

PROMPT PACK · VOL. 2

Ship It Safe

Lock down your vibe-coded app before launch.

For Cursor · Claude Code · Claude · ChatGPT · Windsurf · Lovable · Bolt · v0 · Replit

24 prompts · 8 sections

Welcome! How to Use This Pack

AI tools can build an app in an afternoon. They're less reliable at the boring parts that keep it safe: keeping keys secret, locking down the database, checking who's allowed to do what, and stopping one visitor from running up your bill.

This pack gives you prompts that make your AI **check those things, explain what it found in plain English, and fix them one step at a time**. It helps you catch common mistakes. It can't promise your app is 100% secure. No checklist can do that, and your AI can be wrong too. If your app handles payments, health data, or other sensitive information, get a human security review as well.

How each prompt is laid out

Every prompt comes with:

- **Title:** what the prompt is.
- **Use this when...:** one or two sentences on what the prompt is for and when to reach for it.
- **Fill in:** the blanks to replace before you paste.
- **The prompt:** the box you copy.
- **Tips to customize:** optional ways to change it (included where it helps).

Filling in the placeholders

Anything in **[BRACKETS IN CAPS]** is a blank for you to fill in. Delete the brackets and type your own details. If you don't know what to put, write **"not sure, please check my code and tell me"**.

Common placeholders

Placeholder	What to put there	Example
[APP IDEA]	One or two sentences about your app	"A journaling app where each user has private entries"
[TECH STACK]	Your languages, frameworks, and services	"Next.js, Supabase, Vercel"
[HOST]	Where the app is deployed	"Vercel" / "Netlify" / "Lovable" / "not deployed yet"
[DATABASE]	Your database or backend service	"Supabase" / "Firebase" / "none"
[AUTH PROVIDER]	How users log in	"Supabase Auth" / "Clerk" / "Firebase Auth"

Placeholder	What to put there	Example
[SERVICES]	Paid APIs and services you use	"OpenAI, Stripe, Resend"
[SKILL LEVEL]	How much coding you know	"total beginner"

Five safety habits

1. **Never paste real secret keys into an AI chat.** Use the name (like `STRIPE_SECRET_KEY`) instead of the value.
2. **Make the AI show you, not tell you.** "It's secure" means nothing. Ask it to show the exact file, line, setting, or policy.
3. **Test it like an attacker would, gently.** Log in as a second test user and try to see the first user's data. Log out and try again.
4. **Fix one thing at a time and commit after each fix.** Security fixes can break features. Small steps make it easy to see what changed.
5. **If a key leaked, rotate it first.** Deleting it from your code doesn't make it safe again (see Section 8).

Quick Start: Your First 30 Minutes

Short on time? Do these five in order. They catch some of the most common problems in vibe-coded apps.

1. **Find exposed secrets.** Use **Prompt 1.1 (Find Every Secret in My Project)**.
2. **Check your Git history.** Use **Prompt 1.3 (Did I Ever Commit a Key?)**. If anything turns up, go straight to **Section 8**.
3. **Check your database rules.** Use **Prompt 2.1 (Supabase RLS Audit)** or **Prompt 2.3 (Firebase Security Rules Check)**.
4. **Try to read someone else's data.** Use **Prompt 3.2 (Can User A See User B's Data?)**.
5. **Cap your spending.** Use **Prompt 5.2 (AI Spend Caps and Usage Limits)**.

When you have more time, run **Prompt 7.1 (Full Pre-Launch Security Review)** before you share your link.

Golden rule: Don't accept "done" or "it's secure." Ask the AI what it checked and how you can check it yourself.

What's Inside

1. Secrets & API Keys (4 prompts)
2. Database Security: Supabase & Firebase (3 prompts)
3. Login & Auth Hardening (3 prompts)
4. Bad Input & Common Web Attacks, in Plain Words (3 prompts)
5. Cost & Abuse Limits (3 prompts)
6. Packages & Dependencies (2 prompts)
7. Pre-Launch Checks & "Prove It's Done" (3 prompts)
8. Oops, I Leaked Something (3 prompts)

Using the Vibe Code Starter Pack too? This pack goes deeper than Starter Pack prompts 3.2 (Protect Pages and User Data), 6.3 (Launch Readiness Checklist), and 8.1 (Add an AI Chat Feature). Use those to build, and these to check and harden.

SECTION 1

Secrets & API Keys

Keep the keys that cost money or unlock data out of places anyone can see.

1.1 Find Every Secret in My Project

Use this when you're about to deploy or make a repo public, or any time you're not sure where your keys live. It's for finding API keys, passwords, and tokens sitting in your code, and checking which ones end up in the browser.

Fill in: [SKILL LEVEL], [TECH STACK], [SERVICES], [HOST].

```
Act as a careful security reviewer. Scan my project for secrets and explain what
you find in plain English. I'm a [SKILL LEVEL].
```

```
Tech stack: [TECH STACK]
Services I use: [SERVICES]
Host: [HOST]
```

Please:

1. Search every file (including config files, scripts, and old or unused files) for anything that looks like an API key, secret, password, token, private key, or database connection string. List each one with the file and line. Show only the first 4 characters of any value, never the full value.
2. For each one, tell me: is it meant to be public (like a Supabase publishable key, called the anon key on older projects, or a Firebase web config), or must it stay secret (like an OpenAI key, a Stripe secret key, or a Supabase secret key, called the service_role key on older projects)?
3. Tell me which values end up in the browser. Explain how my framework decides this (for example NEXT_PUBLIC_ in Next.js or VITE_ in Vite), and flag any secret that's exposed that way.
4. Check that .env and similar files are listed in .gitignore.
5. For each secret that's in the wrong place, show how to move it to an environment variable used only on the server, one fix at a time.
6. Create a .env.example file with the variable names but no real values.

Don't print full secret values anywhere in your answer.

 **Tips to customize:** After fixing, also check what the browser actually receives: open your live site, open DevTools, and search the Sources/Network tabs for the first few characters of each secret key. If you find one, treat it as leaked (Section 8).

1.2 Move My API Calls to the Server

Use this when Prompt 1.1 found a secret key used in front-end code, or your AI app builder put an API call in a component. It's for fixing the common mistake where the browser calls a paid API directly with a secret key.

Fill in: [SERVICE] (e.g. OpenAI), [TECH STACK], [HOST], [FILE NAME OR "PLEASE FIND IT"].

```
My app calls [SERVICE] directly from the browser, which exposes my secret key.  
Help me move it to the server.
```

```
Tech stack: [TECH STACK]
```

```
Host: [HOST]
```

```
Where the call happens now: [FILE NAME OR "PLEASE FIND IT"]
```

```
Please:
```

1. Explain in 2-3 sentences why a key in browser code is visible to anyone, even if it's in an environment variable.
2. Create a server-side route (API route, server action, or serverless/edge function, whichever fits my stack) that makes the call using the key from a server-only environment variable.
3. Update the front end to call my route instead of the service directly.
4. Make the route check that the user is logged in (if my app has logins) and validate the input before calling the service.
5. Return only the data the front end needs, never the raw key or full error details.
6. Tell me exactly where to add the environment variable on [HOST], and to remove any browser-exposed version of it.
7. List every file you changed.



Tips to customize: If you use Supabase, ask for a Supabase Edge Function instead. Once the key has been in browser code on a live site, rotate it (Prompt 8.1), because someone may already have copied it.

1.3 Did I Ever Commit a Key?

Use this when you've ever had a key in your code, even briefly, and your repo is (or might become) shared or public. It's for checking your Git history for secrets that were deleted from the code but are still saved in old commits.

Fill in: [SKILL LEVEL], [WINDOWS / MAC / LINUX], [GITHUB / GITLAB / NONE], [YES / NO / NOT SURE], [SERVICES].

```
I need to check whether any secrets were ever committed to my Git history, not  
just the current code. I'm a [SKILL LEVEL], so give me exact commands and explain  
them.
```

```
My computer: [WINDOWS / MAC / LINUX]
Repo hosted on: [GITHUB / GITLAB / NONE]
Is the repo public? [YES / NO / NOT SURE]
Services I use: [SERVICES]
```

Please:

1. Explain in plain English why deleting a key from my code doesn't remove it from Git history.
2. Recommend a free, well-known secret scanning tool (such as gitleaks or trufflehog), with exact install and run commands that scan the full history.
3. Tell me how to read the results and how to spot false alarms.
4. Tell me whether my repo host has built-in secret scanning or push protection and how to turn it on. (On GitHub, secret scanning runs free and automatically on public repos, and push protection for personal accounts blocks known secret types in pushes to public repos by default. Private personal repos usually aren't covered, so a local scanner still matters.)
5. If a real secret is found: tell me to rotate it at the provider FIRST (point me to Section 8 of my security pack), and only then explain the options for cleaning history (such as git filter-repo), including the downsides (rewriting history, force-pushing, and forks or clones that still have the old copy).

Don't ask me to paste any secret values into this chat.

 **Tips to customize:** Add a pre-commit secret scan so it can't happen again: "Set up a pre-commit hook that runs the scanner before every commit."

1.4 Set Up Environment Variables Properly

Use this when you're about to deploy, or your keys are scattered across files. It's for getting dev and production secrets organized so they're never in your code and don't get mixed up.

Fill in: [TECH STACK], [HOST], [SERVICES].

```
Help me organize the environment variables for my app properly.
```

```
Tech stack: [TECH STACK]
Host: [HOST]
Services: [SERVICES]
```

Please:

1. List every environment variable my code uses (search for process.env, import.meta.env, Deno.env, etc.) and what each is for.
2. Mark each as "safe for the browser" or "server only," and make sure server-only ones don't use a browser-exposing prefix.
3. Recommend using separate keys for development and production where the service supports it (for example Stripe test keys vs live keys) and explain why.

4. Make the app fail with a clear error message at startup if a required variable is missing, without printing its value.
5. Show me where to set these on [HOST], including which environments (production, preview, development) each one needs.
6. Update `.env.example` and `.gitignore`.

 **Tips to customize:** Preview deployments can use production secrets if you're not careful. Ask: "Should my preview deployments use test keys instead?"

SECTION 2

Database Security: Supabase & Firebase

Make sure the database itself refuses requests it shouldn't allow, not just your app's buttons.

2.1 Supabase RLS Audit

Use this when you use Supabase, especially with Lovable, Bolt, v0, or Cursor, and you've never checked RLS table by table. It's for checking every table for Row Level Security and making sure each policy does what you think.

Fill in: [SKILL LEVEL], [APP IDEA], [TABLE NAMES OR "PLEASE LIST THEM"], [WHO SHOULD SEE WHAT] (e.g. "users see only their own notes; everyone can read published posts").

Act as a database security reviewer. Audit Row Level Security (RLS) in my Supabase project. I'm a [SKILL LEVEL].

App: [APP IDEA]

Tables I know about: [TABLE NAMES OR "PLEASE LIST THEM"]

Who should see what: [WHO SHOULD SEE WHAT]

Please:

1. Explain in plain English what RLS is and why the publishable key (the anon key on older projects) being public makes it essential. Mention that tables created in the Supabase Table Editor get RLS turned on automatically, but tables created with SQL (including by an AI tool or a migration) don't unless the SQL turns it on.
2. Give me a SQL query I can run in the Supabase SQL editor that lists every table in my exposed schemas (usually public) and whether RLS is enabled.
3. Give me a query that lists all existing policies.
4. For each table, tell me: is RLS on? Do the policies match "who should see what"? Flag any table with RLS off, any policy that uses "true" for everyone, and any table with RLS on but no policies (which blocks all access).
5. For each problem, write the fix as SQL, one table at a time, using `auth.uid()` to match rows to the logged-in user. Cover select, insert, update, and delete separately.
6. Check table grants too: RLS decides which rows a role can reach, but grants decide whether the anon and authenticated roles can select, insert, update, or delete at all. Suggest revoking anything a role doesn't need.
7. Check views, functions (especially SECURITY DEFINER ones), and storage buckets, which have their own rules.
8. Give me a way to test each policy as two different users. Remember that the SQL editor runs as an admin role and skips RLS, so a query working there doesn't prove a policy works.

Don't suggest using the secret key (or legacy service_role key) in the app to "fix" access errors.

 **Tips to customize:** Supabase's dashboard also has a Security Advisor (it's also available as `supabase db advisors` in the CLI) that flags issues like tables with RLS turned off or RLS on with no policies. Run it and paste the warnings in: "Here are my Security Advisor warnings: [WARNINGS]. Explain and fix each one."

2.2 Write Safe RLS Policies for a New Table

Use this when you're about to create a new table, or your AI just created one. It's for adding a table with the right policies from the start.

Fill in: [TABLE PURPOSE] (e.g. "comments on posts"), [COLUMNS OR "SUGGEST THEM"], [WHO CAN READ] (e.g. "anyone" / "only the owner" / "members of the same team"), [WHO CAN CREATE, EDIT, DELETE] (e.g. "only the owner can edit or delete their own").

I'm adding a new table to my Supabase database. Write it with Row Level Security from the start.

Table purpose: [TABLE PURPOSE]

Columns I need: [COLUMNS OR "SUGGEST THEM"]

Who can read: [WHO CAN READ]

Who can create, edit, delete: [WHO CAN CREATE, EDIT, DELETE]

Please:

1. Write the CREATE TABLE SQL with an owner column linked to the user (if it fits) and sensible defaults.
2. Enable RLS in the same migration.
3. Write separate policies for select, insert, update, and delete, explaining each in one sentence.
4. Make sure users can't set the owner column to someone else's ID on insert or update.
5. Add indexes on columns used in policies, if helpful.
6. Give me two test queries showing an allowed and a blocked action.

 **Tips to customize:** For team or shared data, describe the relationship ("users belong to teams through a team_members table") so the AI can write membership-based policies.

2.3 Firebase Security Rules Check

Use this when you use Firestore, Realtime Database, or Firebase Storage and started in test mode, or you're not sure what your rules allow. It's for replacing "test mode" rules with rules that only let people access their own data.

Fill in: [SKILL LEVEL], [FIRESTORE / REALTIME DATABASE / STORAGE], [PASTE RULES], [MY DATA STRUCTURE] (e.g. "users/{userId}/notes/{noteId}"), [DESCRIPTION].

```
Review my Firebase security rules and help me lock them down. I'm a [SKILL LEVEL].
```

```
Products used: [FIRESTORE / REALTIME DATABASE / STORAGE]
```

```
My current rules: [PASTE RULES]
```

```
My data structure: [MY DATA STRUCTURE]
```

```
Who should see what: [DESCRIPTION]
```

```
Please:
```

1. Explain in plain English what my current rules allow, including anything open to everyone or open until a date (test mode).
2. Explain why the Firebase web config being public is normal, and why that makes rules essential.
3. Write new rules that require login and match request.auth.uid to the owner, with separate read and write rules.
4. Add basic validation for writes (required fields, types, sensible size limits).
5. Tell me how to test the rules with the Firebase Emulator or Rules Playground before publishing.
6. Mention App Check: it helps block requests that don't come from my real app, which protects against some abuse, but it complements security rules and doesn't replace them.



Tips to customize: If the AI's rules break your app, paste the exact permission error and ask which rule blocked it. Don't loosen the rules to "allow read, write: if true".

SECTION 3

Login & Auth Hardening

Make sure logged-in means what you think, and that users stay in their own lane.

3.1 Auth Settings Review

Use this when login works, and you want to make sure it isn't too open before launch. It's for checking the auth provider settings that are easy to miss.

Fill in: [SKILL LEVEL], [AUTH PROVIDER], [LOGIN METHODS] (e.g. email + password, magic link, Google), [HOST, URL].

Review the authentication setup for my app and flag risky settings. I'm a [SKILL LEVEL].

Auth provider: [AUTH PROVIDER]
Login methods: [LOGIN METHODS]
Host and live URL: [HOST, URL]

Please check and explain each in plain English:

1. Redirect URLs: are only my real domains allowed (no wildcards that allow other sites, no leftover localhost in production)?
2. Email confirmation: is it required before users can access data, and should it be?
3. Password rules and leaked-password protection, if my provider offers it.
4. Rate limits or bot protection (like CAPTCHA) on sign up, login, and password reset.
5. Session settings: how long sessions last and how logout works.
6. Whether sign ups should be open to anyone, or invite-only, for my app.
7. Whether there's an admin role, and how it's assigned (it should never be something a user can set on themselves).

For each item, tell me where to find the setting in my provider's dashboard and what you recommend.



Tips to customize: Your AI may not know your provider's latest dashboard layout. Ask it to say so if it's unsure, and check the provider's docs.

3.2 Can User A See User B's Data?

Use this when your app has user accounts and private data. It's for testing for the most common data leak: one user reading or changing another user's stuff.

Fill in: [TECH STACK], [PRIVATE DATA IN MY APP] (e.g. "notes, profile, orders"), [HOW DATA IS FETCHED] (e.g. "Supabase client from the browser", "my API routes").

Help me test whether one user can access another user's data in my app. Act as a friendly security tester and explain each step.

Tech stack: [TECH STACK]

Private data in my app: [PRIVATE DATA IN MY APP]

How data is fetched: [HOW DATA IS FETCHED]

Please:

1. List every page, API route, and database query that reads or changes private data.
2. For each, tell me where the "is this the owner?" check happens: in the database (RLS/rules), on the server, or only in the front end. Flag anything checked only in the front end.
3. Look for routes that take an ID from the URL or request body (like /api/notes/123) and check whether they confirm the logged-in user owns that ID.
4. Give me a manual test plan: create two test accounts, then try to view, edit, and delete User A's data while logged in as User B, and while logged out. Include how to do this from the browser DevTools console or with curl.
5. Fix each problem you find, one at a time, and list the files changed.

 **Tips to customize:** This goes further than Starter Pack Prompt 3.2, which sets protection up. Run this one afterwards to check it actually works.

3.3 Protect Admin Pages and Roles

Use this when your app has an admin dashboard, moderator tools, or paid-only features. It's for making sure admin features can only be used by admins, checked on the server.

Fill in: [ADMIN PAGES / PAID-ONLY FEATURES / MODERATOR TOOLS], [TECH STACK], [AUTH PROVIDER], [HOW ROLES ARE STORED NOW] (e.g. "isAdmin column in profiles table" / "not sure").

My app has [ADMIN PAGES / PAID-ONLY FEATURES / MODERATOR TOOLS]. Make sure they're properly protected.

Tech stack: [TECH STACK]

Auth provider: [AUTH PROVIDER]

How roles are stored now: [HOW ROLES ARE STORED NOW]

Please:

1. Find where roles or paid status are checked. Flag any check that happens only in the front end (hiding a button doesn't protect the action).
2. Check whether a user could make themselves an admin or paid user, for example by updating their own profile row.
3. Move role checks to the server or database, and show how to store roles where users can't edit them.
4. Protect every admin API route and database action, not just the admin page.
5. Give me a test: as a normal user, try to call each admin action directly.

SECTION 4

Bad Input & Common Web Attacks, in Plain Words

Assume someone will type or send something weird, on purpose.

4.1 Validate Everything Users Send

Use this when your app has forms, API routes, or file uploads. It's for adding server-side checks so bad or oversized input gets rejected with a clear message.

Fill in: [SKILL LEVEL], [TECH STACK], [LIST OR "PLEASE FIND THEM ALL"].

Add input validation to my app. Explain it simply, because I'm a [SKILL LEVEL].

Tech stack: [TECH STACK]

Forms and API routes: [LIST OR "PLEASE FIND THEM ALL"]

Please:

1. Explain in 2 sentences why front-end validation alone isn't enough (anyone can send requests straight to my server).
2. Find every place that accepts user input: forms, API routes, server actions, URL parameters, and file uploads.
3. Add server-side validation for each, using a schema library that fits my stack (such as Zod), checking type, required fields, length limits, and allowed values.
4. For file uploads, limit file type and size, and don't trust the file name.
5. Return friendly error messages that don't reveal internal details.
6. List every file changed and give me 3 bad inputs to test each form with.

4.2 Common Web Attacks Check (XSS, Injection, and Friends)

Use this when you're getting ready to launch, or the AI just added features that display user content or build database queries. It's for checking for the well-known attack types, explained without jargon.

Fill in: [TECH STACK].

Check my app for common web vulnerabilities. For each one, first explain in one or two plain sentences what the attack is, then check my code.

Tech stack: [TECH STACK]

Check for:

1. Cross-site scripting (XSS): user content shown as HTML without escaping (e.g. dangerouslySetInnerHTML, innerHTML, v-html, or Markdown rendered without sanitizing).
2. Injection: database queries or shell commands built by gluing user input into strings, instead of using parameters or the query builder.
3. Insecure direct object references: routes that trust an ID without checking ownership.
4. Open redirects: redirecting to a URL taken from user input.
5. CORS set to allow any origin on routes that use cookies or return private data.
6. Error messages or logs that reveal stack traces, keys, or private data to users.
7. Missing basic security headers where my host doesn't set them.

For each finding: the file and line, how serious it is (high/medium/low) and why, and the fix. If you find nothing for an item, tell me what you checked.

 **Tips to customize:** If your app has AI features, add: "Also check for prompt injection: could user text or uploaded documents trick my AI feature into revealing its system prompt, other users' data, or calling tools it shouldn't?"

4.3 Verify Webhooks (Stripe and Others)

Use this when your app receives webhooks from Stripe, Lemon Squeezy, Clerk, GitHub, or similar. It's for making sure webhook requests really come from the service, so nobody can fake a "payment succeeded" event.

Fill in: [TECH STACK], [WEBHOOK FROM] (e.g. Stripe), [FILE NAME OR "PLEASE FIND IT"].

Check that my webhook endpoint is secure.

Tech stack: [TECH STACK]

Webhook from: [WEBHOOK FROM]

Endpoint file: [FILE NAME OR "PLEASE FIND IT"]

Please:

1. Explain in plain English why anyone can send a request to my webhook URL, and why signature verification matters.
2. Verify the signature using the provider's official library and the webhook signing secret from a server-only environment variable. Make sure the raw request body is used if the provider requires it (Stripe does).
3. Reject requests that fail verification, with a 400 response.
4. Handle duplicate deliveries safely (providers retry), so the same event doesn't grant access or credit twice.
5. Confirm important details (like the amount and the product) from the verified event or the provider's API, not from data the browser sent.
6. Tell me how to test it locally with the provider's CLI or test mode.

SECTION 5

Cost & Abuse Limits

Stop one bot, bug, or curious visitor from emptying your wallet.

5.1 Add Rate Limits to My Routes

Use this when your app has public API routes, AI features, sign up forms, or anything that sends email or costs money. It's for limiting how often each user or IP can hit your API routes, especially expensive ones.

Fill in: [TECH STACK], [HOST], [EXPENSIVE OR SENSITIVE ROUTES] (e.g. "AI chat, sign up, contact form" or "please find them").

Add rate limiting to my app. Explain it simply.

Tech stack: [TECH STACK]

Host: [HOST]

Expensive or sensitive routes: [EXPENSIVE OR SENSITIVE ROUTES]

Please:

1. List every route and rank them by risk (costs money, sends email, writes to the database, public).
2. Recommend a rate limiting approach that works on my host (serverless functions don't keep memory between requests, so explain if I need a store like Redis/Upstash or a host feature).
3. Add limits per logged-in user and per IP, with sensible starting numbers I can change, and explain why you picked them.
4. Return a clear "too many requests" response (status 429) and show a friendly message in the UI.
5. Tell me how to test that the limit works.



Tips to customize: Check whether your host or provider has built-in options (such as a firewall or bot protection) that you can turn on as well.

5.2 AI Spend Caps and Usage Limits

Use this when your app calls an AI API, or you're about to share it publicly. It's for putting hard limits on AI API spending, in your code and at the provider.

Fill in: [AI PROVIDER], [TECH STACK], [AI FEATURES] (e.g. "chat assistant", "image generation"), [WHO CAN USE THEM] (e.g. "anyone" / "logged-in users" / "paid users"), [MONTHLY BUDGET] (e.g. "\$20").

My app uses [AI PROVIDER] and I want to make sure it can't run up a huge bill.

Tech stack: [TECH STACK]

AI features: [AI FEATURES]

Who can use them: [WHO CAN USE THEM]

Monthly budget: [MONTHLY BUDGET]

Please:

1. Walk me through setting a monthly spend limit in the [AI PROVIDER] dashboard, and tell me clearly whether it's a hard stop or only an alert. Some settings only send an email while requests keep going, and even hard limits can let a little extra spend through before they kick in. If you're not sure how this provider works today, say so and tell me where to check.
2. In my code: require login for AI features (if appropriate), cap input length, set a max output tokens value, and limit conversation history length.
3. Add a per-user daily usage limit stored in my database, with a friendly message when it's reached.
4. Make sure the API key is only used on the server (if not, use the "Move My API Calls to the Server" prompt (1.2 in this pack) first).
5. Add basic logging of usage per user (no message contents unless I need them) so I can spot abuse.
6. Give me a rough way to estimate cost per user from the provider's published prices, and remind me to check current prices myself.



Tips to customize: At the time of writing, OpenAI, Anthropic (Claude Console), and Google's Gemini API all let you set monthly spend limits that can stop requests. Make sure you turn on the hard limit and don't just set an alert. Dashboards change, so check your provider's docs. Do the same for other paid services (email sending, SMS, maps, image generation). Search your code for every API key and ask: "What's the worst monthly bill this could cause?"

5.3 Stop Bots and Spam Sign Ups

Use this when you see junk sign ups, spam messages, or emails sent to random addresses. It's for cutting down fake accounts and form spam.

Fill in: [FAKE SIGN UPS / FORM SPAM / OTHER ABUSE: DESCRIBE IT], [TECH STACK], [AUTH PROVIDER].

```
My app is getting [FAKE SIGN UPS / FORM SPAM / OTHER ABUSE: DESCRIBE IT]. Help me cut it down without annoying real users.
```

```
Tech stack: [TECH STACK]
```

```
Auth provider: [AUTH PROVIDER]
```

```
Please:
```

1. Explain the likely cause in plain English.
2. Suggest options from least to most annoying for users: rate limits, honeypot fields, email confirmation, CAPTCHA or invisible challenges (like Cloudflare Turnstile or hCaptcha), and blocking disposable email domains.
3. Implement the two you'd recommend for my app, with server-side checks (not front end only).
4. Tell me how to clean up existing fake accounts safely.

SECTION 6

Packages & Dependencies

Code you didn't write is still code you ship.

6.1 Check That AI-Suggested Packages Are Real

Use this when your AI adds a package you've never heard of, before you install it. It's for avoiding fake or look-alike packages, since AI tools sometimes suggest package names that don't exist and attackers can register those names (sometimes called "slopsquatting").

Fill in: [PACKAGE NAMES], [NPM / PNPM / YARN / PIP / OTHER].

```
Before I install these packages, help me check they're real and trustworthy.
```

```
Packages the AI suggested: [PACKAGE NAMES]
```

```
Package manager: [NPM / PNPM / YARN / PIP / OTHER]
```

```
For each package, tell me how to check (and check yourself if you can browse):
```

1. Does it exist on the official registry, under exactly this name? Watch for typos and look-alike names.
2. Is it the well-known package for this job, or would a more popular, maintained option do the same thing?
3. How to look at weekly downloads, the linked source repo, recent updates, and the number of maintainers.
4. Whether it runs install scripts, and why that matters.
5. Whether I need it at all, or my framework already does this.

```
If you can't verify something, say "I couldn't verify this" instead of guessing.
```



Tips to customize: Add to your rules file (Starter Pack Prompt 1.3): "Ask before adding any new dependency, and tell me why it's needed."

6.2 Dependency Health Check

Use this when you're getting ready to launch, and again about once a month. It's for finding packages with known vulnerabilities and updating them safely.

Fill in: [SKILL LEVEL], [NPM / PNPM / YARN / PIP], [GITHUB / OTHER / NONE].

```
Help me check my project's dependencies for known security issues and update them safely. I'm a [SKILL LEVEL].
```

```
Package manager: [NPM / PNPM / YARN / PIP]
```

Repo host: [GITHUB / OTHER / NONE]

Please:

1. Give me the exact audit command for my package manager and explain how to read the output (severity, direct vs. indirect dependencies).
2. For each high or critical issue, tell me whether it actually affects my app (for example, a dev-only tool vs. code that runs in production).
3. Suggest the safest fix for each: a patch update first, and warn me before any major version upgrade that could break things.
4. Tell me to commit before updating, then run the app and test the main features afterward.
5. Make sure my lockfile is committed.
6. Explain how to turn on automatic alerts (such as Dependabot on GitHub).

 **Tips to customize:** Don't run "npm audit fix --force" without reading what it changes. It can install breaking major versions.

SECTION 7

Pre-Launch Checks & "Prove It's Done"

The last 20% is where the security gaps hide.

7.1 Full Pre-Launch Security Review

Use this when you're about to launch, or about to start charging money. You get one structured review of your whole app before you share it, producing a prioritized fix list.

Fill in: [SKILL LEVEL], [APP IDEA], [TECH STACK], [HOST], [DATABASE], [AUTH PROVIDER], [SERVICES], [SENSITIVE DATA I STORE] (e.g. "emails, private notes, payment status").

Act as a senior security reviewer doing a pre-launch review of my app. I'm a [SKILL LEVEL], so explain findings in plain English. Don't change any code yet.

App: [APP IDEA]
Tech stack: [TECH STACK]
Host: [HOST]
Database: [DATABASE]
Auth: [AUTH PROVIDER]
Paid services: [SERVICES]
Sensitive data I store: [SENSITIVE DATA I STORE]

Review these areas and, for each, say what you checked and what you found:

1. Secrets: any in front-end code, in the repo, or in Git history?
2. Database access: RLS / security rules on every table or collection, tested for two users.
3. Auth: redirect URLs, email confirmation, admin roles checked on the server.
4. Every API route: login check, ownership check, input validation, rate limit.
5. Payments and webhooks: signature verification, amounts confirmed on the server.
6. Cost exposure: AI and other paid APIs capped in code and at the provider.
7. Common web issues: XSS, injection, open CORS, leaky error messages.
8. Dependencies: known vulnerabilities, unfamiliar packages.
9. Privacy basics: what personal data I collect, and whether I need a privacy policy.

End with a table: issue, severity (high/medium/low), file or setting, and the fix. Order it by severity. Then tell me what you could NOT check from the code alone, so I can check it myself.

 **Tips to customize:** Fix high-severity items one at a time with the matching prompts in this pack. This goes deeper than Starter Pack Prompt 6.3 (Launch Readiness Checklist). Use both.

7.2 Make It Prove It's Done

Use this when the AI just finished a fix, especially a security fix. It's for stopping the AI from saying "done" or "fixed" without actually checking.

Fill in: [DESCRIBE OR "THE LAST CHANGE"].

Before we call this finished, prove it works.

What you just changed: [DESCRIBE OR "THE LAST CHANGE"]

Please:

1. List every file you changed, with a one-line summary of each change.
2. Tell me exactly what you ran or checked to confirm it works. If you didn't run anything, say so plainly.
3. If you can run commands, run the app, the build, and any tests now, and show me the actual output.
4. Give me a short manual test I can do myself, including one test that should FAIL (for example, a blocked user trying to access data) to show the protection works.
5. Tell me anything you're unsure about or couldn't verify.

Don't say "done" until steps 1-5 are complete.



Tips to customize: For security fixes, the "should fail" test matters most. A lock that you've never tried to open while locked out hasn't really been tested.

7.3 Safe Logging and Error Messages

Use this when you're getting ready to launch, or you see full error details on your live site. It's for making sure errors help you debug without leaking secrets or user data.

Fill in: [TECH STACK], [HOST].

Review how my app handles errors and logging.

Tech stack: [TECH STACK]

Host: [HOST]

Please:

1. Find places where errors are shown to users and make sure users see a friendly message, not stack traces, SQL, or internal details.
2. Find logs that might print secrets, tokens, passwords, or full user data, and remove or mask them.
3. Make sure useful details still go to the server logs so I can debug.
4. Tell me where to find my logs on [HOST].

5. Suggest a simple error tracking option, if I want one, and what data it would collect.

SECTION 8

Oops, I Leaked Something

It happens. Act fast, in the right order.

8.1 I Leaked an API Key: What Do I Do Right Now?

Use this when a key was committed to Git, shown in browser code, pasted somewhere public, or flagged by a scanner or provider email. You get a calm, ordered checklist for a leaked key.

Fill in: [WHICH KEY] (e.g. "OpenAI API key", "Supabase secret key", "Stripe secret key"), [WHERE IT WAS EXPOSED] (e.g. "public GitHub repo", "front-end code on my live site", "posted in a chat"), [TIME], [HOST].

I think I leaked a secret and I need step-by-step help, in the right order. Keep it calm and simple.

Which key: [WHICH KEY] (don't include the value)

Where it was exposed: [WHERE IT WAS EXPOSED]

For roughly how long: [TIME]

Where the app is deployed: [HOST]

Please give me:

1. Step 1, right now: how to revoke or rotate this specific key at the provider and create a new one. Tell me where to find this in the provider's dashboard (if you're unsure of the current layout, say so and point me to the docs).
2. How to put the new key into my host's environment variables and redeploy, so the app keeps working.
3. How to check the provider's usage, logs, or billing for activity I didn't cause.
4. Only after rotating: how to remove the key from my code and, if needed, Git history (and why rotating matters more than cleaning history).
5. Whether I should contact the provider's support (for example, about unexpected charges).
6. How to stop this from happening again (secret scanning, pre-commit hooks, server-only keys).

 **Tips to customize:** If the leaked key is a database admin key (like a Supabase secret key or legacy service_role key), also review your data for unexpected changes and read Prompt 8.2.

8.2 Someone Might Have Accessed My Data

Use this when you find RLS was off, rules were open, or you see data changes you didn't make. It's for figuring out what happened and what to do when you suspect a data exposure.

Fill in: [WHAT I FOUND] (e.g. "RLS was off on the profiles table for 2 weeks"), [DATA INVOLVED] (e.g. "emails and display names"), [DATABASE, HOST], [NUMBER OR "NOT SURE"].

I think user data in my app may have been exposed. Help me respond calmly and properly.

What I found: [WHAT I FOUND]

Data involved: [DATA INVOLVED]

Database / host: [DATABASE, HOST]

Roughly how many users: [NUMBER OR "NOT SURE"]

Please:

1. First, help me close the hole now (turn on RLS / fix rules / rotate keys), with exact steps.
2. Show me which logs my database and host provide, and how to look for unusual access during the exposure window. Be honest about what the logs can and can't tell me.
3. Help me make a simple written timeline of what happened.
4. Remind me: this isn't legal advice, check your local rules.
5. Help me draft a short, honest notice to users, in case I need one.
6. List what to change so it doesn't happen again.

8.3 Post-Incident Cleanup Checklist

Use this when the emergency is handled and you want to tie up loose ends. It's for making sure nothing is left over after you've fixed a leak.

Fill in: [SHORT DESCRIPTION], [TECH STACK], [SERVICES].

I've handled a security incident: [SHORT DESCRIPTION]. Help me do a thorough cleanup.

Tech stack: [TECH STACK]

Services: [SERVICES]

Please give me a checklist that covers:

1. Every key that might be affected (not just the one I noticed), and whether each has been rotated.
2. Old keys deleted at each provider, not just replaced.
3. Copies of the secret in other places: forks, clones, CI settings, local .env files on other computers, chat logs, screenshots, docs.
4. Secret scanning and push protection turned on.
5. Billing alerts and spend caps set.

6. A short note in my project brief or rules file (for example "never put keys in front-end code") so my AI tool remembers it.

Bonus: The Security "Magic Phrases" Cheat Sheet

Add these to the end of any prompt:

- **"Show me the exact file, line, or setting."** Turns vague reassurance into something you can check.
- **"Don't print full secret values."** Keeps keys out of your chat history.
- **"Check this on the server or database, not just the front end."** Hiding a button isn't protection.
- **"Give me one test that should fail."** Proves the lock actually locks.
- **"What could you NOT check from the code alone?"** Surfaces dashboard settings and blind spots.
- **"If you couldn't verify it, say so."** Cuts down on confident guesses.
- **"Rotate first, clean up second."** The right order for any leaked key.

You're Ready to Ship (More Safely)

That's the whole Ship It Safe pack: **24 prompts across 8 sections**, from hidden keys and open databases to spend caps and what to do when something leaks.

No pack can make an app perfectly secure, but checking these common mistakes puts you well ahead of shipping blind. Run the Quick Start today, fix one thing at a time, and make your AI prove its work.

Happy (and safe) vibe coding!