FREE PROMPT PACK

The Fable Prompt Pack

The five prompts from the video — the exact runs I used on Fable 5, scrubbed of my specifics and rebuilt with clearly-marked placeholders so you can drop them straight into your own setup. They work on Fable 5 during the free window, and on any top model (Opus, GPT) at high effort after it closes.

By Preston Chin · copy-paste ready

How to use this pack

Each prompt below is a full run — a real task you can hand to an AI coding agent like Claude Code or Codex and walk away from. Four steps:

1. **Pick your model and set the effort high.** During the free window (through July 7), switch to Fable 5 and set effort to **high** or **X-high**. After that, any top model — Opus, GPT — at high effort works the same way. Effort level is the trade-off dial between intelligence, speed, and cost; these are the tasks worth spending on.
2. **Fill in the placeholders.** Every `[LIKE THIS]` is a blank for you to fill. The key is in the next section. Don't skip them — the context you paste in is where the quality comes from.
3. **Paste it in and let it ask questions.** Each prompt ends by telling the model to ask its clarifying questions first. Answer them, then say go. That two-second habit prevents it running an hour down the wrong path.
4. **Let it run.** These are long-horizon tasks — that's the whole point. Start it, go do something else, come back to a finished report or a working build.

> **The advisor strategy — match the model to the job**
>
> Use the top model (Fable, Opus) for the hard thinking: the audits, the design, the one-pass build. For grind-y execution afterward — applying a hundred small fixes, boilerplate — hand it down to a cheaper, faster model like Sonnet or Codex. You delegate up for judgment and down for labor.

The placeholder key

Fill every one of these before you run a prompt. If a placeholder doesn't apply to you, delete that line rather than leaving it blank.

`[YOUR WEBSITE URL]`	Your live site, e.g. yoursite.com.
`[YOUR STACK]`	What your site is built with — Astro, Next.js, WordPress, Webflow, etc.
`[YOUR PRODUCT]`	The app or software you're hardening, and its URL(s).
`[YOUR REPO]`	The GitHub repo (or local path) the code lives in.
`[PATH TO YOUR PROJECT]`	The folder on your computer where your project lives.
`[MODEL]`	The model you run day to day, e.g. Opus 4.8.
`[LIST YOUR KEY FILES]`	The 5–10 files that matter most — your layout, stylesheet, nav, homepage, content model.
`[WHAT YOU'RE BUILDING]`	The prototype or tool you want, in one line.
`[LIST YOUR BRANDS / INCOME LINES]`	Each brand, product, or income stream and how it makes money.

Not sure what a placeholder wants? Paste the prompt in anyway and add: "Some placeholders I left rough — ask me about anything you need before you start." A good model will fill the gaps by interviewing you.

1. Audit your AI setup

Point it at your Claude Code / Codex config and have it tell you what is bloated, contradictory, or stale. Cheap, read-only, compounds every future session.

Report only. Run from your home directory.

PROMPT

```
I'm optimizing my Claude Code setup — my CLAUDE.md files, skills, hooks, memory, and settings — because [MODEL, e.g. Opus 4.8] is my daily driver for [WHAT YOU USE IT FOR]. My setup has grown organically and is probably bloated, contradictory in places, and partly stale. I need it sharp so every routine session runs better.

Read my whole setup:
- Global: ~/.claude/CLAUDE.md, ~/.claude/skills/, ~/.claude/settings.json, and my memory directory
- Project: [PATH TO YOUR PROJECT]/.claude/ (CLAUDE.md, rules/, skills/, agents/, hooks/, settings.json, scripts/)

Request: audit it like a senior AI engineer taking over my setup on day one, and tell me exactly how to make it work better for daily use.

Judge each piece on: does it actually change model behavior, or is it dead weight? Where do rules contradict each other? What's stale — references to tools, models, or files that no longer exist? Which skills overlap or should merge? What's missing that I clearly need? Is my CLAUDE.md too long to be followed reliably, and what should move to skills or memory?

Output format: a prioritized report at ~/.claude/harness-audit/README.md. Every finding tagged P0–P3, with the exact file + line, the concrete fix, and whether it's a quick win (safe to apply now) or a judgment call (needs my sign-off). End with the 5 changes that would most improve my daily sessions and the 5 safest quick wins you can apply the moment I say go.

Constraints: the deliverable is your assessment. Report and stop. Do not edit, delete, merge, or rewrite anything until I say go — this is my live setup and a wrong change breaks real automations. Don't assume a skill is unused just because it looks redundant; flag it, don't touch it. Ask any clarifying questions up front, then produce the report end-to-end.
```

Why it works. The cheapest, safest run to start with. It only reads. You get one prioritized report, and every fix you apply makes every future session sharper. Watch for the token count it flags before you type a word — most people are shocked how heavy their setup got.

2. Website design + SEO audit

Have it boot your site, screenshot every page, and hand you a ranked redesign plan built around getting found in search.

Plan + screenshots. Run from your website project folder.

PROMPT

```
I'm redesigning [YOUR WEBSITE URL] (built with [YOUR STACK, e.g. Astro / Next.js / WordPress]) to be cleaner, more polished, and far more discoverable in search. Priority #1 is getting found in search; then a cleaner, more polished look; then making my content easy to browse.

Request: audit the site like a senior design + SEO lead on their first day, then give me a redesign plan. Boot the local dev server and screenshot every public page (desktop + mobile).

Cover:
1. SEO (top priority) — per-page titles, descriptions, and canonicals; structured data (Organization / Person / WebSite / Breadcrumb / Article); default OG images; sitemap and robots hygiene (make sure any private or admin pages aren't leaking into search); internal linking; and how to make individual pages actually rank.
2. Information architecture — how a visitor picks a topic and lands on the right content, resources, and products in one place. Give a concrete nav + page structure with 2 layout options for the main hub.
3. Resources / library page — propose a cleaner, organized, filterable structure.
4. Design system health — flag dead or duplicated CSS, token duplication, and the missing reusable components that would make the redesign maintainable.

Key files to read first: [LIST YOUR KEY FILES — e.g. the layout/head file, the global stylesheet, nav + footer, the homepage, the content model]. Treat any admin, dashboard, or API routes as private — audit them only for search or security leakage.

Output format: a prioritized plan at docs/site-audit/README.md, with screenshots in docs/site-audit/assets/. Every issue tagged P0–P3, which it hurts (SEO / understanding / trust / conversion), the exact file + line, and the specific fix. End with the 5 changes that most improve search discoverability and the 5 safe quick wins you can apply today.

Constraints: fix only trivially safe things on the spot — a meta tag, a robots or sitemap exclusion, a typo, dead CSS you can prove is unused. Everything structural — new nav, new routes, the redesign itself — is a spec and recommendation only; do not implement until I approve. Never touch payment, checkout, auth, or API routes. Ask your clarifying questions up front, then run the audit end-to-end.
```

Why it works. A design + SEO audit is a perfect long-running task to hand off — it's read-heavy, visual, and the payoff compounds. The constraints matter: without the "spec only, don't implement" fence, a strong model will happily start rewriting your whole site.

3. Security + reliability audit

Turn parallel agents loose on your product to find what will break for a paying user before they find it. Findings only — nothing gets changed.

Report only. Run from your product repo.

PROMPT

```
I'm hardening [YOUR PRODUCT] — [the marketing site and/or the app] — in the repo [YOUR REPO]. Before I put more paying users on it, I need to know what will break in real use before they find it. This is an audit only: you produce the findings, and I'll hand the implementation to [me / my developer / another model] — so write every finding so it can be implemented cleanly from your report alone.

Request: audit this codebase from first principles for reliability and security, then attack it. Don't pattern-match to generic checklists — start from what actually has to be true for a paying user's session to work end-to-end, and find where it isn't.

Spin up parallel agents, each in one role: hostile user, careless user, performance, security, and data integrity. Each hunts only from its angle — what input breaks it, what edge case got skipped, where it crashes, leaks, loops, corrupts, or loses a user's data, or silently fails. Pay special attention to auth / session / token handling and access control (can one user reach another user's data?), the core pipeline, external API calls, payment or subscription state, and error handling on every write.

Output format: one report at docs/reliability-audit/README.md, findings worst-first. Write each finding as a self-contained task: the exact trigger + reproduction, the file + line, the blast radius, a concrete proposed fix, and how to verify it (ideally a regression test to add). Separate "will break in production" (P0/P1) from "should fix eventually" (P2/P3). End with the 5 that would hurt a paying user most.

Constraints: this is an audit — findings only. Do NOT change any code, migrations, or config. Don't defend the code; attack it. Don't touch production data or run anything destructive. Flag anything that needs a secret or credential I have to provide rather than guessing. Ask clarifying questions up front, then run the audit end-to-end.
```

Why it works. The "attack it, don't defend it" framing is what makes this work — models are trained to be helpful, so you have to explicitly give them permission to be adversarial. Findings-only keeps it safe to run against a live product, and the report is written so whoever implements the fixes can work straight from it.

4. Build a prototype in one pass

Get a polished, working first version out the door in one run instead of days of back-and-forth. Best after you spend 20 minutes planning on a cheaper model first.

Builds a working local prototype. Run at high or X-high effort.

PROMPT

```
I want to build [WHAT YOU'RE BUILDING — e.g. a simple internal dashboard, a client-facing prototype, an internal tool]. Here's the goal and why it matters: [ONE OR TWO SENTENCES ON THE OUTCOME YOU WANT AND WHO IT'S FOR].

Context to ground this: [PASTE OR POINT TO anything real — a rough plan, the data it should read (a spreadsheet, a sample file), an existing prototype to extend, your brand colors and fonts, and 1–2 examples of the look and feel you want].

Request: build the smallest version that actually works, end to end, in one pass. Make real design decisions — I want it to look polished, not like a wireframe. Wire it to the real data source above (read-only unless I say otherwise). Where you have to assume something, pick the most sensible default and note it rather than stopping to ask on every small thing.

Output: a working [app / page / tool] I can run locally, plus a short list of the choices you made and the 3–5 things you'd polish next.

Constraints: don't over-build — no features I didn't ask for, no speculative flexibility. Never write to or delete real user data, and never auto-post or publish anything. If a decision is genuinely irreversible or only I can make it, pause and ask. Otherwise keep going until it runs. Ask your clarifying questions up front, then build it end-to-end.
```

Why it works. Two-step move: first spend 20–30 minutes on a cheaper model (Sonnet, or Opus at medium effort) letting it interview you to get the plan on paper. Then paste that plan into the Context slot above and run this on a top model at high effort. The planning is where the quality comes from — the build is just execution.

5. Business + strategy audit

An honest outside-operator read on where your leverage is, what to cut, and the most credible paths to more revenue.

Writes a strategy memo. Reads whatever real context you point it at.

PROMPT

```
I run several things [solo / with a small team] and I want a hard, strategic outside read on where to focus to grow — especially revenue. My portfolio: [LIST YOUR BRANDS / PRODUCTS / INCOME LINES, one line each — what each is and roughly how it makes money]. My worry is that I'm spread too thin and leaving leverage on the table.

Read my real business context to ground this — don't guess where you can read: [POINT TO anything real — your notes, plans, a strategy doc, a folder of context]. If you don't have files to read, work from what I tell you below.

Request: audit my business like a sharp operator or advisor who just joined, and tell me where I could go better strategically. Specifically: where is my highest-leverage focus; what is spreading me too thin or should be cut or parked; how the brands do (or don't) reinforce each other; and the 3 most credible paths to more revenue. Be direct and rank everything by leverage, not by how much work I've already put in.

Before you start, ask me for the few numbers that would change your answer — rough revenue per line, audience or list sizes, and how my week splits across these. Use them; don't invent them.

Output format: a strategy memo — (1) the one-paragraph honest read, (2) a ranked "where the leverage is" list, (3) what to cut or park and why, (4) 3 revenue paths with the first concrete move for each, (5) the single thing you'd do this month if you were me. Flag every place you're inferring versus certain.

Constraints: analysis only — don't change any files. Don't flatter me or hedge; if something should be killed, say so. Don't fabricate numbers, testimonials, or facts — mark unknowns as unknowns. Ask your clarifying questions up front, then write the memo end-to-end.
```

Why it works. Don't take the output as gospel — take it as a gut check. Its value is forcing you to defend the decisions you've already made, or admit you can't. Ranking by leverage instead of sunk effort is the line that makes it actually useful.

The one rule

If you take one thing from this pack, take this: nail the **why** behind your prompt — the goal and why it matters — and let the model figure out the how. Every prompt above gives it the outcome, the context, the output format, and a fence of constraints, then gets out of the way. You're not writing instructions for a script. You're briefing a very capable operator on what a good result looks like.

Fill the placeholders, set the effort high, let it ask its questions, and let it run. That's the whole thing.