Building Well with AI

A step-by-step guide to software, agents, and orchestrations — in the order that produces better results

Rehm KI Consulting

Version 1.0 · First edition · July 2026

Why order is the hidden variable

When an AI-assisted build goes wrong, the instinct is to blame the model — it wasn't smart enough, it hallucinated, it wrote sloppy code. That's usually not the real cause. The real cause is almost always **sequence**: the work was done in the wrong order. Code was written before the problem was framed. The model was asked to remember instead of being given the material. The thing that built the system was also the only thing that checked it. Real users were the first testers.

Get the order right and an ordinary model produces good work. Get it wrong and the best model in the world produces a confident mess.

This guide lays out that order. It applies to three kinds of work, which differ less than they appear:

- **Plain software** — a normal application or service.
- **AI agents** — systems where a model takes actions through tools, with some autonomy.
- **Orchestrations and routines** — multiple agents or steps composed into a pipeline, where the coordination is the product.

The steps are the same for all three. Where an agent or an orchestration needs something extra, the step says so.

Three ideas run underneath everything that follows, and they're worth stating plainly before the steps begin. **Frame before you build** — the stated request is rarely the real problem. **Ground, don't recall** — AI is trustworthy when the material is in front of it and unreliable when it reaches into memory. **Separate the doer from the checker** — nothing that made a thing should be the only judge of whether it's good. Almost every step below is one of these three ideas applied at a different moment.

Step 0 — Frame the real problem

Do this: Brainstorm the idea down to a single coherent idea document. In it, state the goal, your intentions, and — this is the part people skip — concrete examples of what **success** looks like and what **failure** looks like.

Why it matters: The stated request is a starting point, not a specification. People describe symptoms, not causes; they describe an assumed solution, not the underlying need. If you build exactly what was asked for, you'll often build the wrong thing beautifully. The failure examples matter as much as the success ones, because they draw the boundary the system must not cross.

What good looks like: One page a stranger could read and correctly explain what you're building, for whom, and how you'll both know it worked.

Start with the simplest useful version. Resist framing the whole ambitious system at once. Pick the smallest problem that delivers real value, get that working really well, and add capability in later iterations (Step 6). A simple thing that works beats a complex thing that half-works — and the simple version teaches you what the complex one actually needs.

Common failure: Vague success criteria ("make it good," "users should like it"). If success isn't concrete, you can't test for it and you can't tell when you're done.

- **For agents:** Frame what the agent is responsible for and, equally, what it must never do. "Success" and "failure" become specific behaviors — the actions you want and the actions that would be a disaster.
- **For orchestrations:** Frame the end-to-end outcome first, before decomposing. A pipeline of individually correct steps that don't add up to the goal is a classic and expensive mistake.

Step 1 — Assemble the context pack

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

Why it matters: This is the single biggest lever on quality and the strongest defense against hallucination. A language model predicts plausible text; when it lacks a fact it invents a plausible one, in the same confident voice it uses for things it knows. The fix is not to hope it remembers — it's to put the truth in front of it. A build grounded in a context pack you supplied is working from your reality, not the model's approximation of it.

What good looks like: A folder of clear documents. Nothing load-bearing exists only in your head or in a chat history you'll lose next week.

Common failure: Critical constraints (a licence requirement, a security rule, a data format) that were "obvious" and therefore never written down — so the model never knew them.

- **For agents:** The context pack becomes the agent's system prompt, its tool definitions, its guardrails, and a set of worked examples. Be explicit about tools available and boundaries fixed.

- **For orchestrations:** Define the shared context and the contracts between components — what each one can assume about the others. Ambiguity between agents is where orchestrations rot.

Step 2 — Write the spec

Do this: Have a strong reasoning model draft a written specification from the context pack. Then you read it, correct it, and approve it. The human sign-off is not optional.

Why it matters: The spec is the contract between intention and implementation. An error in the spec is cheap to fix now and expensive to fix once it's been built on. Reviewing the spec is the highest-leverage half hour in the whole project.

What good looks like: A specification you have personally read and genuinely agree with — precise enough to remove ambiguity, not so rigid that it dictates every implementation choice and forecloses better ones.

Common failure: Skimming the spec because it "looks right." A spec that looks right and is subtly wrong is worse than no spec, because everyone now trusts it.

- **For agents:** Specify the role, the inputs and outputs, the tools, and the stop conditions — when the agent should hand back, escalate, or refuse.
- **For orchestrations:** Specify each hand-off. The messages between steps are the real interface; spec them as carefully as you'd spec a public API.

Step 3 — Write a detailed build plan

Do this: From the approved spec, produce a detailed, ordered build plan — detailed enough that a smaller, cheaper model could execute it to completion without needing the reasoning that produced it.

Why it matters: A strong plan is what lets you delegate the bulk of the work to economical models and keep costs sane (see "Right model for the job"). It also forces modularity: to write the plan you must find the seams where the system naturally separates, and those seams become clean interfaces instead of tangled dependencies.

What good looks like: Discrete, ordered tasks with clear interfaces between them — each one either directly buildable or directly checkable.

Common failure: A plan that's really a monolith in disguise — steps so entangled that nothing can be built or tested in isolation.

- **For agents & orchestrations:** Define the message contracts, the routing (who calls whom, and when), and the failure handling before building. Decide up front what happens when a

step times out, returns garbage, or loops. Build modular, with a communication bus between components by default unless there's a specific reason not to.

Step 4 — Build with discipline

Do this: Execute the plan. Build anti-fragile and secure from the first line, not as a later pass. Work test-first, in modules. Run tests continuously as the system is built, and require every test to pass locally before any push.

Why it matters: Security and resilience retrofitted at the end never fit properly — the assumptions are already baked in. Anti-fragile means the system is designed so that when one part fails, the rest keeps working; failure is isolated instead of cascading. This is the difference between a bad afternoon and a catastrophe, and for anything running as a service it is not optional.

What good looks like: A modular build, tests green locally, and demonstrable graceful degradation — you can kill a component and watch the rest hold.

Common failure: "We'll add security and tests later." Later doesn't come, and when it does, it's a rewrite.

- **For agents:** Guardrails, input validation, sandboxed tool access, and sane timeouts and retries. Assume the model will sometimes produce bad output and design so that bad output can't do damage.
- **For orchestrations:** Isolate failure domains so one failing agent doesn't poison the pipeline. Make steps idempotent where you can, and add circuit breakers so a stuck or looping component gets stopped rather than left to run.

Step 5 — Independent review and hardening

Do this: Have **two separate** AI sessions adversarially critique the work. They only critique — they do not fix anything. Their findings go back to the original builder to implement. Then run penetration testing, several times, not once.

Why it matters: You don't let the bookkeeper audit their own books, for the same reason you shouldn't let the agent that wrote the code be the only one to judge it: the conflict of interest is structural, not a matter of good intentions. A reviewer with fresh context and no stake in the work sees what the author is constitutionally blind to. Keeping critique and fixing separate preserves that independence — the moment a critic starts fixing, it acquires a stake.

The non-negotiables. Three things here are not matters of judgment. First, everything runs through a CI/CD pipeline — every change, every time, automatically built and tested. Second, all tests pass, always; a red build is never merged and never shipped "to fix later." Third, penetration

testing is mandatory, not a luxury for when there's time — an unpentested system is an untrusted system.

What good looks like: Specific, actionable critiques that get implemented; a doer and a checker that are never the same agent; and penetration tests that have been repeated until they stop turning up material problems.

Common failure: A single review pass, or a "reviewer" that's really the same session wearing a different hat. Independence you didn't actually enforce is independence you don't have.

- **For agents & orchestrations:** Adversarially test the prompts and the flow — prompt injection, jailbreaks, malformed inputs, hostile tool responses. Red-team the whole orchestration, not just individual agents; the interesting failures live in the hand-offs.

Step 6 — Human testing, then iterate

Do this: Run the first human test when the system is fully up and running — not before. From there, develop iteratively on real feedback.

Why it matters: Testing a half-assembled system spends human attention on problems the build would have resolved on its own. Human judgment is scarce and valuable; point it at a real, running thing where the feedback is real.

What good looks like: A working system in front of a human, producing a short prioritized list of genuine improvements — not a scramble to get it to start.

Common failure: Using real users as the first integration test. They'll find that it's broken, not that it's imperfect, and you'll have burned trust to learn something a smoke test would have told you.

Right model for the job

Give the hardest thinking — framing, spec, planning, and critique — to the strongest model you have access to. Give routine execution to the cheapest model that does the job reliably. Using an expensive model for everything wastes money; using a cheap model for the hard reasoning wastes everything, because the errors propagate into every step downstream.

A practical tier, **stated honestly as a working set that dates quickly** — model names and versions change constantly, and any specific ranking should be re-checked before you rely on it:

- **Top tier — hardest reasoning:** framing, spec, build plan, adversarial critique.
- **Capable mid-tier and economical models — routine execution:** implementing well-specified tasks from a detailed plan.

The discipline isn't the specific list; it's the habit of matching capability to task difficulty and re-benchmarking periodically instead of trusting a ranking that's six months stale.

Separate the doer from the checker

This is the heart of good orchestration, so it gets its own treatment. The principle is the accountant's: whoever does the work should not be the only one who checks it. When you decide how to split and assign work across models or agents, run every task through four questions:

1. **Size** — how big is the task, in tokens, for a single session? Tasks that don't fit cleanly in one context window need splitting or a different structure.
2. **Independence** — can the task be split without losing effectiveness? Some things decompose cleanly; some lose their meaning when cut apart. Split along natural seams, not arbitrary ones.
3. **Separation of concerns** — would having one agent do both halves create a conflict of interest? If so, separate them, even at some cost in efficiency.
4. **Checkability** — is checking the work much cheaper than doing it? When it is, use several independent checking agents. Cheap verification run in parallel buys a lot of reliability for little money.

That last point is the quiet superpower of orchestration: verification is often far cheaper than generation, so you can afford to check the same output several ways and catch what any single check would miss.

The anti-patterns, in one place

- **Solving the stated problem instead of the real one.** Frame first.
- **Letting the model recall instead of giving it the material.** Ground everything.
- **Writing code before the spec and plan exist.** Sequence is the whole game.
- **The monolith in disguise** — a "plan" whose steps can't be built or tested in isolation.
- **Security and tests "later."** Later is a rewrite.
- **Self-review** — the builder grading its own work.
- **One pen test.** Repeat until it goes quiet.
- **Users as first testers.** Human testing starts on a fully running system.
- **One model for everything** — either overpaying for routine work or under-powering the hard reasoning.

The honest bottom line

None of this is exotic. Every step is obvious once stated. The difficulty is that under the pressure to just get building, the defaults reassert themselves — skip the framing, trust the model's

memory, let it check itself, ship to users to “see what happens.” Doing this well is mostly the discipline of following the order even when skipping ahead feels faster. It isn’t faster. It only feels that way until Step 6.

This is a general, overall guide to building well with AI. It deliberately 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

Rehm KI Consulting helps organizations build with AI the way it should be built — with real engineering discipline behind it, not a demo dressed up as a product. The approach is grounded in decades of hardware and software engineering, from chip design and network processors at Intel to years of building web applications, now focused on helping businesses adopt AI in a way that’s honest, reliable, and built to last.

The stance is simple: straight answers about where AI genuinely helps, and where it doesn’t. No hype, no magic.

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

Model names, tools, and version rankings mentioned in this guide change quickly — treat them as examples to re-verify, not fixed facts.