FOR SMALL-BUSINESS OPERATORS BUILDING WITH AI · V1.0 · FREE

Your AI Built You Something Pretty

(And It Doesn't Work.) Why your tenant portal, inventory tool, or customer dashboard collapsed in month two — and how to actually plan one that survives.

> *The reason it broke isn't the AI. It's that you asked it to build before you decided what to build. Vibe coding without architecture is the small-business version of pouring concrete before drawing the foundation.*

BY	Alex Jahn
EDITION	v1.0 — May 2026
AUDIENCE	Small-business operators building internal tools with AI
READING TIME	~20 minutes
COST	Free

The "pretty plus broken" problem

You've seen this build, or done this build. A friend, a partner, or you yourself spend a month with an AI assistant putting together an internal tool. A tenant portal. A job-tracker. An inventory dashboard. You can describe it in two sentences. The AI is helpful. Every prompt gets a result. The screens look good — rounded corners, tasteful colors, animated transitions. You demo it to your spouse and they nod.

Then you actually try to use it. Submitting a form errors out half the time. The login forgets you. Two pages show different numbers for the same record. You ask the AI to fix it; it adds three more files and the error moves but doesn't go away. Another week goes by. Now there are 47 files, you can't tell which ones matter, and every "fix" introduces two new bugs. You don't know how to deploy it. You don't know if the data is being saved or just displayed. You eventually stop touching it.

This is not a rare outcome. It's the *default* outcome when a non-engineer drives an AI assistant straight at a multi-screen web app without a plan. The AI didn't fail you. It did exactly what you asked. The problem is that "**build me a tenant portal**" is not enough information for any builder — human or AI — to produce a working result. The architecture didn't exist before the code did, so the code grew without one.

> A pretty UI is the last 10% of a working app. The other 90% is plumbing the AI didn't know to put in because you didn't tell it to.

HI, I'M ALEX. I'm a working carpenter and house-flipper in Fond du Lac, Wisconsin. I run a multi-agent operations stack for my own trade business — proposals, scheduling, jobsite notes, comp lookups, customer follow-ups. I've built most of it with AI assistants, and I've made every mistake in this brief at least once. The shortcut I wish I'd had on day one is the order of operations — the part that's not in any vibe-coding tutorial. I run Agent Logic, where I help other small-business operators build the same kind of stack for their own work.

Why this happens (the mechanism)

It's tempting to blame the AI. The AI didn't blame itself, and it isn't lying when it says "I've fixed the issue." The mechanism is simpler and more important to understand than that.

An AI assistant has no memory between prompts

Every time you send a message to Claude, ChatGPT, Cursor, or any other AI, the model gets your message plus whatever conversation history is loaded in that window. **It does not remember the architectural decisions you made in last Tuesday's session.** It doesn't know that the database column you renamed three days ago broke an endpoint someone (it) wrote five days ago. It doesn't know that you decided users would log in with email + password before you also decided they'd log in with Google. It can *guess* based on what's in the current context, and it'll write something plausible. But plausible is not the same as consistent with the rest of the codebase.

So the architecture has to live *outside* the AI

Carpenters know this instinctively, even if they don't call it architecture. You don't frame a wall and *then* decide where the doors go. You draw the floor plan first, on paper, and the framing follows. The floor plan exists outside any one carpenter's head. Anyone who walks onto the site can read it and continue the work without re-deciding what kind of house this is.

Software is the same. The plan — data model, who logs in, what each page does, where it runs — has to exist *outside* the AI. In a document. In your head, written down. Pinned to the project. Every prompt the AI gets has to be a step that fits inside the plan, not a step that re-invents it.

What "vibe coding" actually is

"Vibe coding" is the name for prompting an AI to build software by describing the vibe you want, then iterating until it works. For prototypes and one-off scripts, this is fine and even fast. For anything someone is going to *use* — meaning anything with users, data, login, or money — vibe coding without an outside-the-AI plan produces the pretty-plus-broken outcome described on the last page. It's not a moral failing. It's a structural one. The plan didn't exist; the code couldn't follow it.

> **ONE SENTENCE:**
> The AI doesn't remember what you decided yesterday, so the architecture has to live outside the AI — in a plan you keep handing it back — or every prompt is a fresh start that contradicts the last ten.

The architectural decisions that matter (before any code)

Before you ask an AI to write a single line of code, seven decisions need to live in your plan. They are not optional. If you skip any of them, the build will compensate by guessing, and the guesses will contradict each other.

1. The data model

What "things" does this app store? A tenant, a unit, a payment, a maintenance request? For each thing, what fields? Which fields are required? What makes one row different from another (the natural unique identifier — not just an auto-incrementing ID)?

2. Who logs in (auth)

Who has accounts? Tenants? Landlords? Both? Just you? How do they log in — email + password, magic link, Google sign-in, phone number? What can each type of user see? **Auth bolted on at the end always leaks** because the original code assumed every request was the same user.

3. The API contract

The list of "things the screen can ask the back-end to do." Get a list of tenants. Create a payment. Delete a maintenance request. For each one, what does it take in, what does it give back, and who's allowed to call it? The API contract is the floor plan for how data moves.

4. Where it runs (deployment target)

Web only, mobile, both? On your laptop, on a server you rent, on a managed service like Vercel or Render? This matters *before* you write code because some choices (file storage, background jobs, sessions) work on one platform and not on others.

5. State boundaries

What does the front-end know about, and what does it have to ask the back-end for? Anything the user has to be able to refresh and not lose — saved data, logged-in status, pagination cursors — lives on the back-end. Anything that's just "what's currently on screen" stays on the front-end. Confusing the two is how identical pages start showing different numbers.

6. File / folder structure

Sounds boring. Matters most. The pretty-plus-broken builds always have 47 files in one folder by month two. A simple convention — one folder for the data model, one for API routes, one for pages, one for shared utilities — lets the AI find what already exists instead of re-inventing it five times.

7. Error-handling philosophy

What happens when something fails? The form submission, the database save, the network call to a third-party service. Each one will fail eventually. *Show the user a message and stop, or retry silently and log it?* Decide once at the start. If you don't, the AI will pick differently in every file.

> **HEURISTIC:**
> If you can't answer all seven of these in plain English without opening your IDE, stop and write the answers down before the next prompt. The plan is the document the AI doesn't have but pretends to.

The sequence that doesn't collapse

The other half of the failure mode is order. Most builds that go wrong started with a UI and tried to wire it up later. The order has to flip. Build the boring plumbing first while it's cheap to change; build the pretty UI last, on top of plumbing that already works.

Phase	What you produce	Why it's first / last
1. Spec	One page, plain English. What the app does for one user. Verbs.	If you can't write this page, the app isn't real yet. Stop.
2. Data model	The list of "things" the app stores, in plain English. Fields, relationships, identifiers.	Every screen and every endpoint depends on this. Wrong here = re-do everything.
3. API contract	Plain-English list of every back-end endpoint. Verb, path, inputs, outputs, who's allowed.	The screens and the back-end can be built in parallel only if both sides agree on this contract.
4. Backend	Database + API endpoints, returning real data. No login yet.	Real data through real endpoints proves the data model works before the UI hides bugs.
5. Frontend	One page at a time, plain styling, hooked to real endpoints.	Each page proves the API contract end-to-end. Plain styling on purpose.
6. Auth	Real login, separate middleware, applied consistently across endpoints.	Bolted on now, after endpoints exist, so the auth layer is one reviewable thing — not scattered through every file.
7. Deploy	App running at a public URL. The same app the user demos.	Things break in production that don't break locally. Find them now, while there are no real users.
8. Polish	Now make it pretty.	Pretty on top of working is design work. Pretty on top of broken is theater.

> Plain styling for the first working version. On purpose. The goal of phase 5 is to prove the plumbing works — not to impress anyone. Pretty comes after working.

THE DISCIPLINE:
Each phase has one deliverable, and you don't move to the next phase until that deliverable exists in writing. The seven prompts on the next pages do exactly this — they force the AI to produce the right deliverable for the right phase, in the right order.

The seven drop-in prompts

The next four pages contain seven prompts, one per phase. Copy any of them into your AI assistant exactly as written. Each one is built to:

- State the role explicitly — planner first, coder later
- Forbid skipping ahead (no code in the planning phases)
- Force the AI to ask *you* clarifying questions before producing the deliverable
- Name the deliverable shape so the output is reviewable

You can use them one at a time as you go, or paste this whole document to your AI as context and say "*use the framework in this PDF to help me build my app, starting at phase 1*." Either works.

PHASE 1 — ONE-PAGE SPEC

```
You are helping me plan a small web app before we write any code. Do not write any code in this response
— your only job here is to help me produce a one-page spec.

The app I want to build, in plain English: [REPLACE THIS WITH 2-3 SENTENCES DESCRIBING YOUR APP]

Ask me up to 5 clarifying questions — one at a time, waiting for my answer between each — until you have
enough to draft the spec. The spec must include:

- The single sentence describing what this app does for one user
- The 3-5 things a user can DO in this app (verbs, not features)
- The data this app needs to store (in plain English, no schema yet)
- Who logs in, and what different types of users (if any) can see different things
- Where this will run (web only? mobile? both?)
- What "done enough to ship" means for v1 — what's IN, what's explicitly OUT

Do not propose a tech stack yet. Do not write code. Do not produce a roadmap. The output is one page of
plain English, nothing else.
```

Save the AI's final spec to a file. You'll paste it into the next prompt.

PHASE 2 — DATA MODEL

```
You are helping me design the data model for the app described in the spec below. Do not write any code
yet — your only job here is the data model in plain English.

[PASTE YOUR SPEC FROM PHASE 1]

Walk me through, in plain English:

1. The list of "things" (entities) this app needs to store. For each one, list the fields it has and
which are required.
2. How those things relate to each other (one-to-many, many-to-many) — explain in sentences, not arrows.
3. For each entity, the natural unique identifier (what makes one row different from another in real life
— not just an auto-incrementing ID).
4. For each entity, the timestamps that matter (created_at, updated_at, anything else).

Then ask me 3-5 clarifying questions about edge cases I probably haven't thought of (deletes, history,
multi-tenancy, soft-delete vs hard-delete, etc.). Wait for my answers before producing the final version.

Output is a plain-English data dictionary. No SQL. No code.
```

PHASE 3 — API CONTRACT

```
You are helping me design the API contract — the list of "things the frontend can ask the backend to do."
No code yet.

Spec: [PASTE]
Data model: [PASTE FROM PHASE 2]

Produce, in plain English:

1. The list of API endpoints — verb (GET/POST/PUT/DELETE), path, what it does, what it takes in, what it
returns.
2. For each endpoint, who is allowed to call it (anyone / logged-in user / specific role).
3. The error cases each endpoint can return and what the frontend should do in each case.
4. The shape of any common errors (validation, not found, unauthorized).

Then call out: which endpoints need pagination, which need filtering, which need sorting. Mark anything
that's likely to be slow at scale.

Output is a plain-English API spec. No code yet.
```

IF YOU GOT STUCK HERE. Phases 2 and 3 are where most builds quietly go off the rails — because they feel boring and the AI is happy to skip them if you let it. If you're stuck and the AI keeps pulling you toward "let's just start coding," that's the signal you're at the load-bearing decision point. Get unstuck before you go further; everything downstream depends on it. Agent Logic runs paid sessions where I sit with you for a couple hours and we get the data model and API contract written together. Sometimes that's all someone needs to unstick the build.

PHASE 4 — BACKEND SCAFFOLD

```
You are now writing code. We have:

Spec: [PASTE]
Data model: [PASTE]
API contract: [PASTE]

Scaffold the backend ONLY. Pick a stack appropriate for a solo non-engineer maintaining this — explain
why you chose it in 2–3 sentences before starting. Then:

1. Set up the project structure
2. Implement the database schema (migrations)
3. Implement the data models / ORM layer
4. Implement the API endpoints from the contract — but make every endpoint return real data from the
database, not mocked data
5. Add the simplest possible test that hits each endpoint and verifies the response shape

Do NOT build the frontend yet. Do NOT add auth yet (we'll do that in a later step — for now, treat all
requests as a single hardcoded test user).

Pause after each major step and confirm I've reviewed before continuing.
```

PHASE 5 — FRONTEND, ONE PAGE AT A TIME

```
You are now building the frontend, one page at a time. We have a working backend at [URL] with the API
endpoints described in [PASTE API CONTRACT].

For this turn, build EXACTLY ONE page: [PAGE NAME].

This page should:
- Display [WHAT IT DISPLAYS]
- Let the user [WHAT THEY CAN DO]
- Talk to these specific API endpoints: [LIST]

Build the simplest possible version that works end-to-end first. Plain styling is fine. Do NOT add:
animations, fancy UI components, multiple themes, advanced state management. Those come later.

After the page is done, write down — in plain English — what state this page manages, where that state
lives, and how the page handles each of: loading, success, empty list, error.
```

Run Phase 5 once per page. Don't ask the AI to build all five pages in one turn — that's the "build the whole app at once" mistake at smaller scale.

PHASE 6 — AUTH

```
We are adding authentication to the app. The backend currently treats every request as a hardcoded test
user. We need real auth.

Before writing any code, answer these in plain English:

1. Who logs in? (Customers? Employees? Both? Just me?)
2. How do they log in? (Email + password? Magic link? Single sign-on with Google? Phone number?)
3. What can each type of user see and do? (Walk through the API contract endpoint by endpoint and label
which user types can call each.)
4. What happens when a session expires? When a password is forgotten? When an account is locked?

Once those are answered, propose the simplest auth approach that fits a solo non-engineer maintaining
this app. Then implement it as a separate, reviewable layer — do not scatter auth logic through the
existing endpoints. Add a middleware that enforces it consistently.
```

PHASE 7 — DEPLOY

```
The app works locally. Now we need to deploy it so other people can use it.

Before writing any code or running any commands, walk me through these decisions in plain English:

1. Where will the backend run? (Why this option vs alternatives, for someone who is not a DevOps
engineer?)
2. Where will the database live?
3. Where will the frontend be served from?
4. How will the frontend find the backend (CORS, environment variables, custom domain)?
5. How will I update the app once it's deployed (manual deploy? auto-deploy on git push?)
6. What does it cost per month at zero users? At 100 users?
7. What can break in production that doesn't break locally? (List the top 5.)

Once I've reviewed and confirmed those decisions, walk me through the deploy step-by-step. Pause for
confirmation between major steps. Verify the app actually works at its public URL before declaring done.
```

Red flags during the build

Six signals that you're heading toward pretty-plus-broken. Any one of them is reason to stop and re-orient. Three or more, and you should restart from a current phase rather than keep patching forward.

1. You can't explain your data model in two sentences

If you can't say "the app stores tenants, units, leases, and payments — each tenant has one current lease, each lease links to one unit," you don't have a data model. The AI is filling in for you, and it's filling in differently every time.

2. The AI is generating components before the API exists

If your AI is showing you React components or page layouts before you've finished phase 3, the build is starting in the wrong place. Pretty UI on top of an unspecified API is exactly how the failure mode forms. Push back.

3. You haven't tested anything end-to-end yet

End-to-end means: type something into a real form, save it, refresh the page, see it still there. If you've never done that with this app, you don't have an app. You have screenshots.

4. You're adding features faster than you can test them

The AI loves adding. It will add a calendar, a chart, a notification bell. If you can't keep up with verifying each addition actually works, the rate is wrong. Slow down. Adding features faster than you test them is how week 4 looks great and week 8 doesn't run.

5. You don't know where deployment will happen

If you're four weeks in and "where this will live" is still an open question, the build is going to need significant changes once you decide. File storage, sessions, background jobs, and environment variables all behave differently across deploy targets.

6. The AI keeps "fixing" by adding new code instead of changing what's there

This is the most diagnostic one. When something is broken and the AI's response is to add a new file, a new helper, a new wrapper — instead of editing the file that's actually wrong — the codebase is duplicating itself. By month two you'll have three functions named almost the same thing, two of them broken, and no clear "right" one. Stop. Ask the AI to identify which existing file the bug is in and fix it there.

> **Bonus red flag.** If you find yourself describing your app to a friend with the phrase "and once it's working" — meaning, the part you're showing isn't actually working yet — you're past month two. Restart from phase 4 with a clean plan. It will be faster than continuing.

Self-diagnostic: is your build going wrong?

Five questions. Answer yes or no, honestly. Score at the bottom.

#	Question	Yes / No
1	Can you describe what your app does for ONE user in ONE sentence?	____
2	Can you list the "things" your app stores (entities) and write them on an index card from memory?	____
3	Have you actually used the app end-to-end — submitted a real form, refreshed the page, and seen the data persist?	____
4	If you opened the project today, could you find the file that handles a given API endpoint in under 30 seconds?	____
5	Do you know where this app will run when it's done (laptop, server, managed service), and have you tested deploying it at least once?	____

Scoring

Yes count	What it means	What to do
5 / 5	Healthy build. Architecture exists outside the AI; you understand it.	Keep going. Polish phase is fine.
3–4 / 5	Course-correctable. The plan exists in your head but is incomplete.	Stop. Write down the answers to the missing questions before the next prompt. Re-run the relevant phase prompt with the new info.
0–2 / 5	Stop and re-architect. You have screenshots, not an app.	Restart from Phase 1. Genuinely. The time to back up is now — the gap only widens with more code.

> Restarting feels like wasted work. It isn't. The plan you build the second time is much faster — you already know what the app should do. The first attempt taught you that.

The honest limits

This brief solves a specific problem: the small-business operator who's drowning in vibe-coded files and pretty broken screens. It doesn't solve every problem. Where it falls short:

This is for small internal tools

The seven phases are designed for tenant portals, inventory trackers, customer dashboards, scheduling apps, light internal CRMs — the kind of tool a small-business operator builds for their own ops or a small known user base. Not for high-traffic consumer apps, not for anything with regulatory load (HIPAA, PCI-DSS, SOC 2), not for hard-real-time systems (trading, gaming, live media). Those need professional engineers and they need them from day one.

The seven prompts make you a competent client, not a developer

Following this brief doesn't make you a software engineer. It makes you a competent *client* of an AI software engineer. There's a ceiling. Beyond a certain complexity, the AI's suggestions stop being load-bearing and a human engineer's judgment starts to matter more. You'll know you've hit the ceiling when the AI's answers stop being decisive — "you could do X or Y" with no clear preference — and start contradicting earlier answers.

Some apps need a real engineer, period

If your app moves money, handles sensitive personal data at scale, integrates with external services more complex than "fetch one API," or has uptime requirements where downtime costs more than a salary — budget for review by a human engineer. AI assistance speeds them up; it doesn't replace the judgment.

This won't fix an unclear idea

If you can't answer "what does this app do for one user in one sentence," the AI can't either. No amount of architectural rigor will save a build whose purpose is fuzzy. Sit with the idea longer before opening the IDE. Not glamorous. Load-bearing.

The plan is a tool, not a contract

You will learn things in phase 4 that change phase 2. That's fine. The plan is something you maintain through the build, not something you finish before the build. The discipline is keeping the plan and the code in sync — not freezing the plan once it's written.

> **Trust signal:** if you read this section and thought "okay, this isn't the magic shortcut I wanted" — good. The shortcut is doing the boring planning before the fun coding. Anyone selling you faster than that is selling you the same broken outcome you came here to avoid.

What's next

If you're going to build this yourself with the seven prompts, here's a realistic timeline for an evenings-and-weekends pace, working with an AI assistant:

Phase	Realistic time	Where you'll feel stuck
1. Spec	1 evening	Writing it short enough.
2. Data model	1–2 evenings	The edge cases the AI surfaces.
3. API contract	1 evening	Permissions per endpoint.
4. Backend	1 weekend	The first time something doesn't work and the AI doesn't know why.
5. Frontend (per page)	1 evening per page	State management on the page that has both a list and a detail view.
6. Auth	1 weekend	Choosing a method. They all sound fine.
7. Deploy	1 evening — if it goes well	It often doesn't go well. Budget two.

Three tiers of getting this done

Tier	What it is	When to choose
DIY	You drive an AI assistant through the seven prompts in this brief.	You enjoy the build, want to learn, time isn't the bottleneck.
DWY *(Done With You)*	Workshops where I sit with you and we get through the planning phases together. Late 2026.	You can build, but you keep stalling on Phases 2–3 and want a person to unblock you.
DFY *(Done For You)*	Custom Agent Logic install. I build it on your hardware, set it up at your office, train an employee to run it, and stay on call when it breaks.	You don't want to build it. You want it working in your business in weeks, not months.

ABOUT ME. I'm Alex Jahn. I'm a working carpenter and house-flipper in Fond du Lac, Wisconsin. I've been building agent-based tooling for my own trade business since the beginning of 2026 — from day one of autonomous AI agents being practical inside real business workflows — and the seven-phase sequence in this brief is the one I actually use. I run Agent Logic, where I help other small-business operators build the same kind of stack for their own work. The unfair part of the pitch: no AI consultancy actually runs a real trade business. No carpenter actually runs a multi-agent stack. I'm the only person dumb enough to do both, which means I've already solved the problems you're about to hit.

Talk to me

If anything in this brief was useful, the rest of what I do probably is too. Three ways to reach me:

Email	alexanderjahn79@icloud.com
Text	920-539-8814 — my actual cell, real human, no bot
Schedule a call	920-679-6207 — this number rings my AI assistant. Tell it you want time with me; it'll check my calendar and book the slot.

What I'd want to know on the first call

- What's the app you're trying to build, in one sentence
- What you've tried so far, and roughly where it's stuck
- Whether you'd rather learn the build or have someone else do it
- Your rough budget and timeline — honest answers, not the nice ones

I'll tell you whether DIY, DWY, or DFY is the right shape for you, and I'll tell you honestly if it's none of them. Sometimes the answer is "you don't need an app, you need a spreadsheet." That's a real answer too, and it's free.

> **One last thing.** If you read this brief, applied the seven prompts, and got a working app on your own — tell me about it. I collect those stories. They're how I learn what's missing from the next version of this brief.

Your AI Built You Something Pretty (And It Doesn't Work)
v1.0 · May 2026 · Alex Jahn · Agent Logic