BEST PRACTICES

Ask

Best practices

Recommended workflows and usage patterns

AI agents are changing the way we build software, but they don't replace developers — they amplify them. AI agents aren't magic or mind readers. Just like your human co-workers, you need the right mental model and collaboration practices.

Engineering manager vs. tech lead

It's tempting to think of AI agents as a remote engineering team: you delegate tasks, they ship code, and you review the output. While that may be where we're headed, today's agents aren't there yet — and there are structural reasons they may never fully reach that model:

- Agents don't share your context or product intuition unless you give it to them.
- They lack the kind of deep, implicit knowledge transfer humans get through hallway conversations, design docs, and team culture.
- Even in human teams, you wouldn't expect one Slack message or one GitHub issue to capture all the nuance required to build something right the first time.

Bottom line: Expecting "set-and-forget" delegation leads to frustration, rework, and wasted effort.

Instead, think of today's AI agents like capable (but spiky) engineers on your team. You're the tech lead, and your job is to set direction and maintain coherence:

- Clarify product requirements
- Design system architecture
- Break work into achievable, agent-friendly tasks
- Review output and keep standards high
- Pair with agents on hard problems
- Lead post-mortems to capture lessons learned and improve team process

Clarify product requirements and design system architecture

Product requirements

You wouldn't expect a human developer to build a complex system from a two-sentence Slack message. Agents are no different.

- Be explicit. Agents can't read your mind. If a requirement isn't written down, assume it doesn't exist.
- Iterate interactively. It's fine to have the agent propose a first draft of requirements — but you own the final decisions.
- Document the vision. Create a high-quality README that explains:
  - The problem you're solving
  - The intended users and workflows
  - Key usage examples and expected outcomes
- Review specs before implementation. Treat specs like contracts between you and the agent. Catch ambiguities before they turn into bugs.

Just like with today's software development: expect this part to be hard. Requirements take time to get right. As with any software project, requirements will change as you discover constraints through building. Agents are fast at iterating — lean into that.

System design

Agents are surprisingly good at writing isolated pieces of code, but maintaining a coherent architecture across a growing codebase is still hard for them.

Why it matters

- Without guidance, agents often introduce inconsistent patterns, duplicate logic, or conflicting abstractions.
- If you don't understand the system's architecture, you won't be able to debug when the agent gets stuck.
- One of the most common failure cases we see: an agent produces a maze of ad-hoc code, the user loses track of how it works, and the only fix is to start over.

Your role as the architect

- Own the high-level design. Create a project-architecture.md that outlines:
  - Major entities and their responsibilities
  - Folder and file structure
  - Chosen tech stack and frameworks
  - Key design patterns and conventions
- Keep it simple. Fewer moving parts mean fewer ways for an agent to get lost.
- Stay in the loop. Even if agents get better at proposing architectures, you need to remain the source of truth for system design.
- If you can't explain your system's structure to a teammate in five minutes, your agent can't reliably extend it either.

Expect evolution. Your architecture will change — and that's okay. Just make sure the changes are intentional and documented.

Break work into achievable, agent-friendly tasks

It’s not just the final design that matters — how you get there determines speed, quality, and your ability to course-correct.

Agents tend to approach features in a linear, compartmentalized way. If a feature requires changes across files A, B, and C, they’ll often:

1. Fully implement A
2. Fully implement B
3. Fully implement C
4. Finally integrate everything together

This approach makes sense when you're reading the todo list up front, but in practice it creates headaches:

- The hardest bugs (integration issues) are discovered at the very end.
- You hit these problems when you have the most code, making them harder to diagnose.
- Fixes require large, risky refactors instead of small, targeted adjustments.

A better approach: iterative, integrated development

Set your agent up for success by driving the process in small, testable increments:

- Break features into vertical slices. Each slice should deliver something functional end-to-end, even if minimal.
- Integrate early and often. Catch integration issues while the surface area is small.
- Review smaller diffs. It’s far easier to spot mistakes when reviewing 50 lines of code than 500.

- Test as you go. Verify each slice works before moving on — you'll save time later.

Git is still your friend

Standard git hygiene already mirrors this planning approach. If you'd ask a coworker to split a large PR into smaller ones, ask your agent to do the same!

Create a git-usage.md file that explains your preferred workflow:

- Use feature branches for meaningful units of work, not one massive "feature-dump" branch.
- Keep commits small and descriptive — this helps both you and the agent reason about history.
- When you get stuck, rolling back to a clean commit is faster and safer than untangling a mess of untested changes.

Review output and keep standards high

A TL may not have written every line of code in their project, but they're still responsible for it. Spec and code reviews are still a vital part of agentic development. But if you focus your reviews on what matters most, your review load will shrink over time.

Agents are weakest on a blank page and strongest when extending an existing, working structure. Your review process should reflect that.

- New files and features: Review closely — architecture, naming, patterns, and correctness.
- Reuse: Be vigilant for duplicate code and make sure the agent is reusing existing modules where possible. Less code means less review!
- Style and conventions: Don't expect agents to guess your team's quirks or preferences. If you leave the same PR comment twice, make a coding-guidelines.md and write it down.

Pair with agents on hard problems

The best tech leads know when to delegate and when to dive in. Two ways to pair effectively

- Step-by-step guidance: For tricky refactors or complex bug hunts, walk the agent through one step at a time.
- Write it yourself when needed: Sometimes the fastest path forward is to take control of the keyboard for a few minutes, especially for:
  - Delicate API integrations
  - Multi-file migrations
  - One-off optimizations

It's tempting to aim for "never touch the code again," but stepping in at the right moments keeps your agent productive and unblocked. Tessl makes this seamless:

- Open file formats make it trivial to switch between agent-driven work and direct edits.
- The `document` tool can sync any changes you make back into the spec automatically.

Remember, updating specs or code manually isn't a failure mode — it's part of the process.

Lead post-mortems to capture lessons learned and improve team process

When the agent gets stuck or takes a wrong turn, it's frustrating — and once you fix it, it's tempting to move on. But skipping the retrospective means repeating mistakes.

Make time to analyze what went wrong

As the tech lead, your job is to understand why the agent struggled:

- Was context missing?
- Were requirements ambiguous?
- Did the problem need to be broken down differently?

If you're using Git (you should be!):

1. Stash your working changes — keep your progress safe.
2. Clear the agent's context window — avoid "cheating" with the new knowledge.
3. Try adding context to any of the files we've mentioned so far:
    a. The spec for the file
    b. README.md
    c. project-architecture.md
    d. coding-guidelines.md
    e. git-usage.md
4. Run it a few times to get a feel for what context helps. Try a few different problem breakdowns!
    a. Update your specs and guidelines with what you've learned.
    b. If you find a better way to structure problems, add it to AGENTS.md so future tasks can follow the improved approach.

Treat failed runs as data for better prompts, stronger specs, and smarter processes so you'll

Last updated 2 months ago