The AI Build Playbook

Build software, agents, and orchestrations in the right order — and get far better results

A field checklist from Rehm KI Consulting

Version 1.0 · First edition · July 2026

Most poor results from AI-assisted building don't come from a weak model. They come from doing the steps in the wrong order — jumping to code before the problem is framed, letting one AI mark its own work, testing on real users instead of before them.

This is the order that works. Follow it top to bottom. Each step lists **what** to do, **why** it matters, and **what "done well" looks like** so you know when to move on.

> **How to read this:** the main text works for anyone directing an AI build. Lines marked **[Builder]** add the technical detail for developers. Skip them if they're not for you.

Step 0 — Frame the real problem

What: Before anything else, brainstorm the idea down to a single, coherent idea document. Write down the goal, your intentions, and concrete examples of what **success** and **failure** look like.

Why it matters: The stated request is a starting point, not a specification. Most wasted builds solve the wrong problem well. An hour spent here saves days later.

Done well: One page you could hand to a stranger and they'd understand what you're building, for whom, and how you'll know it worked.

Start small: Pick the simplest useful version of the problem first. Get that one thing working really well, then add to it in the iterations later (Step 6). Don't try to build everything at once — a simple thing that works beats a complex thing that half-works.

Step 1 — Assemble the context pack

What: Gather into documents everything the AI should build from: the idea doc, goal, intentions, feature list, success and failure examples, your values and constraints, and coding standards. Produce a short design document from these.

Why it matters: AI is reliable when the material is in front of it and unreliable when it has to recall from memory. A good context pack is also your strongest defense against hallucinated output — you're grounding the work in facts you supplied.

Done well: A folder of clear docs. Nothing critical lives only in your head or in a chat history you'll lose.

> **[Builder]** Include interface expectations, data shapes, and non-negotiables (security posture, licences, target runtime). This becomes the source the spec is written against.

Step 2 — Write the spec

What: Have a strong model draft a written specification from the context pack. Then you read it, correct it, and approve it.

Why it matters: The spec is the contract. Errors caught here are cheap; the same errors caught after building are not.

Done well: A specification you've personally reviewed and agree with — not one you skimmed.

> **[Builder]** Use a top-tier reasoning model for this (see "Right model for the job"). Spec should be precise enough to remove ambiguity, not so rigid it forecloses good implementation choices.

Step 3 — Write a detailed build plan

What: From the spec, produce a detailed, step-by-step build plan — detailed enough that a smaller, cheaper model could execute it to completion.

Why it matters: A strong plan lets you delegate the heavy lifting to economical models and keeps the build modular instead of tangled.

Done well: Discrete, ordered tasks with clear interfaces between them. Someone (or something) less capable than the planner could follow it.

> **[Builder]** Design modular, with a communication bus between components by default unless there's a good reason not to. Define the seams where the system naturally separates.

Step 4 — Build with discipline

What: Execute the plan. Build anti-fragile and secure from the first line. Work test-first, in modules. Run tests continuously as it builds, and make every test pass locally before any push.

Why it matters: Security and resilience bolted on at the end never fit. Anti-fragile design means one component failing doesn't take the whole system down.

Done well: A modular build where tests are green locally, and a failure in one part degrades gracefully instead of collapsing everything.

> **[Builder]** TDD throughout. Anti-fragility is a design requirement, not a nice-to-have — isolate failure domains. No push on red.

Step 5 — Independent review & hardening

What: Have **two separate** AI sessions adversarially critique the work. They only critique — they do not fix. Their findings go back to the original builder to fix. Run penetration testing several times.

Why it matters: Don't let the bookkeeper audit their own books. A reviewer with no stake in the work catches what the author is blind to. The separation is the point.

Done well: Critiques are specific and acted on; the doer and the checker are never the same agent; pen tests have been repeated, not run once.

> **[Builder]** Fresh context per critic — no shared memory with the builder. Repeat the review/fix loop until critics stop finding material issues.

Non-negotiable: Run everything through a CI/CD pipeline, every time — automatically built and tested on every change. All tests pass, always; a red build is never merged and never shipped "to fix later." And penetration testing is an absolute must, not a step to skip when time is short — an unpentested system is an untrusted one.

Step 6 — Human testing, then iterate

What: The first human test happens when the system is fully up and running. From there, develop iteratively.

Why it matters: Testing a half-assembled system wastes everyone's attention on problems the build would have resolved anyway. Real feedback belongs on a real, running thing.

Done well: A working system in front of a human, and a short list of the next improvements — not a scramble to make it start.

Two rules that run through every step

Right model for the job. Give the hardest thinking — framing, spec, plan, critique — to the best model available. Give routine execution to the cheapest model that does the job well. Matching the tool to the task is both more reliable and cheaper.

> **[Builder]** A working tier, tuned over time (names and versions change): top tier for hard reasoning; capable mid-tier and economical models for routine execution. Re-benchmark periodically — this list dates quickly.

Separate the doer from the checker. The same conflict of interest that keeps accountants from auditing themselves applies to AI. When you decide how to split and assign work, ask four questions:

1. **Size** — how big is the task in tokens for one session?
2. **Independence** — can it be split without losing effectiveness?
3. **Separation of concerns** — does one agent doing both create a conflict of interest?
4. **Checkability** — if checking is much cheaper than doing, use several checking agents.

Before you ship — the 90-second check

> 1. Did I frame the real problem, or just the first one that came to mind?
> 2. Is the work grounded in a context pack, or in the model's memory?
> 3. Did an independent agent review it — not the one that built it?
> 4. Do all tests pass locally, and has it survived repeated pen testing?
> 5. Has a human used the fully running system?

If every answer is yes, you're in far better shape than most.

This is a general, overall guide to the build process — it stays at the level of principle and sequence, and can't cover the specifics of any one project. For detailed guidance tailored to your situation, get in touch.

About Rehm KI Consulting

We help businesses build with AI properly — with the engineering discipline that separates a reliable system from an impressive demo. Straight answers about where AI fits and where it doesn't. No hype.

Rehm KI Consulting · Christopher Rehm · Bavaria, Germany · contact@christopherrehm.de