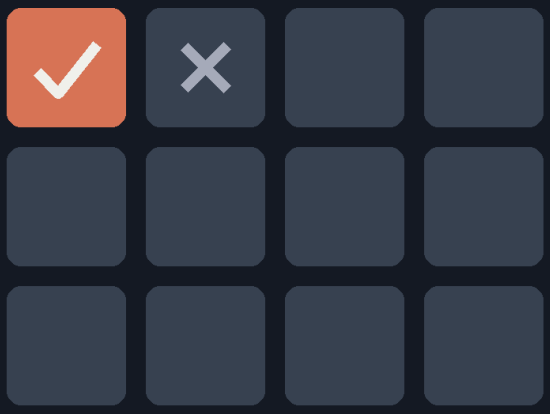


ARCHITECTING WITH CLAUDE

Best Practices

Dos and don'ts for starting and reviewing AI projects

Daniel Wanja
Nouvelles Solutions · November 2026 · Based on The Principles v1.1

Architecting with Claude: Best Practices

Nov 1, 2026 · @Daniel

Use this guide at the start of an AI project and again at each review. It turns the twelve principles from *Architecting with Claude: The Principles* into plain dos and don'ts.

How to use this guide

Work through the ten sections in order. They follow the life of a project, from the first idea to the day another team runs the system.

- **Starting a project:** read each section with the team and agree on the dos before anyone builds.
- **Reviewing a project:** look for the don'ts. Each one you find is a risk to fix or to accept on purpose.
- **Short on time:** use the checklist at the end. Every "no" is worth a conversation.

1. Frame: start from the outcome

Decide which number must move, for whom and by how much, before you pick a model.

✅ Do	❌ Don't
Write the goal in one sentence: "For [who], when [this arrives], produce [this] so [metric] goes from [today] to [target]."	Start with "we need an AI assistant" and look for a problem later.
Measure how the process works today, before you build.	Claim a gain you can't compare to a baseline.
Let code do rules, lookups and math. Let Claude do the messy reading and judgment.	Ask the model to do arithmetic or apply fixed rules that code can do exactly.
Pick one main goal (time, cost, quality). Treat the others as limits you must not break.	Chase five goals at once with no clear winner.
Count the full cost: model, retries, human review time and mistakes that slip through.	Judge cost by the price per token alone.

✅ Do	❌ Don't
Check or contain every output before it can trigger something you can't undo.	Let unchecked model output send money, delete data or email a customer.
Plan what happens when Claude is slow or down.	Assume the model is always available.

2. Shape: use the simplest design that works

Start small and add steps or autonomy only when tests show you need them. The ladder: one call, then a model with tools and data, then a workflow, then an agent.

✅ Do	❌ Don't
Try one well-written prompt first. Climb the ladder only with evidence.	Start with a multi-agent system because it sounds impressive.
Use a workflow when the steps are known: your code decides what happens next.	Use an agent for a fixed, predictable process.
Give every agent a step limit, a budget limit and a pause before anything irreversible.	Let an agent loop with no stop rule.
Make sure an agent can see real results, like tests that pass or data that comes back.	Let an agent judge its own work with nothing to check against.
Design tools so the common mistake is impossible, with clear names, examples and limits.	Wrap every API endpoint as a tool and hope the model picks the right one.
Split the work into clear subtasks with clear inputs and outputs before adding agents.	Add more agents when one agent with better instructions would do.
Plan the feedback step: how errors come back and get fixed.	Ship with no way to learn from mistakes.

3. Configure: pick the model, then spend context carefully

Choose the model with tests, then give it only the information it needs, with the parts that never change first.

✅ Do	❌ Don't
Pick the model that passes your tests on the hardest common cases.	Pick a model by reputation or habit.
Not sure? Start with the most capable model, then step down while tests still pass.	Switch models before trying a lower effort setting on the same model.
Compare cost per finished task, not price per token.	Assume the cheapest model is the cheapest system.
Give the model the smallest set of useful information. Load more only when needed.	Paste everything "just in case". Long context makes the model miss things.
Put stable content first (instructions, examples), and the question last, so it can be cached.	Put timestamps or user names at the top of the prompt. That breaks caching.
Keep prompts in version control, with tests, and log the version used for each request.	Edit production prompts by hand with no record.
Add an instruction only to fix a failure you actually saw.	Write a long rulebook of if-then cases up front.

4. Connect: give agents less, connect them carefully

Give each agent only the tools and access its job needs, and check permissions in code at every step.

✅ Do	❌ Don't
Remove tools an agent doesn't need before you add guards to them.	Give every agent every tool "to be safe".
Split tools that read, change and delete into separate tools.	Offer one tool that can both look up and delete.
Use an allowlist: name what is allowed.	Rely on a list of what is forbidden and hope it is complete.
Check the user's permissions in code at every step, not in the prompt.	Run the agent as a super-user and tell it "only show the user's own data".

✅ Do	❌ Don't
Treat web pages, emails and documents as untrusted. They can carry hidden instructions.	Let content the agent reads change what the agent is allowed to do.
Never let one agent hold private data, read untrusted content and send data out, all at once.	Combine all three in one agent. That is how data leaks.
Give each integration its own credentials.	Pass the user's login token to other systems.

5. Ground: answer from the right data

Send each question to the place where its answer lives, and make every answer show its source.

✅ Do	❌ Don't
Send text questions to search, totals to a database query, and live status to the system that owns it.	Use document search to answer "how many" or "what total".
If all your documents fit in the prompt, put them there and cache them.	Build a search index for a small set of documents that would fit.
Split documents on natural breaks (sections, clauses) and keep their titles and context.	Cut documents into fixed-size chunks that lose their meaning.
Add keyword search beside meaning-based search when users type codes, IDs or clause numbers.	Rely only on meaning-based search for exact codes.
Filter by the user's permissions before ranking results.	Retrieve everything and ask the model to hide what the user can't see.
Show the source with every answer, so a person can check it.	Accept answers with no evidence.
When answers go wrong, check what was retrieved first. Most failures start there.	Blame the model before checking the data and the index.

6. Prove: define "good" before you build

Write down what "good" means as measures and test cases before building, then change one thing at a time and measure it.

✅ Do	❌ Don't
For each goal, write a measure, a pass mark, a set of test cases and a way to grade them.	Say "it should be accurate and fast" and stop there.
Build a test set from real inputs, with expert answers. Include hard cases and tricky ones.	Test only on a few friendly examples or screenshots.
Include cases where the right answer is to ask, hand off to a person or say no.	Test only cases where the system should answer.
Grade with code where you can, a model judge checked against people where you can't.	Trust a model judge you never compared with human ratings.
Test serious mistakes on their own, separate from the average score.	Let a good average hide a dangerous error.
Change one thing at a time. Report cases fixed and cases broken.	Change the prompt, model and data together, then guess what helped.
Turn every real incident into a test case before you ship the fix.	Fix a bug and let it come back next release.
Roll out in steps: offline tests, then silent shadow runs, then a small group, then everyone.	Switch all users at once.

7. Operate: make cost and speed visible

Find where the time and money actually go, fix the biggest item first, and watch quality while you do.

✅ Do	❌ Don't
Break down the cost of one typical task and find the biggest item.	Optimize a line that is 5% of the bill while a 60% line grows.

✅ Do	❌ Don't
Set a time budget for each step, and decide if users care about the first words or the full answer.	Treat "fast" as one number for the whole system.
Reuse long, repeated prompt content with caching.	Pay full price for the same instructions on every call.
Send work nobody is waiting for to batch processing, at about half price.	Run overnight jobs at full real-time price.
Stream answers so people see progress sooner.	Expect streaming to cut the bill. It only cuts the wait people feel.
Keep one trace ID for each request across every step. Log structure always, content rarely, secrets never.	Log full prompts and personal data everywhere, or log nothing at all.
Watch quality next to cost and speed, split by route and version.	Celebrate lower costs while answers quietly get worse.

8. Govern: enforce, don't request

Put every rule that matters where the model can't ignore it, and match human review to the stakes.

✅ Do	❌ Don't
Ask of every rule: "If the model ignored this, what would still stop the harm?"	Trust a line in the prompt to enforce a hard limit.
Enforce hard limits in code, permissions and sandboxes. A refund limit belongs in the refund tool.	Write "never refund more than $500" in the prompt and call it done.
Low stakes: automate and spot-check. Medium: a person checks when the system is unsure. High or irreversible: a person approves first.	Use the same review for everything, or none at all.

✅ Do	❌ Don't
Make approvals real: show the exact action, lock it to what was approved, treat no answer as "no", and log it.	Show a vague "Approve?" that people click by habit.
Keep reviewers alert: send fewer, better prompts, and plant known errors to test them.	Assume reviewers catch everything. People approve most prompts by reflex.
Map each legal or policy rule to a control and an owner, and test it.	Treat compliance as a statement of good intentions.
Report results for each group of users, not just the average.	Let a 95% average hide 75% for one group.

9. Align: bring people along

Say the limits, trade-offs and failure rates out loud and early, so nobody who funds, approves, uses or runs the system is surprised.

✅ Do	❌ Don't
List everyone who can block go-live, and talk to each of them early.	Discover the security or legal reviewer a week before launch.
Read 50 real documents or requests before you design.	Design from a description of the data.
Show two to four real options, including one with no AI.	Present a single option as the only way.
Write down key decisions and why. Replace old decisions; don't rewrite them.	Rely on memory and meeting chat for why things are the way they are.
Turn "100% accurate", "instant" or "always up" into a measurable target plus a plan for the misses.	Promise perfection.
Show sponsors real failures before launch, and how each is caught.	Demo only the happy path.

✅ Do	❌ Don't
Hand over everything that shapes behavior: prompts, model version, test set, runbooks, known limits.	Hand over only the code repository.

10. Enable: make the team faster, safely

Set up the AI tools for the whole team. Instructions only inform; settings, permissions and hooks enforce.

✅ Do	❌ Don't
Keep shared instruction files (like CLAUDE.md) short and focused.	Grow a huge instruction file the model half follows.
Back every "never" with a real control: a deny rule, a hook or a sandbox.	Put a rule that would cause an incident if broken only in an instruction file.
Set team-wide policy centrally, so it applies on every machine.	Let each developer configure safety on their own.
Sandbox the shell so the tool can work with fewer approval prompts.	Train people to click "allow" all day.
Approve tool servers centrally, by address, not by name.	Let anyone connect any tool server.
Plan big changes first and approve the plan before edits. Keep a human checkpoint.	Let the assistant make large changes with no review.
Debug one layer at a time: replay the trace, find the layer, make the smallest fix, add a test.	Change several things at once and hope the problem goes away.

Project review checklist

Tick each item the project can show evidence for. Every unticked box is a risk to fix or to accept on purpose.

Frame

- [ ] The goal fits in one sentence, with today's number and a target.
- [ ] Nothing the model outputs can trigger an irreversible action unchecked.

Shape and configure

- [ ] The design is the simplest that meets the target, and we can say why we climbed past a single call.
- [ ] Every agent has a step limit, a budget and a pause before irreversible actions.
- [ ] The model was chosen by tests, and prompts are in version control.

Connect and ground

- [ ] Each agent has only the tools it needs, and permissions are checked in code.
- [ ] No agent holds private data, untrusted content and a way to send data out, all at once.
- [ ] Each question type goes to where its answer lives, and answers show their source.

Prove and operate

- [ ] "Good" is written down as measures, pass marks, test cases and graders.
- [ ] Changes are tested one at a time, and incidents become test cases.
- [ ] We know the biggest cost item and track quality next to cost and speed.

Govern, align and enable

- [ ] Every rule that matters is enforced outside the prompt.
- [ ] Human review matches the stakes, and approvals show the exact action.
- [ ] Everyone who can block go-live has been heard, and decisions are written down.
- [ ] The handover includes prompts, model version, test set and runbooks.
- [ ] Team settings and permissions are set centrally, with a human checkpoint on big changes.

Based on *Architecting with Claude: The Principles* by Daniel Wanja, version 1.1 (n-so.com). For details and current limits, see Anthropic's documentation. Independent work, not affiliated with Anthropic.