for you.
- 3 **Wait a minute.**
Vercel builds the site and gives you an address ending in `.vercel.app`.
- 4 **Open it on your phone and computer.**
Click every link.
- 5 **Connect GitHub** in v0 if you haven't, so your code is backed up.

11.8 Step 8: Add your own domain (optional)

CONNECT A DOMAIN ON VERCEL

- 1 **Buy a domain** from a registrar such as Cloudflare, Namecheap, or Porkbun.
- 2 **In Vercel, open your project**, then `Settings` > `Domains`, and add the domain.
- 3 **Copy the DNS records Vercel shows you** (usually an A record and a CNAME for `www`).
- 4 **Add them at your registrar**, in its DNS settings.
- 5 **Wait.**
It can take minutes to a few hours. Vercel shows a green check and sets up HTTPS automatically.

The “Deployment and Domains” chapter explains DNS records in more detail.

11.9 If something goes wrong

PROBLEM	FIX
Site looks generic	Give real content, a reference screenshot, and three adjectives
Layout breaks on phone	Paste a phone screenshot and say what's wrong
Deploy failed	Copy the build error from Vercel into v0: “The deploy failed with this error. Fix it.”
Domain not working after hours	Check records at your registrar match Vercel's exactly; remove old conflicting records

- ☐ Real content and images, no placeholder text left
- ☐ Looks good at phone width and desktop width
- ☐ Alt text, page title, description, and sharing image set
- ☐ Contact button works; published and connected to GitHub

Project 2: A Landing Page with a Sign-Up Form

A **landing page** is a single page with one goal: getting visitors to take one action, such as joining a waitlist. This project adds your first piece of back end: a form whose answers are saved in a database, plus an optional confirmation email.

IN THIS PROJECT

- Write a landing page that persuades
- Save form sign-ups in a database
- Protect the form from spam, and handle personal data properly
- Send a confirmation email (optional)
- A no-database alternative with Netlify Forms

Time	2–3 hours
Cost	Free tiers are enough to start
Tools	Lovable with Lovable Cloud (or Bolt Cloud, Base44, Replit)
You'll learn	Databases, forms, secrets, email services

12.1 Step 1: Plan the page

Good landing pages follow a familiar pattern. Fill this in before prompting:

LANDING-BRIEF.MD

Product: "Bloomcast", a weekly email telling you which plants to water, feed, or repot this week.
Audience: busy houseplant owners.
Goal: join the waitlist (email + first name).
Sections:
1. Hero: headline, one-line promise, sign-up form.
2. Three benefits with icons.
3. How it works: three steps.
4. FAQ: four questions.
5. Final call to action with the form again.
Feel: fresh, botanical, confident.

12.2 Step 2: Build the page with a placeholder form

PROMPT

Build a landing page from this brief: [paste brief]. Make the form look real but don't save anything yet. Mobile-first. Use a plant illustration style for icons.

Iterate on the words as much as the design. Ask: "Rewrite the headline five ways, focused on saving time. Show them to me before changing anything."

12.3 Step 3: Save sign-ups to a database

PROMPT

Now make the form work. Save each sign-up (first name, email, created time) to a "signups" table in the built-in database. Validate the email format. Show a friendly thank-you message after submitting. If the same email signs up twice, don't create a duplicate; just show the thank-you message again. Visitors must NOT be able to read the signups table; only I can, from the dashboard.

That last sentence is crucial. A sign-up form needs to let anyone **add** a row, but nobody but you should **read** the list. This is an access rule, and in Supabase-based back ends (including Lovable Cloud) it is enforced by **row-level security**. More on that in the "Databases and

Supabase” chapter.

CHECK IT WORKS

- 1 **Submit the form** with your own email.
- 2 **Open the database view** in your builder (in Lovable, Cloud’s **Database** view).
- 3 **Find your row** in the signups table.
- 4 **Submit again** with the same email. There should still be one row.
- 5 **Run the builder’s security scan** and fix anything marked critical.

12.4 Step 4: Stop spam

Public forms attract bots. Two cheap defenses:

PROMPT

Add spam protection to the sign-up form: a hidden "honeypot" field that real people never see (reject submissions that fill it in), and a limit of 5 submissions per minute from the same visitor.

A **honeypot** is a hidden field that only bots fill in. **Rate limiting** caps how often someone can submit. If spam persists, ask about adding a CAPTCHA service.

12.5 Step 5: Handle personal data properly

An email address is personal data. Before you go live:

- **Say what you’ll do with it.** Add a line under the form: “We’ll email you when Bloomcast launches. No spam; unsubscribe anytime.”
- **Add a privacy policy page.** Ask the AI for a plain-English draft, then read it carefully and adjust it to what you really do.
- **Collect only what you need.** Name and email, not phone numbers “just in case.”

12.6 Step 6: Send a confirmation email (optional)

To send email from an app, you use an **email service** such as **Resend**. Its free plan allows 3,000 emails a month (up to 100 a day), plenty for a waitlist.

ADD A CONFIRMATION EMAIL

- 1 **Create a Resend account** and add your domain (email from your own domain is far less likely to land in spam).

- 2 **Create an API key** in Resend.
- 3 **Store the key as a secret** in your builder, named `RESEND_API_KEY`. In Lovable, use the Cloud **Secrets** manager, never the `.env` file.

- 4 **Ask the AI to send the email from the back end:**

PROMPT

After a successful sign-up, send a short confirmation email using Resend from a back-end function. Use the `RESEND_API_KEY` secret. Never expose the key to the browser. Subject: "You're on the Bloomcast list". Friendly, two sentences, unsubscribe link.

- 5 **Test with your own address**, and check your spam folder.

12.7 Alternative: Netlify Forms, no database needed

If your page is a simple static site hosted on Netlify, Netlify can collect form submissions for you. You add a `data-netlify="true"` attribute to the form, and submissions appear in your Netlify dashboard. Ask your AI:

PROMPT

This site will be hosted on Netlify. Make the sign-up form a Netlify Form (`data-netlify="true"`), with a honeypot field and a thank-you page.

Form submissions use some of your Netlify plan's monthly credits, so check usage if your page gets busy.

12.8 Step 7: Launch and measure

1. **Publish** and test the live form once more.
2. **Add simple analytics.** Many hosts include visitor analytics; ask the AI to add your host's built-in option or a privacy-friendly one.
3. **Export your list** as a CSV (a spreadsheet file) from the database view when you're ready to email people.

PROBLEM	FIX
Form says "error" with no details	Ask: "Show the real error in the browser console and fix the cause"
Sign-ups don't appear in the table	Check the access rule allows inserting; ask the AI to test a submission
Visitors can read the list	Stop and fix the access rule now; re-run the security scan
Emails land in spam	Verify your domain in the email service; send from your domain

- ☐ Page has one clear goal and one form
- ☐ Sign-ups save, with no duplicates
- ☐ Visitors cannot read the sign-up list
- ☐ Honeypot and rate limit are in place
- ☐ A privacy note and policy page exist
- ☐ Any email API key is stored as a back-end secret

Project 3: A To-Do App with a Database and Logins

This is the project where your app becomes a real, multi-user product. Each person signs in and sees only their own tasks. You'll set up **Supabase**, a popular hosted database with logins built in, so you understand what builders do behind the scenes.

IN THIS PROJECT

- Design a simple data model
- Create a Supabase project and connect it to a builder
- Add sign-in with magic links
- Lock down data with row-level security, and prove it works
- Configure redirect URLs for the live site

Time	3–4 hours
Cost	Free (Supabase Free plan, builder free tier)
Tools	Lovable (or Bolt) plus your own Supabase project
You'll learn	Tables, auth, row-level security, testing with two users

NOTE

Lovable Cloud and Bolt Cloud are built on the same technology and do most of this for you. Doing it once with your own Supabase project teaches you what's going on, and gives you a project you fully control.

13.1 Step 1: Design the data

Before prompting, decide what you're storing. For a to-do app, one table is enough:

COLUMN	TYPE	MEANING
<code>id</code>	unique ID	Identifies each task
<code>user_id</code>	user reference	Who owns the task
<code>title</code>	text	What to do
<code>done</code>	true/false	Finished or not
<code>due_date</code>	date (optional)	When it's due
<code>created_at</code>	timestamp	When it was added

The `user_id` column is what makes a multi-user app possible: every row records its owner.

13.2 Step 2: Create a Supabase project

SET UP SUPABASE

- 1 **Sign up** at supabase.com (signing in with GitHub is easiest).
- 2 **Create a new project.**
Pick a name, a region near your users, and a strong database password. Save the password in a password manager.
- 3 **Wait a minute** while the project starts.
- 4 **Find your keys** under [Settings](#) > [API Keys](#). You'll see a **publishable key** (starts `sb_publishable_`) and a **secret key** (starts `sb_secret_`).

⚠ DANGER

The **publishable key** is safe in your front end: it only reaches what your security rules allow. The **secret key** bypasses all security rules. It must never appear in front-end code, a chat, a screenshot, or GitHub. Older projects show legacy keys called `anon` and `service_role`; Supabase is retiring these by the end of 2026.

i NOTE

On the Free plan, Supabase pauses projects after a week of inactivity and allows two active projects. Paused projects can be restored from the dashboard.

13.3 Step 3: Connect your builder

In Lovable, use the Supabase integration (Lovable's docs call it **Connect to Supabase**); Bolt has a similar option in its database settings. Authorize it and pick the project you just made. Then:

PROMPT

Build a to-do app using the connected Supabase project. Create a "todos" table with: id, user_id (the owner), title, done, due_date (optional), created_at. Show the signed-in user's tasks, let them add, tick off, edit, and delete tasks. Clean, minimal design. Don't add sign-in yet.

13.4 Step 4: Add sign-in

PROMPT

Add sign-in with email magic links using Supabase Auth. A magic link is a one-time sign-in link sent by email, so there are no passwords. Show a simple sign-in page to signed-out visitors. New tasks must be saved with the signed-in user's id. Add a sign-out button in the header.

13.5 Step 5: Lock down the data

Now the crucial part. **Row-level security** (RLS) is a set of rules inside the database that decide which rows each user may read or change. With RLS on and the right policies, even someone who pokes at your app with technical tools can't see other people's tasks.

PROMPT

Enable row-level security on the todos table and add policies so that signed-in users can only select, insert, update, and delete their own rows (where user_id matches their user id). Show me the SQL you're running and explain each policy in one sentence.

The AI should produce something like this. **SQL** is the language databases understand:

TODOs-RLS.SQL

```
alter table public.todos enable row level security;

create policy "Users can view their own todos"
  on todos for select to authenticated
  using ((select auth.uid()) = user_id);

create policy "Users can add their own todos"
  on todos for insert to authenticated
  with check ((select auth.uid()) = user_id);
```

Similar policies cover `update` and `delete`. In plain English: `auth.uid()` is “the ID of the person asking,” and each rule says “only if that matches the row’s owner.” The `with check` part stops someone creating a task that belongs to someone else.

13.6 Step 6: Prove it with two users

Never assume security works. Test it.

THE TWO-USER TEST

- 1 **Open your app in a normal window** and sign in as user A (your main email).
- 2 **Add three tasks** whose names start with “A:”.
- 3 **Open a private window** and sign in as user B (a second email address).
- 4 **Check user B sees none of A’s tasks.**
Add a “B:” task.
- 5 **Go back to user A** and confirm the B task isn’t visible.
- 6 **Check Supabase’s Security Advisor** in the dashboard, and fix any warnings with the AI’s help.

13.7 Step 7: Publish and fix redirects

Magic links send people back to your app after they click. Supabase only redirects to addresses you approve.

CONFIGURE URLS FOR THE LIVE SITE

- 1 **Publish your app** and copy its live address.
- 2 **In Supabase, open** `Authentication` > `URL Configuration`.

- 3 **Set Site URL** to your live address (it starts as `http://localhost:3000`).
- 4 **Add Redirect URLs** for your live address and your builder's preview address.
- 5 **Test a magic link** on the live site.

13.8 If something goes wrong

PROBLEM	LIKELY CAUSE AND FIX
Tasks list is empty after enabling RLS	No <code>select</code> policy yet, or <code>user_id</code> isn't being saved. Ask the AI to check both
"new row violates row-level security policy"	The insert doesn't set <code>user_id</code> to the signed-in user
Magic link opens localhost or an error page	Update Site URL and Redirect URLs
Magic link email never arrives	Check spam; the built-in email sender has low limits, so set up custom email (SMTP) for real use
App stopped working after a quiet week	The free project paused; restore it in the dashboard

- ☐ Data model written down before building
- ☐ Secret key never pasted anywhere public
- ☐ RLS enabled, with select, insert, update, and delete policies
- ☐ Two-user test passed
- ☐ Site URL and Redirect URLs point at the live site

Project 4: A Chrome Extension

A **browser extension** is a small add-on that changes how your browser works: a button in the toolbar, a tweak to certain websites, a handy popup. Extensions are perfect vibe-coding projects: small, useful, and entirely yours. This project uses an AI agent on your computer (Claude Code, Codex, or Cursor), because browser builders don't make extensions.

IN THIS PROJECT

- How extensions are put together
- Build a “Read Later” extension with an AI agent
- Load and test it in Chrome
- Iterate, add icons, and optionally publish to the Chrome Web Store

Time	1–2 hours
Cost	Free (publishing has a one-time registration fee)
Tools	Google Chrome; Claude Code, Codex, or Cursor
You'll learn	Working with local files, manifests, permissions, reloading

14.1 How an extension is put together

An extension is just a folder of files:

FILE	WHAT IT DOES
<code>manifest.json</code>	The ID card: name, version, permissions, which files to use
<code>popup.html</code>	The little window that opens when you click the toolbar icon
<code>popup.js</code>	The popup's behavior
<code>popup.css</code>	The popup's look
<code>icons/</code>	Toolbar and store icons in a few sizes

Chrome extensions use **Manifest V3**, the current format. If an AI ever writes

```
"manifest_version": 2
```

, ask it to use version 3.

Permissions are what your extension asks the browser to allow, such as storing data or reading the current tab. Ask for as few as possible: users see the list when they install, and fewer permissions means less risk.

14.2 Step 1: Make a folder and open your agent

SET UP

- 1 **Create a folder** called `read-later` somewhere easy, like your Documents folder.
- 2 **Open it in your tool.**
In Cursor or VS Code: **File** > **Open Folder**. For Claude Code, open a terminal in that folder and run `claude`, or use the desktop app and choose the folder.
- 3 **Optional but wise:**
ask the agent to set up Git: “Initialize a Git repository here and commit after each working step.”

14.3 Step 2: Describe the extension

PROMPT

Build a Chrome extension (Manifest V3) called "Read Later".
Clicking the toolbar icon opens a popup with:

- a "Save this page" button that stores the current tab's title and address;
- a list of saved pages (newest first), each clickable to open it in a new tab, with a small "remove" button.

Store the list with `chrome.storage.local`. Use only the permissions it truly needs and explain each one. Clean, compact design, 320px wide. No external libraries, no tracking, no network requests.

The agent will create the files. Its manifest should look something like this:

MANIFEST.JSON

```
{
  "manifest_version": 3,
  "name": "Read Later",
  "version": "1.0",
  "description": "Save pages to read later.",
  "action": { "default_popup": "popup.html" },
  "permissions": ["storage", "activeTab"],
  "icons": { "128": "icons/icon128.png" }
}
```

`storage` lets it save your list; `activeTab` lets it read the current tab's title and address, but only when you click the extension.

14.4 Step 3: Load it into Chrome

LOAD AN UNPACKED EXTENSION

- 1 **Open Chrome** and type `chrome://extensions` in the address bar.
- 2 **Turn on Developer mode** with the toggle at the top right.
- 3 **Click Load unpacked** and choose your `read-later` folder.
- 4 **Pin it.**
Click the puzzle-piece icon in Chrome's toolbar and pin Read Later.
- 5 **Try it.**
Open any article, click the icon, and press "Save this page."

14.5 Step 4: Iterate

Every time the agent changes files, Chrome needs to reload the extension: on `chrome://extensions`, click the circular **reload** arrow on its card, then reopen the popup.

Ideas for next slices, one at a time:

1. "Add a search box to filter saved pages."
2. "Show how many pages are saved as a number on the toolbar icon."
3. "Add a keyboard shortcut to save the current page."
4. "Add an 'export as text' button that copies the list to the clipboard."

14.6 Finding errors in an extension

Extensions have their own error areas:

- On `chrome://extensions`, an **Errors** button appears on the card when something fails. Click it and copy the message to your agent.
- To see the popup's console, right-click inside the open popup and choose **Inspect**.

PROMPT

The popup opens but saving does nothing. Here's the error from the Errors button on `chrome://extensions`: [paste]. Find the cause and fix it.

14.7 Step 5: Icons and polish

PROMPT

Create simple icons for the extension at 16, 32, 48, and 128 pixels: a bookmark shape in deep purple on a transparent background. Add them to the manifest. Then review the popup for accessibility: keyboard navigation, focus outlines, and readable contrast.

14.8 Step 6: Publish to the Chrome Web Store (optional)

You can keep using an unpacked extension forever. To share it publicly:

PUBLISH

- 1 **Register as a Chrome Web Store developer** in the developer dashboard and pay the one-time registration fee. Google suggests a dedicated email address for this account.
- 2 **Zip the extension folder** (ask your agent to create the ZIP, excluding Git files).
- 3 **Upload it** in the dashboard and fill in the listing: description, screenshots, category.
- 4 **Complete the privacy section** honestly. Read Later stores data only on the user's computer and sends nothing anywhere; say so.
- 5 **Submit for review.**
Reviews can take a few days. Fewer permissions usually means a smoother review.

PROBLEM	FIX
"Manifest file is missing or unreadable"	You picked the wrong folder; choose the one containing <code>manifest.json</code>
Changes don't show up	Click reload on the extension's card, then reopen the popup
Saved list disappears	Make sure it uses <code>chrome.storage</code> , not variables in the popup
Store review rejected	Read the reason; usually unused permissions or an unclear privacy statement

- ☐ Extension uses Manifest V3 and minimal permissions
- ☐ Loads unpacked without errors
- ☐ Saved pages survive closing and reopening Chrome
- ☐ Icons at all sizes
- ☐ Code committed to Git

Project 5: A Simple Mobile App

Phone apps feel like a big step up, but modern tools make a simple one surprisingly approachable. You'll build a habit tracker with **Expo**, a free toolkit for making iPhone and Android apps with React Native, and test it on your own phone with the **Expo Go** app.

IN THIS PROJECT

- Two routes: a browser builder (Bolt or Replit) or an agent on your computer
- Build a habit tracker that stores data on the phone
- Test it on a real phone with Expo Go
- What it takes to publish to the App Store and Google Play

Time	2–3 hours
Cost	Free to build and test; store accounts cost extra
Tools	Bolt or Replit, or Claude Code/Codex/Cursor with Node.js
You'll learn	Mobile screens, on-device storage, testing on a phone

15.1 Route A: In a browser builder

Bolt and Replit both build Expo apps from a prompt and show a QR code you scan with Expo Go. This is the easiest route: nothing to install on your computer.

15.2 Route B: With an agent on your computer

You'll need **Node.js** installed (from nodejs.org; choose the LTS version). Then, in a terminal, create a starter project and start it:

TERMINAL

```
npx create-expo-app@latest habit-hero  
cd habit-hero  
npx expo start
```

A QR code appears in the terminal. Open your agent in the `habit-hero` folder for the building steps below.

15.3 Step 1: Install Expo Go on your phone

Install **Expo Go** from the App Store or Google Play. On an iPhone, sign in to Expo Go with the same Expo account you use on your computer (run `npx expo login` there), or the app may refuse to open your project.

15.4 Step 2: Describe the app

PROMPT

Build a mobile habit tracker called "Habit Hero" with Expo.
Screens:
1. Today: a list of my habits with a big tap-to-complete circle each.
2. Add habit: name, emoji, and which days of the week.
3. Stats: current streak and a 30-day grid per habit.
Store data on the device so it survives closing the app.
Large tap targets, friendly colors, works in light and dark mode.
Start with the Today screen using sample habits.

15.5 Step 3: Test on your phone

Scan the QR code: with the Camera app on iPhone, or inside Expo Go on Android. The app opens on your phone, and it updates automatically when files change.

TIP

If the QR code won't connect (common on office or café Wi-Fi), stop the server and run `npx expo start --tunnel`, then scan the new code.

15.6 Step 4: Build the rest in slices

1. “Make the Add habit screen work and save to the device.”
2. “Calculate streaks correctly, counting only scheduled days.”
3. “Add the Stats screen with the 30-day grid.”
4. “Add a daily reminder notification at a time I choose.”

Test each slice on the phone itself: tap with your thumb, rotate the screen, close and reopen the app.

15.7 Step 5: Publishing (when you're ready)

Testing with Expo Go is free and enough for personal use. Putting an app in the stores is a bigger step:

STORE	ACCOUNT COST	WHAT TO EXPECT
Apple App Store	\$99 a year (Apple Developer Program)	Review against Apple's guidelines; screenshots, privacy details
Google Play	\$25 one-time	Identity checks; testing requirements for new accounts

Replit can build, sign, and upload iOS apps for you, and Bolt has an Expo publishing guide. With your own project, ask your agent to walk you through Expo's build service.

NOTE

Expo Go supports a specific Expo SDK version. If it says your project is incompatible, ask the AI: “Make this project use the Expo SDK version my Expo Go app supports.”

PROBLEM	FIX
QR code scans but nothing loads	Same Wi-Fi for phone and computer, or use <code>--tunnel</code>
Data disappears on restart	Ask it to use on-device storage (such as AsyncStorage)
Red error screen on the phone	Photograph or copy the message and paste it to the AI
Looks wrong on notch phones	"Respect safe areas on all screens"

- ☐ App runs on my own phone through Expo Go
- ☐ Data survives closing and reopening the app
- ☐ Tested with my thumb, in light and dark mode
- ☐ Code committed to Git

Project 6: An AI Chatbot App

Many people’s first “real” product idea is an AI assistant for something they know well: a recipe helper, a study buddy, a customer FAQ bot. This project builds one safely, with your AI key hidden on the server and limits that stop a surprise bill.

IN THIS PROJECT

- How an AI chatbot app is wired together
- Get an API key and store it as a back-end secret
- Build a chat interface with a custom personality
- Add usage limits and spending caps

Time	2–3 hours
Cost	Builder free tier, plus pay-per-use AI charges (often cents while testing)
Tools	Lovable, Bolt, Replit, or Google AI Studio; an AI provider account
You’ll learn	Server functions, secrets, AI APIs, rate limits

16.1 How it’s wired

1. The visitor types a message in your **front end**.
2. The front end sends it to **your back end** (a server or edge function).
3. Your back end adds your instructions and your **secret API key**, and calls the AI provider.
4. The reply comes back, often **streamed** word by word so it appears as it’s written.

⚠ DANGER

The API key must live only in step 3. If it is in the front end, anyone can copy it from their browser and run up charges on your account.

16.2 Step 1: Choose an AI provider and get a key

You pay providers per **token** of text processed. Popular choices:

- **Anthropic (Claude):** keys at platform.claude.com.
- **OpenAI:** keys at platform.openai.com.
- **Google (Gemini):** keys in Google AI Studio, which also builds Gemini apps for you with the key kept server-side.

Some builders, such as Lovable Cloud, include built-in AI features billed through your plan's credits, with no key needed. That's the simplest option for a first bot.

PREPARE YOUR ACCOUNT

- 1 **Create an account** with the provider and add a small amount of credit or a payment method.
- 2 **Set a monthly spending limit** in its billing or limits settings. Start low.
- 3 **Create an API key** and copy it once. Treat it like a password.
- 4 **Store it as a secret** in your builder, for example `ANTHROPIC_API_KEY` or `OPENAI_API_KEY`.

16.3 Step 2: Build the chat

PROMPT

Build "Kitchen Buddy", a cooking assistant chat app. A clean chat screen: messages, input box, send button, and a "new chat" button. Call the AI from a back-end function using the secret `ANTHROPIC_API_KEY`. Never send the key to the browser. Stream replies. Keep the conversation history for the current chat only.

16.4 Step 3: Give it a personality and boundaries

The **system prompt** is the hidden instruction that shapes every reply. Ask the AI to put it on the back end:

PROMPT

Add this system prompt on the server: "You are Kitchen Buddy, a friendly home-cooking assistant. Give practical recipes and tips. Keep answers under 200 words unless asked for more. Always mention common allergens. If asked about anything unrelated to cooking, politely steer back to food. Never give medical advice."

Test it: ask a cooking question, an off-topic question, and try to trick it ("ignore your instructions"). Adjust the wording until you're happy.

16.5 Step 4: Add limits

AI features are the most common source of surprise bills. Add three layers:

PROMPT

Add protections: require sign-in to chat; limit each user to 30 messages per day and show a friendly message when they hit it; cap message length at 2,000 characters; and log daily usage per user to a table I can view.

Together with the provider's spending limit, a bug or a bot can no longer cost you more than you've decided.

16.6 Step 5: Test, publish, watch

1. **Run your test script** including the limit: send 31 messages on a test account.
2. **Publish**, then check the provider's usage page the next day.
3. **Read some conversations** (with users' knowledge; mention it in your privacy policy) to improve your system prompt.

PROBLEM	FIX
"401" or "invalid API key"	Re-copy the key into the secret; check the variable name matches
"429" or "rate limit"	You or the provider hit a limit; wait, or raise your provider tier
Replies stop mid-sentence	Ask the AI to raise the maximum reply length or fix streaming
Key found in front-end code	Revoke it at the provider immediately, make a new one, move it to secrets

- ☐ API key stored only as a back-end secret
- ☐ Provider spending limit set
- ☐ Sign-in, daily message limit, and length cap in place
- ☐ System prompt tested against off-topic and trick questions

Project 7: An Internal Tool or Dashboard

Internal tools are apps used inside a business or club rather than by the public: an inventory tracker, a booking board, a sales dashboard. They are where vibe coding saves the most time, because they replace messy spreadsheets. This project builds an order dashboard in **Base44**, which wires up data, logins, and permissions for you. Replit and Lovable work well too.

IN THIS PROJECT

- Turn a spreadsheet into an app with real data
- Build charts and summary numbers
- Give different people different permissions
- Test safely with a separate test database

Time	2–4 hours
Cost	Free plan to try; a paid plan for daily team use
Tools	Base44 (or Replit, Lovable)
You'll learn	Importing data, roles, permissions, charts

17.1 Step 1: Start from your real spreadsheet

Export your current sheet as a **CSV** file (in Google Sheets, [File](#) > [Download](#) >

[Comma-separated values](#) ; in Excel, **Save As** and choose CSV). Remove anything sensitive you don't need, like personal phone numbers.

17.2 Step 2: Describe the tool

PROMPT

Build an internal "Order Desk" for my bakery. I'll upload a CSV of orders with columns: date, customer, item, quantity, price, status (new, baking, ready, collected).

Pages:

1. Dashboard: today's orders count, this week's revenue, a chart of revenue per day for 30 days, and orders by status.
2. Orders: a searchable, filterable table; click a row to edit status.
3. New order form.

Use plan mode first and show me the data structure before building.

Review the proposed tables. Check the types: money as numbers, dates as dates. Then let it build and import your CSV.

17.3 Step 3: Roles and permissions

Decide who can do what before inviting anyone:

ROLE	CAN SEE	CAN CHANGE
Owner (you)	Everything, including revenue	Everything
Staff	Orders	Order status, new orders
Viewer	Today's orders only	Nothing

PROMPT

Add roles: owner, staff, viewer, with these permissions: [paste table]. Enforce them on the back end, not just by hiding buttons. Only invited people with company email addresses can sign in.

Base44 lets you review **data permissions** in plain English in its settings; read them after the AI changes anything. Then run its **security scan**.

17.4 Step 4: Test without risking real data

Base44 can use a **separate test database**, so experiments don't touch live orders. Turn it on while building, and switch back when you're confident. In other tools, ask the AI to "seed some fake test data" and to keep a backup export of the real data.

TEST AS EACH ROLE

- 1 **Invite a second email of yours** as staff. Sign in from a private window.
- 2 **Check the dashboard revenue is hidden** for staff.
- 3 **Change an order's status.**
It should work.
- 4 **Try deleting an order.**
It should be refused.

17.5 Step 5: Make it useful daily

- "Add a 'mark ready' button that emails the customer." (See the email section of the "Logins, Payments, and Email" chapter.)
- "Every morning at 7am, email me yesterday's summary."
- "Add an export to CSV button on the Orders page."

PROBLEM	FIX
Imported numbers are wrong	Currency symbols or commas in the CSV; ask the AI to clean them on import
Charts are empty	Dates imported as text; ask it to convert the date column
Staff can see revenue	Permission enforced only in the page; insist on back-end rules
Team can't sign in	Check invitations and allowed email domains

- ☐ Real data imported and checked against the spreadsheet
- ☐ Roles enforced on the back end, tested from each role
- ☐ Security scan run, critical issues fixed
- ☐ A backup export of the data saved

Databases and Supabase

Sooner or later your app needs to remember things. This chapter explains databases in plain terms, then digs into Supabase, the database behind many builders, and the security rules that keep data safe.

IN THIS CHAPTER

- Tables, rows, columns, and relationships
- Where your builder keeps data
- Supabase plans, keys, and row-level security
- Backups and checking your data

18.1 Front end, back end, database

- 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.

18.2 Relationships between tables

Real apps have several tables that point at each other. A recipe app might have:

TABLE	HOLDS	POINTS TO
<code>users</code>	Accounts	–
<code>recipes</code>	Title, steps, photo	The user who wrote it
<code>ingredients</code>	Name, amount	The recipe it belongs to
<code>favorites</code>	Who liked what	A user and a recipe

That “points to” link is a **foreign key**: a column holding another row’s ID. Ask the AI to “show me the data model as a table list with relationships” whenever it adds tables; it’s the best way to catch a muddled design early.

TIP

Changing the shape of a database once real data is in it is harder than changing a page. Spend extra time getting the tables right early, using plan mode.

18.3 Supabase in brief

	FREE	PRO
Price	\$0	From \$25/mo
Active projects	2	First included; more from \$10/mo
Database size	500 MB	Larger, scales up
Monthly active users	50,000	100,000, then pay per user
Catch	Pauses after 1 week of inactivity	Spend cap on by default

18.4 The two kinds of key

KEY	STARTS WITH	WHERE IT CAN GO
Publishable	<code>sb_publishable_</code>	Front end, mobile apps, public code; limited by RLS
Secret	<code>sb_secret_</code>	Back end only; bypasses RLS completely

Find them in [Settings](#) > [API Keys](#). Older tutorials and AI answers mention `anon` and `service_role` keys (long strings starting `eyJ`): these are the legacy versions, being retired by the end of 2026.

18.5 Row-level security, again

Row-level security (RLS) is the database deciding, row by row, who may read or change what. Without it, a table in Supabase's public schema can be read and written by anyone who has your publishable key, which is everyone who visits your app. Rules of thumb:

1. **RLS on for every table.** No exceptions.
2. **A policy for each action you need:** select, insert, update, delete.
3. `using (true)` means “everyone”. Only for genuinely public data, like published blog posts.
4. **Test with two accounts**, as in Project 3.
5. **Check the Security Advisor** in the dashboard after every database change.

PROMPT

Audit the database. For every table, tell me: is RLS on, what policies exist in plain English, and who can read or change which rows. Flag anything readable by signed-out visitors.

18.6 Looking after your data

- **Look at it.** Every builder has a table view. Glance at real rows weekly.
 - **Back it up.** Export important tables as CSV regularly, and before big changes.
 - **Never test destructive changes on live data.** Use a test database, or a copy.
 - **Store files, not just text.** Photos and uploads go in **storage buckets**, with their own public/private settings.
- ☐ I can list my app's tables and how they connect
 - ☐ RLS is on for every table, with tested policies
 - ☐ My secret key is only on the back end
 - ☐ I have a recent export of important data

Logins, Payments, and Email

Three features turn a project into a product: knowing who's using it, taking money, and sending messages. Each one has a proven service you should lean on instead of letting the AI invent its own.

IN THIS CHAPTER

- Sign-in options and how to choose
- Taking payments safely with Stripe
- Sending email that actually arrives
- API keys and environment variables

19.1 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.

METHOD	GOOD FOR	WATCH OUT FOR
Magic link	Simple apps; no passwords to forget	Depends on email arriving quickly
Email + password	Familiar to everyone	Needs password reset; enforce decent passwords
Google / Apple sign-in	Fast, trusted	Extra setup in Google’s or Apple’s console
One-time code	Phone-friendly	Codes must expire quickly

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: it is what row-level security enforces. **Authorization** is the related idea of deciding what a signed-in person may do, like “staff can edit orders but not delete them.”

 TIP

After adding sign-in, always set the live address in your auth settings (in Supabase, [Authentication](#) > [URL Configuration](#)). Otherwise sign-in links send people back to `localhost` .

19.2 Payments with Stripe

Use **Stripe** (or your builder’s payments integration) for anything involving money. Stripe handles card details so they never touch your app. Standard US pricing is 2.9% + 30¢ per successful card charge, with no monthly fee. Lovable offers built-in Stripe or Paddle payments; Bolt and Base44 have Stripe setup guides.

OPTION	EFFORT	USE IT FOR
Payment Links	No code: create in the Stripe dashboard, paste a link	Selling one product or a simple subscription
Checkout	A Stripe-hosted payment page your app opens	Most apps
Builder integration	Done by chat	Fastest within Lovable, Bolt, Base44

ADD A PREMIUM PLAN SAFELY

- 1 Work in a sandbox first.**
Stripe’s sandboxes are test environments where no real money moves.
- 2 Ask the AI to use Stripe-hosted Checkout**, not a home-made card form.
- 3 Store the Stripe secret key** as a back-end secret, never in the front end.
- 4 Have Stripe tell your back end about payments** with a **webhook** (an automatic message from Stripe to your app). Only the webhook should mark someone as premium.
- 5 Test with Stripe’s test card** `4242 4242 4242 4242` , any future expiry date, and any three-digit security code. Also test a declined payment and a cancellation.
- 6 Switch to live keys** only when every test passes.

PROMPT

Explain, step by step, how a payment marks someone as premium in this app. Where is that decided: browser or server? What happens if the payment fails, or the user cancels next month?

 WARNING

If “premium” is decided in the browser, anyone can unlock it for free. The server, informed by Stripe’s webhook, must decide.

19.3 Sending email

Apps send **transactional email**: sign-in links, receipts, reminders. Use an email service such as **Resend** (free plan: 3,000 emails a month, up to 100 a day) or your builder’s built-in email.

- **Send from your own domain** and complete the service’s verification steps; otherwise mail lands in spam.
- **Supabase’s default sender is for testing only.** Its sending is limited, so connect a custom email service (SMTP) before launch.
- **Include an unsubscribe link** in anything that isn’t strictly necessary, like newsletters.

19.4 API keys and environment variables

An **API key** identifies your app to a service. 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.

Some variables are deliberately public. Variables whose names start with `VITE_` (Vite apps) or `NEXT_PUBLIC_` (Next.js apps) are baked into the front end, where anyone can read them. Only publishable values, like a Supabase publishable key, belong there.

.ENV

```
# Public: fine in the browser
VITE_SUPABASE_URL=https://yourproject.supabase.co
VITE_SUPABASE_PUBLISHABLE_KEY=sb_publishable_...
# Secret: back end only, never with a public prefix
STRIPE_SECRET_KEY=sk_test_...
RESEND_API_KEY=re_...
```

 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.

- ☐ Sign-in uses built-in auth; live URLs configured
- ☐ Payments use Stripe Checkout or Payment Links, tested in a sandbox
- ☐ Premium status set only by the server via webhook
- ☐ Email sent from a verified domain through a real email service
- ☐ No secret has a `VITE_` or `NEXT_PUBLIC_` prefix

Leveling Up: Editors and Agents

Browser builders take you far. When you need more control, the same project can move to an AI editor or agent on your computer without starting over.

IN THIS CHAPTER

- Signs you've outgrown a browser builder
- Moving a project to your computer through GitHub
- Working in Cursor, and with agents like Claude Code
- Working through pull requests

20.1 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.
- You're spending more on credits than a flat-rate agent plan would cost.

20.2 Moving your project

OPEN YOUR BUILDER PROJECT ON YOUR OWN COMPUTER

- 1 Make sure it's on GitHub.**
Use the builder's GitHub sync.
- 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.

NOTE

Running locally usually needs **Node.js** installed, plus a `.env` file with your project's public settings. The `.env` file isn't in GitHub (it's ignored on purpose), so ask the AI which values it needs and where to find them.

20.3 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

20.4 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. They usually ask permission before running commands or editing files; read what they want to do before approving.

PROMPT

Read this project and AGENTS.md. Then add weekly email reminders as described in docs/reminders-spec.md. Plan first and wait for my OK. Work on a new branch, run the app to test, and commit as you go.

20.5 Working through pull requests

Pair agents with GitHub so every change arrives as a reviewable **pull request**. You read the summary, look at the list of changed files, try the preview deployment many hosts create for each pull request, and merge only when happy.

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*.

- ☐ My project runs on my own computer
- ☐ I know which `.env` values it needs
- ☐ I've had an agent make one change on a branch
- ☐ I've reviewed and merged a pull request

Deployment and Domains

Deploying (also called publishing or shipping) means putting your app on a computer that's always online so others can use it. This chapter compares the main hosts and walks through connecting your own domain.

IN THIS CHAPTER

- Where to host, and what the free plans allow
- Deploying from GitHub
- Environment variables in production
- Custom domains and DNS records
- A pre-launch checklist

21.1 Hosting options

OPTION	GOOD FOR	FREE PLAN
Your builder's Publish button	The easiest path	Usually a free subdomain; custom domains often need a paid plan
Vercel	Next.js and React apps; great with v0	Hobby: personal, non-commercial use
Netlify	Static sites and front-end apps, forms	300 credits a month
Cloudflare Workers	Fast sites and apps worldwide	100,000 requests a day; static files free
GitHub Pages	Simple static websites (see the <i>GitHub</i> manual)	Free for public repos
Replit	Apps built in Replit	One published app on Starter

A few details worth knowing:

- **Vercel Pro** is \$20/mo with \$20 of included usage. New teams get a default on-demand budget of \$200 with alerts, and you can set a hard limit that pauses projects.
- **Netlify** charges in credits: on the free plan, each production deploy costs 15 credits, so about 20 deploys a month. Deploy previews are unlimited. Personal is \$9/mo (1,000 credits); Pro starts at \$20.
- **Cloudflare** now recommends **Workers** for new projects; Pages still works for existing sites.

21.2 Deploying from GitHub

DEPLOY A GITHUB REPO TO VERCEL OR NETLIFY

- 1 **Sign in to the host with GitHub.**
- 2 **Import the repository.**
Choose it from the list.
- 3 **Accept the detected settings.**
Hosts recognize Vite and Next.js projects automatically.
- 4 **Add environment variables** before the first deploy (see below).
- 5 **Deploy.**
You get a live address. From now on, every push to `main` redeploys automatically, and other branches get preview addresses.

21.3 Environment variables in production

Your `.env` file stays on your computer. The host needs its own copy of each value, entered in the project's settings (on Vercel, the project's **Settings**, then **Environment Variables**). After changing a variable, **redploy**: running deployments don't pick up changes on their own.

21.4 Your own domain name

Buy a domain (like `plantpal.app`) from a registrar such as Cloudflare, Namecheap, or Porkbun. Cloudflare Registrar charges only what the registry charges, with no markup. Lovable, Bolt, Base44, and Replit also sell domains inside the app, which sets up DNS for you.

Then, in your host's **Domains** settings (on Vercel, **Settings** > **Domains**), 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.

RECORD	WHAT IT DOES	TYPICAL USE
A	Points a name to a numeric address	The bare domain, <code>plantpal.app</code>
CNAME	Points a name to another name	<code>www.plantpal.app</code> → your host
TXT	Holds text for verification	Proving you own the domain; email setup
MX	Says where email goes	Receiving email at your domain

 WARNING

Copy record values exactly from your host's screen, and don't delete MX or TXT records you don't understand: they may be running your email.

HTTPS certificates are issued automatically once DNS points at the host.

21.5 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)
- ☐ Auth redirect URLs updated to the live domain

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. This chapter is your escape plan, plus a playbook for the error messages you’ll see most.

IN THIS CHAPTER

- Breaking the fix-and-break loop
- Where to find error messages
- An error-message playbook
- Prompts for diagnosis

22.1 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.

22.2 Where errors hide

Error messages look scary but are just clues. Copy the *whole* message, including file names and line numbers, and paste it to the AI.

WHERE	HOW TO OPEN IT	SHOWS
Browser console	<code>F12</code> , or right-click → Inspect → Console	Front-end errors
Network tab	Same panel → Network ; click a red row	Failed requests to your back end
Builder logs	Your builder's logs or Cloud/database panel	Back-end and database errors
Host logs	Vercel/Netlify deployment and function logs	Build failures, server errors
Terminal	The window running <code>npm run dev</code>	Local errors

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.

22.3 Error-message playbook

YOU SEE	IT USUALLY MEANS	SAY TO THE AI
<code>TypeError: cannot read properties of undefined</code>	Code used data that hadn't arrived	“Handle the case where this data is missing or still loading”
<code>404 Not Found</code>	A page or address doesn't exist	“This link/request goes to a missing address; fix the route”
<code>401 Unauthorized</code> / <code>403 Forbidden</code>	Not signed in, or not allowed	“Check auth and permissions for this request”
<code>500 Internal Server Error</code>	The back end crashed	“Check the server logs for this request and fix the cause”
<code>CORS policy error</code>	A service refused requests from your site's address	“Fix CORS for this API call; call it from the back end if needed”
<code>violates row-level security policy</code>	Database rule blocked the action	“Explain which policy blocks this and fix the data or policy safely”
<code>Module not found</code> / <code>Cannot find package</code>	A library isn't installed	“Install the missing package and check the import”
<code>429 Too Many Requests</code>	Rate limit hit	“Add retries with waiting, and reduce how often we call this”
Build failed on deploy	Code works in preview, fails on the host	Paste the full build log
Blank white page	The front end crashed on load	Paste the console error

22.4 Diagnosis prompts

PROMPT

Here's exactly what happens: [steps]. Here's what I expected: [x].
Here's the full error: [paste]. Before fixing, explain in plain
English what's going wrong and why your fix will work.

PROMPT

We've tried to fix this three times. Step back. Summarize what we've
tried, what we learned, and propose a completely different approach.

- ☐ I roll back instead of piling on fixes
- ☐ I know how to open the console and network tab
- ☐ I paste full errors, with what I did and what I expected

Security for Vibe Coders

AI tools make it easy to build apps that *work* but aren't *safe*. Most leaks from vibe-coded apps come from the same handful of mistakes, so a few habits prevent nearly all of them.

IN THIS CHAPTER

- The big five security habits
- The most common vibe-coded app leaks
- Using built-in scanners, and asking the AI to audit itself
- What to do if a key leaks
- Privacy basics

23.1 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.

23.2 Common leaks in vibe-coded apps

LEAK	HOW IT HAPPENS	PREVENTION
Whole database readable	RLS off, or a <code>using (true)</code> policy	RLS on everywhere; two-user test
Secret key in the browser	Key given a <code>VITE_</code> / <code>NEXT_PUBLIC_</code> prefix, or pasted in code	Back-end secrets only
Admin pages open to all	Hidden link but no server check	Server-side role checks
Public file buckets	Uploads stored publicly by default	Private buckets for private files
Keys in GitHub	<code>.env</code> committed	<code>.gitignore</code> , push protection
Open AI endpoint	Anyone can call your chatbot's back end	Require sign-in, rate limits

23.3 Use the scanners

- **Lovable** runs a Quick security scan every time you publish, and offers a Deep scan of your app's logic and permissions.
- **Bolt** scans your Cloud database for issues like a missing RLS policy.
- **Base44** has a one-click security scan with suggested fixes.
- **Supabase** has a Security Advisor in its dashboard.

Scanners catch common problems but can't guarantee safety. For sensitive data, get a professional review.

23.4 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.

23.5 If a key leaks

RESPOND TO A LEAKED KEY

- 1 **Revoke or rotate it immediately** in the service's dashboard. Deleting it from your code isn't enough: it's in your history.
- 2 **Create a new key** and store it as a secret.
- 3 **Check usage and billing** for anything unexpected.
- 4 **Find how it leaked** (chat, screenshot, commit, front end) and close that gap.

23.6 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.

- ☐ RLS on for every table; two-user test passed
- ☐ No secrets in the front end or Git history
- ☐ Server-side checks for roles, prices, and premium status
- ☐ Built-in security scan run before each launch
- ☐ Spending limits on every paid service
- ☐ Privacy policy matches what I actually collect

Costs and Credits

Most vibe coding tools charge with **credits** (or **tokens**), units of AI work. Bigger requests and longer conversations use more. The running app has costs too: hosting, database, email, and AI calls. A little planning keeps the total low.

IN THIS CHAPTER

- Where the money goes
- Making credits last
- Free tiers worth knowing
- A typical budget, and avoiding surprise bills

24.1 Where the money goes

COST	CHARGED BY	TYPICAL PATTERN
Building	Builder or agent plan	Monthly plan plus top-ups
Hosting	Builder, Vercel, Netlify, Cloudflare	Free at first; grows with traffic
Database	Supabase, builder cloud	Free until you need more or stop pauses
AI features in your app	AI provider or builder credits	Per use; can spike
Email	Resend or similar	Free tier, then monthly
Domain	Registrar	Yearly
Store accounts	Apple (\$99/yr), Google (\$25 once), Chrome Web Store (one-time)	Only if you publish there

24.2 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. In Lovable, a Plan mode message costs one credit.
- **Keep projects focused.** Bolt and others use more tokens as projects grow.
- **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.

24.3 Free tiers worth knowing

SERVICE	FREE ALLOWANCE
Lovable	5 build credits a day, up to 30 a month
Bolt	300K tokens a day, 1M a month
v0	7 messages a day
Base44	25 message credits a month
Supabase	2 projects, 500 MB database, 50,000 monthly users (pauses when idle)
Netlify	300 credits a month
Cloudflare Workers	100,000 requests a day; static files free
Resend	3,000 emails a month (100 a day)
Copilot Free	2,000 completions a month

24.4 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

24.5 Avoiding surprise bills

1. **Set limits everywhere:** Supabase's spend cap (on by default on Pro), Vercel's budget and hard limit, AI provider spending limits, Google Cloud budget alerts.
2. **Rate-limit AI features** per user (see Project 6).
3. **Watch usage weekly** for the first month after launch.
4. **Cancel what you don't use.** Credits on many plans expire anyway.

- ☐ I know every service my app pays for
- ☐ Spending limits or alerts are on for each
- ☐ I use plan/chat modes and visual edits to save credits

Maintaining and Handing Off

Launching is the start, not the end. Apps need small amounts of care: updates, backups, and documentation. And one day you may want to hand the project to a developer, a colleague, or a future you who has forgotten everything.

IN THIS CHAPTER

- A light maintenance routine
- Keeping libraries up to date safely
- Writing a README that saves hours
- Handing off to a developer

25.1 A light maintenance routine

HOW OFTEN	TASK
Weekly	Glance at usage, errors, and billing; read user feedback
Monthly	Export a backup; run the security scan; check auth and email still work
Every few months	Update libraries; review who has access to every service
Yearly	Renew the domain; review the privacy policy

TIP

Free Supabase projects pause after a week with no activity. If your app is used rarely, expect the occasional restore, or upgrade when it matters.

25.2 Updating libraries safely

Your app depends on dozens of **packages** (code libraries). They get security fixes and new versions.

PROMPT

Check this project for outdated or vulnerable packages. Update only the ones with security fixes or safe minor updates, on a new branch. Run the app and test script, then summarize what changed.

Big “major version” upgrades can break things; do them one at a time, on a branch, with your test script.

25.3 Write a README

A **README** is the front page of your project: a plain-text file explaining what it is and how to run it. Ask the AI to write it, then check it:

README.MD

```
# Plant Pal
Phone-first app reminding people to water houseplants.

## Stack
React + Vite front end; Supabase (database, auth); hosted on Vercel.

## Run locally
1. `npm install`
2. Copy `.env.example` to `.env` and fill in the values.
3. `npm run dev`

## Services and accounts
Supabase (owner: me), Vercel, Resend, domain at Cloudflare.

## Known issues
Reminder emails sometimes send twice (see docs/issues.md).
```

Keep a `.env.example` file listing variable *names* with no real values, so others know what to set.

25.4 Handing off to a developer

PREPARE A CLEAN HANDOFF

1

Make sure everything is in GitHub, with a README and `.env.example`.

- 2 **List every service** the app uses, who owns each account, and how billing works.
- 3 **Share your brief, specs, and build diary.**
They explain *why* things are the way they are.
- 4 **Give access properly.**
Invite people to GitHub, Supabase, and the host as team members; never share your passwords or secret keys in chat.
- 5 **Ask for a review first.**
“Spend two hours reviewing security and structure; tell me the top five risks” is a great first job.

☆ **IMPORTANT**

Own your accounts. Even if someone else builds or maintains the app, the domain, GitHub organization, database, and payment accounts should be registered to you or your business.

- ☐ Maintenance routine in my calendar
- ☐ README and `.env.example` in the repo
- ☐ A list of every service and account owner
- ☐ I know how to grant and remove access to each service

Troubleshooting Table

Find your symptom, try the fix, and if that fails, roll back and use the diagnosis prompts in “When You Get Stuck.”

26.1 Building

SYMPTOM	LIKELY CAUSE	FIX
AI built something different from what I meant	Vague prompt	Restore; re-prompt with who, what, example, and “done means”
Each fix breaks something else	Changes too big; long chat	Roll back; smaller slices; fresh chat with summary
AI says it fixed it, but nothing changed	Cached preview, or wrong file edited	Hard refresh; ask it to show which file changed
Credits vanish quickly	Large project, big requests	Plan/chat modes, visual edits, focused requests
Design keeps drifting	No lasting rules	Put style rules in Knowledge / AGENTS.md
AI invents features I didn’t ask for	Open-ended prompt	Add “Change only X. Don’t add anything else.”
AI loops on the same error	It’s guessing	Ask for three causes and tests, not a fix

26.2 Data and logins

SYMPTOM	LIKELY CAUSE	FIX
Data disappears on refresh	Stored in the page, not a database	“Save this to the database”
List is empty after adding RLS	No select policy, or user_id missing	Ask AI to check both
Users see each other’s data	Missing or <code>using (true)</code> policy	Fix RLS now; two-user test
Sign-in link goes to localhost	Site URL not updated	Authentication > URL Configuration
Sign-in emails don’t arrive	Default sender limits; spam	Custom email service; verify domain
App broke after a quiet week	Free Supabase project paused	Restore it in the dashboard
“Invalid API key”	Wrong or legacy key, or wrong name	Re-copy into secrets; check names

26.3 Payments and email

SYMPTOM	LIKELY CAUSE	FIX
Payment works but user not upgraded	Webhook missing or failing	Check webhook logs in Stripe; ask AI to fix handler
Anyone can get premium	Decided in the browser	Move the decision to the server
Real card charged during testing	Live keys used	Switch to sandbox keys until launch
Emails go to spam	Domain not verified	Complete the email service's DNS steps

26.4 Deploying and domains

SYMPTOM	LIKELY CAUSE	FIX
Works in preview, broken live	Missing environment variables	Add them on the host; redeploy
Build failed	Code or package error	Paste the full build log to the AI
Blank white page live	Front-end crash	Open the console; paste the error
Domain shows "not found"	DNS not pointing at host yet	Check records match exactly; wait
"Not secure" warning	Certificate not issued yet	Wait for DNS; re-check in host settings
Email stopped after domain change	MX records deleted	Restore MX records from your email provider

26.5 Costs and limits

SYMPTOM	LIKELY CAUSE	FIX
Surprise AI bill	No limits; bots or loops	Spending limit, sign-in, per-user rate limit
"429 Too Many Requests"	Rate limit hit	Wait; reduce calls; raise tier
Netlify deploys paused	Free credits used up	Fewer production deploys; upgrade
Hosting site paused	Budget hard limit reached	Review usage; raise limit deliberately

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

27.1 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.

27.2 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.

27.3 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

28.1 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.

28.2 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.

28.3 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.

28.4 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.

28.5 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.

28.6 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].

28.7 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.

28.8 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

32.1 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).

32.2 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.

32.3 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.

32.4 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.

32.5 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

33.1 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
```

33.2 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.

34.1 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.

34.2 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.

34.3 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.

34.4 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."

34.5 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.

34.6 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.

34.7 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.

35.1 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.

35.2 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.

35.3 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)

35.4 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.

36.1 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"

36.2 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.

37.1 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.

37.2 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

37.3 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

37.4 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.

38.1 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."

38.2 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."

38.3 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."

38.4 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”

Feedback: see / expect / where. Say what to keep, too.

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

Data safety: RLS on every table · two-user test · secret key only on the server ·

`VITE_ / NEXT_PUBLIC_ = public`

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

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

Test card: 4242 4242 4242 4242, any future date, any 3-digit code (sandbox only)

Chrome extension: `chrome://extensions` → Developer mode → Load unpacked → reload after changes

Expo: `npx expo start` → scan QR with Expo Go → `--tunnel` if it won't connect

Glossary

Agent	An AI that takes actions, such as editing files and running commands, in a loop until a task is done.
Alt text	A short written description of an image, read aloud by screen readers.
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.
Authorization	Deciding what a signed-in user is allowed to do.
Back end	The part of an app that runs on a server: data, logins, emails, payments.
Branch	A separate line of work in Git, for experimenting without touching the main version.
Browser extension	A small add-on that changes how a web browser works.
Bug	Something in an app that doesn't work the way it should.
Checkpoint	A saved, working version of your project you can return to.
Clone	Download a full copy of a repository to your computer.
Code	The written instructions that make an app work.
Code editor	A program for viewing and editing code files, such as VS Code or Cursor.
Commit	A saved snapshot of your project in Git, with a message describing the change.
CORS	A browser safety rule that controls which websites may call a service.
Credits	Units many AI tools use to measure and charge for usage.

CSV	A simple spreadsheet file format: values separated by commas.
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.
Domain	A human-friendly web address, such as plantpal.app.
Edge case	An unusual situation an app must handle, like an empty list or a very long name.
Edge function	A small piece of back-end code that runs on demand; Supabase's and Lovable's name for serverless functions.
Environment variable	A setting stored outside your code, used for secrets like API keys.
Expo	A free toolkit for building iPhone and Android apps with React Native.
Expo Go	A phone app for testing Expo projects by scanning a QR code.
Foreign key	A column holding the ID of a row in another table, linking the two.
Front end	The part of an app that runs in the user's browser.
Git	A program that records the history of your project's files.
GitHub	A website that stores Git repositories online and supports reviews and collaboration.
Honeypot	A hidden form field that only bots fill in, used to block spam.
Hosting	Running your app on an always-online computer.
HTTPS	The encrypted connection shown by the browser's padlock.
Internal tool	An app used inside a business or group rather than by the public.
JSON	A simple text format of labels and values that programs use to exchange data.
Landing page	A single web page with one goal, such as joining a waitlist.
Last-20% wall	The slower final stretch of building: edge cases, logins, and polish.
Localhost	An address meaning "this computer"; only you can see apps running there.
Magic link	A one-time sign-in link sent by email, so no password is needed.

Manifest V3	The current format for Chrome extensions, set in manifest.json.
Node.js	Software that runs JavaScript on a computer or server.
npm	The tool that installs JavaScript libraries (packages).
Package	A reusable code library your app depends on.
Plan mode	A builder mode that writes a plan for your approval before changing code.
PRD	Product requirements document: a brief describing what to build and why.
Preview	A live, clickable version of your app inside a builder.
Programming language	A precise written language computers follow, such as JavaScript or Python.
Prompt	The instructions you give an AI.
Publishable key	A Supabase key safe to use in the front end; limited by row-level security.
Pull request	A proposal to merge changes into the main version, laid out for review.
Push	Send new commits from your computer to GitHub.
Rate limiting	Capping how often someone can do something, such as submit a form.
React	A popular way to build interfaces from reusable components.
README	A file explaining what a project is and how to run it.
Repository	A project folder whose history is tracked by Git, often stored on GitHub. Also “repo.”
Row-level security	Database rules ensuring each user can only access the records they’re allowed to.
Rules file	A file of lasting instructions an AI tool reads automatically, such as <code>AGENTS.md</code> or <code>CLAUDE.md</code> .
Sandbox	A test environment, such as Stripe’s, where no real money moves.
Secret key	A key with full access (in Supabase, bypassing row-level security) that must stay on the back end.
Server	A computer that is always on and connected to the internet, running back-end code.
Serverless function	Back-end code that runs only when called, without managing a server.
Spec	A short, structured description of how a feature should behave.
SQL	The language used to query and change databases.

Static site	A website that's the same for every visitor and needs no back end.
Storage bucket	A folder-like store for uploaded files, set as public or private.
Streaming	Showing an AI reply word by word as it's generated.
Stripe	A payments service that handles card details and checkout.
Supabase	A popular hosted database service with built-in logins and file storage.
System prompt	Hidden instructions that shape every reply of an AI chatbot.
Terminal	A text window for typing commands.
Test script	A written list of things to try after each change.
Token	A small chunk of text used to measure AI usage, roughly three-quarters of a word.
Transactional email	Emails an app sends automatically, like sign-in links and receipts.
Vibe coding	Building software mainly by describing what you want to an AI and iterating on the results.
Vite	A fast tool for building and running front-end apps; common in Lovable and Bolt.
Webhook	An automatic message one service sends to your app when something happens.