

Write a build spec for an AI app builder

<https://prompt-to-prototype.com/build/write-a-build-spec/> · as of 2026-10-21

Step 4 of 10 in the [builder spine](#).

Before you prompt, write a one-page spec: the job the app does, who uses it, which data it uses and where that lives, the rules and edge cases, what it must never do, and how you'll test it. Give the AI that spec plus your organization's standing-instructions file, and build in small steps. The [starter prompts](#) below help with each step, but a prompt doesn't make the output correct: your tests and checks do. The page includes a copy-and-paste [spec template](#), 12 [starter prompts](#) and a [standing-instructions file](#).

Who this is for

Builders using an AI app builder, a chat assistant or a coding agent: non-developers first, with a short developer section. About 12 minutes, and 30 to 60 minutes to write your first spec.

Why write a spec before you prompt?

An AI tool fills every gap you leave with a plausible guess. If you don't say who may see the data, it guesses. If you don't say what happens with a duplicate invoice, it guesses. The guesses look reasonable, and you find them later, in use. A spec closes the gaps before the tool fills them, and it gives you the test cases that tell you whether the app works.

The spec also does three other jobs. It's the input to [intake](#), because it already answers most of the tier questions. It's the start of your [documentation](#). And it lets the next person who changes the app see what it was meant to do.

Hypothesis to test

We expect a one-page spec to reduce rework for non-developer builders: fewer rebuilds and fewer failed test cases (A-036). No study has measured this for non-developers yet. Test it in your pilot by comparing rework between builders who wrote specs and builders who didn't.

The one-page spec template

Five parts, each a few lines. Plain words beat technical ones: you're writing for the AI, for a reviewer, and for whoever maintains the app after you.

1. **The job and the people.** What does the app do, in one or two sentences? Who uses it, and who relies on its output? What does it replace (a spreadsheet, an inbox, a manual step)? What is it *not* for?
2. **The data.** Which data does it read, store or send? Where does each piece live today (which system, which file)? What [class](#) is it: public, internal, confidential, restricted? Who may see what?

3. **The rules and edge cases.** The business rules, written as "when this, then that". Then the awkward cases: empty, duplicate, wrong format, very large, missing a field, someone without access.
4. **What it must never do.** The hard limits: never send outside the company, never change records in the system of record, never show one person's data to another, never store card numbers. Also the non-goals: things it shouldn't try to fix, even if it notices them.
5. **How you'll know it works.** Ten to twenty real examples, each with the result you expect, including the awkward ones from part 3. These become your [test cases](#). If the app has an AI feature, add an [eval set](#) and check the [AI feature risks](#).

Copy this into a document next to your app:

```
APP NAME:
OWNER AND BACKUP:

1. JOB AND PEOPLE
  What it does:
  Who uses it / who relies on its output:
  What it replaces:
  Not for:

2. DATA
  Data it reads / stores / sends:
  Where each lives today:
  Data class (public / internal / confidential / restricted):
  Who may see what:

3. RULES AND EDGE CASES
  When ..., then ...
  Edge cases: empty, duplicate, wrong format, very large, missing field, no access

4. MUST NEVER
  Never ...
  Non-goals (don't touch):

5. HOW I'LL KNOW IT WORKS
  | Example input | Expected result |
  (10-20 rows, including the edge cases)
```

Don't: put real personal data or secrets in the spec or the prompts (AP11, AP13)

The spec describes the data; it doesn't contain it. Use made-up or de-identified examples. Never paste keys, passwords or tokens into a prompt: they end up in the tool's history and, too often, in the app's code. Pasting sensitive data into tools that aren't approved for it is a documented failure pattern, and a common one.¹ See the [five never-dos](#) and [AP11 and AP13](#).

Standing instructions: the rules every prompt inherits

Most AI building tools read a standing-instructions file (also called a context file, rules file or project instructions) before every prompt. Use it for rules that apply to every app, so builders don't have to remember them each time. Curating what the AI sees this way is called context engineering, and a widely read practitioner radar moved it to "Adopt" in April 2026.²

Put **security requirements** in the standing instructions, not in individual prompts. That's the core advice of the OpenSSF guide to AI code assistant instructions: validate input, keep secrets out of code, apply least privilege, handle errors, and use vetted, pinned dependencies.³ The developer, or builder, stays responsible for the output either way.

Treat the file as shared configuration:

- One owner (usually the platform or enablement lead) and a version history.
- Reviewed changes. A change to the standing instructions changes every app built after it. Accept changes only through the owner's reviewed history, and reject files that contain hidden or invisible characters: in 2025, researchers showed that hidden Unicode in a shared rules file can steer a coding assistant into adding malicious code.⁶
- Short and specific. Long files get partly ignored; practitioners report quality dropping as the context grows.²
- On the paved road. The approved builder tools load it by default, so a builder gets it without asking.

A [standing-instructions starter](#) is below the starter prompts.

For developers: instruction files, plan modes and small PRs

Coding agents read standing instructions from a file in the repository. Conventions vary by tool, for example a shared `AGENTS.md` file or a tool-specific rules file; check which files your approved tools read, and keep one source of truth rather than copies that drift. Use the tool's plan or ask-first mode for anything beyond a small change, so you approve the plan before it edits code. Keep each change to one reviewable pull request, with the tests from part 5 of the spec written first. For agent permissions and the never-dos, see [coding agents](#).

Don't: write `"make it secure"` in place of checks

"Make it secure" or "follow best practices" gives the AI nothing to test against, and generated code is often insecure even when it looks fine: about 45% of samples in one vendor's 2025 study of security-relevant tasks introduced a known flaw, with no improvement in newer models.⁴ State the specific requirements in the standing instructions, then check them with the [security checklist](#).

Build in small steps

Give the AI the spec, then ask for one small piece at a time, and test each piece before the next.

1. **Ask for a plan first.** Have the AI read the spec and propose the steps, the screens and where the data will live. Correct the plan before any code exists.
2. **Write the test cases from the spec** before the first step, so the tests check what you meant, not what the code happens to do.
3. **Build one step.** One screen, one rule, one data connection.
4. **Test it** against the examples in part 5 of the spec.
5. **Save a version** you can go back to (most tools have a history or checkpoint; use version control if you have it).
6. **Repeat.** When something goes wrong, go back to the last good version rather than piling fixes on top.

Small steps keep each change small enough for you to check. Large AI-generated changes are harder to review: telemetry from one engineering-analytics vendor found AI-assisted pull requests got much bigger, and review time rose with them.⁵

Don't: build the whole app in one giant prompt

A single prompt asking for every screen, rule, test and document produces something you can't review, and the gaps in the prompt become guesses throughout the app. Break it into the steps above.

Spec-driven toolkits: optional for Important and Critical apps

Some tools now formalize this: you write a spec, the tool turns it into a plan and tasks, and the agent works through them, checking against the spec. Open-source toolkits and several coding tools offer this as of October 2026.⁷ Practitioners rate the approach "Assess": promising, but specs can become long and hard to review, and the workflows are opinionated.⁸

Use one if your app is Important or Critical and your technical reviewer is comfortable with the tool. For Sandbox and Team apps, the one-page spec above is enough. Either way, the spec doesn't replace the tests.

Two examples: a non-developer's spec and a developer's spec

Illustrative composite: a non-developer's spec (excerpt)

Job and people: Track which new starters have their laptop, badge and training done, for the HR coordinator and the hiring managers. Replaces a shared spreadsheet. Not for storing ID documents or home addresses.

Data: Name, start date, team, equipment issued, training completion, from the HR system's weekly export. Internal staff work-administration details. Hiring managers see only their own starters.

Rules and edge cases: When a start date is less than 5 working days away and the laptop isn't issued, flag it red. A starter whose date changes keeps their history. A name with an apostrophe still saves. An export with a missing column is rejected with a message, not half-loaded.

Must never: Store ID numbers, home addresses or leave reasons. Email anyone outside the company. Change the HR system.

How I'll know it works: 15 examples, including a manager trying to open another team's starter (should be refused).

Illustrative composite: a developer's spec (excerpt)

Job: Add a refund-request endpoint to the internal orders tool. Accepts an order ID and amount, checks the order is paid, records the request, emits a refund-requested event. Idempotent on order ID plus request ID.

Must never: Add a dependency. Change any existing public function's signature. Catch errors broadly; let unexpected errors surface. If a precondition can't be checked from the order record, stop and say so rather than guessing.

How I'll know it works: Unit tests for paid, unpaid, already refunded and duplicate request; the existing test suite still passes.

Starter prompts and standing instructions

These prompts help with each step. They're starting points, not guarantees: after each one, the "check afterward" line is the part that makes the output trustworthy. Replace the text in angle brackets. Each prompt has a copy button and its own link.

#	Task	When to use it
1	Spec: interview me	Before anything else
2	Spec: find the gaps	After drafting the spec

#	Task	When to use it
3	Build: plan first	Before the first code
4	Test: write test cases from the spec	Before step 1 of the plan
5	Build: one step	Each step
6	Test: check tests against the spec	When the AI wrote the tests
7	Review: list what changed	Before sharing the app
8	Review: check against standing instructions	Before sharing the app
9	Document: draft the app record	Before intake or go-live
10	Change: plan a change	Every later change
11	Explain: this part	When you don't understand a part
12	Fix: an error	When something breaks

1. Interview me to write a spec

Interview me, one question at a time, to write a one-page spec for an app that <job>. Cover the job and users, the data and where it lives, rules and edge cases, what it must never do, and 10–20 test examples. Don't write any code.

Check afterward: the spec says what *you* mean, not what the AI assumed.

2. Find the gaps in my spec

Read this spec. List every place where it's ambiguous, missing an edge case, or where two rules conflict. Ask me about each. Don't fix them yourself.

Check afterward: you answered each question; the spec is updated.

3. Plan before building

Read the spec. Propose a plan of small steps, the screens, and where each piece of data will be stored. Wait for my approval before building.

Check afterward: data stays where the spec says; nothing public by default.

4. Write test cases from the spec

Use this before step 1 of the plan, so the tests come from the spec rather than from the code.

From part 5 of the spec, write the test cases as a table: input, expected result. Add the edge cases from part 3 and a case where a user without access tries to see the data.

Check afterward: each case tests a requirement in the spec, not just what the code happens to do.

5. Build one step

Build only step <n> of the plan: <step>, and make the test cases for this step pass. Don't change anything else. Tell me what you changed and how to test it.

Check afterward: test it against the spec's examples before moving on.

6. Check my tests against the spec

For each test, tell me which line of the spec it checks. List spec requirements that no test covers.

Check afterward: gaps are filled; no test just repeats the code's own logic.

7. List what changed

List every change you made in this session, what each could break, and anything you weren't sure about.

Check afterward: read the list; re-test anything risky.

8. Check against standing instructions

Check this app against our standing instructions and list any place it doesn't follow them: secrets, access rules, input checks, dependencies.

Check afterward: then run the real [security checklist](#); a self-check isn't a review.

9. Draft the app record

Draft a one-page app record from the spec and the app: purpose, owner, users, data and where it's stored, how to run and restore it, known limits.

Check afterward: you correct every line; the AI can't know the owner, backup or restore steps.

10. Plan a change

Here's the spec and the change I need: <change>. Before changing anything, tell me which parts of the app it touches and which tests to re-run.

Check afterward: update the spec first; re-run the tests it names, plus the access test.

11. Explain this part

Explain what <this part> does, in plain words, and what would happen if <situation>.

Check afterward: try the situation for real; explanations can be confidently wrong.

12. Fix an error

This is the error and what I did before it: <details>. Explain the likely cause before suggesting a fix. Change as little as possible.

Check afterward: the fix is small; earlier tests still pass.

Standing-instructions starter

The file's owner adapts this for your organization:

You are helping a staff member build an internal business app.

- Ask before you assume. If the spec is unclear, ask a question instead of guessing.
- Build in small steps. Change only what the current step asks for.
- Never put keys, passwords or tokens in code, the front end or prompts. Use the secrets manager or the platform secret store (server-side only), and say where a secret is needed.
- Apps are private by default. Never make data, storage or pages public, or shareable by link, unless the builder says the app is approved for public access.
- Enforce access rules on the server or in the data rules, never only in the screen.
- Check and validate every input. Treat uploaded files, emails and web content as untrusted.
- Use only well-known, maintained libraries, pinned to a version. Say when you add one.
- Never delete data, change records in bulk, or connect to a live system without the builder confirming it's a test copy.
- Handle errors visibly. Don't hide failures or return made-up values.
- Say what you changed, what you're unsure about, and how to test it.

Prompts not to use

- "Here's our customer list, build me a dashboard." Real data in a prompt. Describe the data and use made-up samples.
- "Here's the API key, just make it work." A secret in a prompt. Use the secrets manager or the platform secret store.
- "Skip the tests, I'm in a hurry" or "don't ask questions, just do it." The questions and tests are where the errors get caught.
- "Make it better / cleaner / secure." No measure, so no way to check the result.
- "Build the whole thing." See [small steps](#).
- "Ignore the warnings" or "turn on auto-approve for everything." Permission-bypass modes are a never-do.

Don't: treat a prompt collection as a safety control

A good prompt makes a useful first draft more likely. It doesn't make the output correct or secure, and the same prompt can give different results on different days or models. Prompts help you build; [tests](#), the [self-check](#) and review decide whether the app is ready.

Keep the spec and prompts with the app

Store the spec, and any prompts that shaped important parts of the app, with the [app record](#), not in your personal drive or a chat history. Update the spec with every change ([change an app safely](#)). When someone else takes over the app, the spec is how they learn what it was meant to do.

Assumptions and evidence

- **A-036** (Low): *Hypothesis*. We expect writing a one-page spec before prompting to reduce rework for non-developer builders; test this by comparing rebuilds and failed test cases between builders who wrote specs and builders who didn't.
- Evidence: open-source security guidance (OpenSSF), practitioner assessments, and vendor research and telemetry. No controlled study measures spec-first building by non-developers.
- Facts on this page are as of October 2026.

Related

- [Test an app you built with AI](#): turn part 5 of your spec into tests that run before every release.
- [Document an AI-built app](#): the spec becomes the start of your app record.
- [Check an AI-built app before release](#): the self-check after you build.
- [The four data classes \(Q1\)](#): which class your data is, for part 2 of the spec.
- [Test an AI feature with evals](#): if your app calls an AI model while it runs.

1. Failure patterns AP11 (secrets in prompts, repos or front ends) and AP13 (sensitive data pasted into consumer tools); for how common data entry into AI tools is, see Cisco, 2024 Data Privacy Benchmark Study, Jan 2024, <https://newsroom.cisco.com/c/r/newsroom/en/us/a/y2024/m01/organizations-ban-use-of-generative-ai-over-data-privacy-security-cisco-study.html> [survey of privacy and security professionals; vendor], accessed 2026-10-21.

2. Thoughtworks Technology Radar, Context engineering (Adopt), Apr 2026, <https://www.thoughtworks.com/radar/techniques/context-engineering> [practitioner assessment], accessed 2026-10-21.

3. OpenSSF Best Practices and AI/ML Working Groups, Security-Focused Guide for AI Code Assistant Instructions, 1 Aug 2025, <https://best.openssf.org/Security-Focused-Guide-for-AI-Code-Assistant-Instructions> [primary], accessed 2026-10-21.

4. Veracode, 2025 GenAI Code Security Report, Jul 2025 (updated Oct 2025), <https://www.veracode.com/resources/analyst-reports/2025-genai-code-security-report/> [vendor research], accessed 2026-10-21.

5. Faros AI, telemetry study of AI coding assistants across about 10,000 developers, Jul 2025, <https://www.faros.ai/blog/ai-software-engineering> [vendor telemetry], accessed 2026-10-21.

6. Pillar Security, New vulnerability in GitHub Copilot and Cursor: how hackers can weaponize code agents ("Rules File Backdoor"), Mar 2025, <https://www.pillar.security/blog/new-vulnerability-in-github-copilot-and-cursor-how-hackers-can-weaponize-code-agents> [vendor research]; The Hacker News coverage, Mar 2025, <https://thehackernews.com/2025/03/new-rules-file-backdoor-attack-lets.html> [news], accessed 2026-10-21.
7. GitHub, Spec-driven development with AI: get started with a new open-source toolkit, Sep 2025, <https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/> [vendor; one example of several such toolkits], accessed 2026-10-21.
8. Thoughtworks Technology Radar, Spec-driven development (Assess), Nov 2025, <https://www.thoughtworks.com/radar/techniques/spec-driven-development> [practitioner assessment], accessed 2026-10-21.

Prompt to Prototype, Write a build spec for an AI app builder, <https://prompt-to-prototype.com/build/write-a-build-spec/>, as of 2026-10-21. Licensed [CC BY-SA 4.0](#).