Engineering Leadership

Technical Guide

Prepared by

Kadharmoideen Fadurudeen

In this guide

2 articles · ~27 min total reading

- How I Structure Cross-Functional Teams for AI Project Delivery
- From Senior Engineer to Tech Lead: What Actually Changes

Engineering Leadership February 5, 2026 13 min read

How I Structure Cross-Functional Teams for AI Project Delivery

Shipping AI features needs a different team shape than shipping CRUD apps. How I structure product, ML, and engineering roles so AI projects actually reach production.

Every AI project I've watched fail didn't fail because the model was bad. It failed because the team around the model was shaped for a different kind of work. Someone owned "build the feature," nobody owned "decide when the feature is wrong," and by the time that gap surfaced, it was in front of a customer instead of in a review meeting. After leading enough of these projects to production, I've settled on a team structure that looks different from a standard product engineering squad — and the difference is the whole point.

Why the Standard Team Shape Doesn't Map

A normal feature team works because the definition of "done" is mostly binary. The checkout flow either processes the payment or it doesn't. You write tests, you catch regressions, you ship. AI features don't behave like that. A model that's 94% accurate isn't "almost done" — it's a system that will confidently produce a wrong answer for one out of every seventeen requests, forever, unless something changes that. The team has to be built around that reality from day one, not discover it in a postmortem.

The pattern I see most often: a product manager writes a spec that describes the happy path, a couple of engineers wire up a model or an API, it works beautifully in the demo, and then it ships. Three weeks later, support tickets start piling up with edge cases nobody tested because nobody was assigned to look for them. The team wasn't badly staffed — it was staffed for the wrong problem. Demos are a solved problem. Production trust is the actual project.

> The single biggest shift: a normal team is organized around building the feature. An AI team has to be organized around deciding whether the feature is good enough — and that requires roles that a typical sprint plan doesn't include.

The Roles That Actually Need to Exist

I don't mean four new hires. On most of my projects, two or three people cover these roles between them. But if nobody is explicitly responsible for each of the following, the responsibility doesn't disappear — it just becomes implicit, and implicit ownership is how quality problems slip through.

1. A product owner who defines tolerable failure, not just success

Most product specs describe what the feature should do when it works. For AI features, I need the product owner to also write down what happens when it's wrong — because it will be, regularly. Is a wrong answer embarrassing but harmless, like a bad restaurant recommendation? Or is it costly, like a wrong number in a financial summary? Those two cases need completely different amounts of guardrail engineering, and if the product owner doesn't decide this up front, the engineering team ends up guessing, usually by over-building safety for a low-stakes feature or under-building it for a high-stakes one.

2. Someone who owns the model or prompt behavior

This person's job isn't "write the prompt" — it's understanding why the system behaves the way it does across a wide range of inputs, and being the one who changes it deliberately rather than by trial and error. On smaller teams this is often a senior engineer who's spent enough time with the failure log to develop real intuition for the model's blind spots. It's a specific skill, closer to debugging a probabilistic system than writing deterministic code, and it needs to be someone's named responsibility, not a shared task that everyone touches occasionally.

3. A platform engineer responsible for evals, guardrails, and observability

This is the role most teams skip, and it's the one I refuse to skip anymore. Someone has to build and maintain the harness that measures whether changes make the system better or worse, the logging that captures what the system actually did in production (not just what it was supposed to do), and the guardrails that catch the failure modes the product owner flagged as unacceptable. Without this role, every improvement is a guess, and every regression is invisible until a customer reports it.

4. A human-in-the-loop reviewer, at least early on

For the first weeks after any AI feature ships, I want a real person looking at a sample of real outputs every single day. Not a dashboard of aggregate metrics — actual transcripts, actual edge cases. This is usually a rotating responsibility rather than a dedicated hire, but someone has to be assigned to it, or it quietly stops happening the moment the team gets busy with the next thing.

Sequencing the Work

The order matters as much as the roles. My default sequence:

- **Prototype with one or two people, not a full team.** The goal at this stage is learning what the model can and can't do reliably, not building production infrastructure. Committing a five-person team before you understand the failure surface is how budgets get burned on the wrong architecture.
- **Build the evaluation harness before you scale the team.** If you can't measure whether a change made the system better, adding more engineers just means more people making changes you can't verify. I've delayed hiring onto AI projects specifically because the eval harness wasn't ready — it felt slow at the time and saved months later.
- **Only then bring in the platform and guardrail work at full strength.** Once you know what "better" looks like and what failure modes actually show up, the observability and safety work has a real target instead of being built against guesses.

The Rituals That Actually Matter

Most of the standard agile ceremonies still apply, but I add two that I consider non-negotiable for AI work:

Weekly failure review

A standing meeting where the team looks at real cases the system got wrong that week — not a metrics dashboard, actual transcripts. Aggregate accuracy numbers hide the specific, fixable patterns that live in the raw cases.

A living edge-case log

A shared, searchable record of every unusual input the system has handled badly, tagged with whether it's been fixed, mitigated, or accepted as a known limitation. New team members read it before their first week is over.

Mistakes I've Made and Seen

- Treating an AI feature like a normal sprint-planned deliverable with a fixed scope and a ship date set before anyone had touched real failure data.
- Leaving evaluation criteria undefined and unowned, so "good enough" ended up being decided by whoever argued most confidently in the launch meeting.
- Scaling the engineering team before the eval harness existed, which multiplied the number of unverifiable changes going into the system rather than the amount of real progress.
- Assuming the human-in-the-loop review would "naturally" keep happening once the team got busy. It doesn't, unless it's someone's explicit job for a defined period.

What's Still Hard

I don't have a clean answer for how long the human-in-the-loop period should last, or exactly when a team can safely stop reviewing raw transcripts and trust the metrics. It varies by how costly a wrong answer is,

and I've been wrong in both directions — pulling review too early on one project, keeping it going long after it stopped finding anything new on another. The team structure above gets you a system that fails safely and visibly. It doesn't remove the judgment calls; it just makes sure someone is explicitly responsible for making them.

Engineering Leadership · December 15, 2025 · 14 min read

From Senior Engineer to Tech Lead: What Actually Changes

The title changes overnight; the job doesn't. What actually shifts when you go from senior engineer to tech lead, and the habits that made the transition work for me.

The email went out on a Friday afternoon: I was now the tech lead for our platform team. I remember closing my laptop that day feeling like something big had just happened. Monday morning, I opened the same laptop, looked at the same backlog, and had absolutely no idea what I was supposed to do differently.

Nineteen years into this career, across a handful of companies and a lot of team shapes, I've now made that jump myself and watched a dozen other engineers make it. The pattern is remarkably consistent: the title changes overnight, but the job takes months to actually change — and if you don't notice it needs to, you end up doing your old job with a new title stapled on top, which helps no one.

The short version

Your technical judgment is still the reason you got the role. But your job is no longer to produce the most output — it's to make sure the team produces the most output. Those are different skills, and nobody hands you a manual for the second one.

What Doesn't Change

It's worth naming this first, because a lot of "leadership advice" implies you should suddenly stop being an engineer. You shouldn't, and you can't — not if you want to be any good at this.

You still need deep technical judgment. Arguably you need it more, because now your calls affect five or ten people's work instead of just your own. You still need to

understand the codebase, not at the level of "I could find my way around it" but at the level of "I can tell when a proposed approach is going to cause pain in eight months." That understanding doesn't come from standups and status reports — it comes from staying close enough to the code that you can still read a diff and have an opinion worth listening to.

I've seen tech leads drift away from the codebase within a few months of the promotion, and it's almost always a mistake. Not because they need to be the best coder on the team — they don't, and pretending otherwise creates its own problems — but because losing the thread of what the codebase actually looks like erodes the exact judgment that got them the role in the first place.

🕓 What Actually Changes

Your time allocation flips

As a senior engineer, a good week meant I closed out a hard ticket, reviewed a couple of PRs, and maybe unblocked a teammate once or twice. As a tech lead, a good week might mean I didn't write a single line of production code — but three people on my team shipped work they'd have been stuck on without a conversation I had with them on Tuesday. That trade is uncomfortable at first. It stops being uncomfortable once you stop measuring your day by lines changed.

The success metric moves

This is the single biggest mental shift, and it's the one that takes longest to actually internalize rather than just intellectually agree with. As a senior engineer, "what did I ship this sprint" is a perfectly good question to ask yourself. As a tech lead, that question stops being useful. The question becomes "what did the team ship, and were the right things blocked or unblocked at the right time."

The uncomfortable part is that your personal output can go down while you're doing a genuinely better job than before. That's a hard trade to make peace with if you've spent a decade being rewarded for individual throughput.

Decisions now have a people dimension

As a senior engineer, a technical decision is mostly a technical decision: which library, which pattern, which trade-off between performance and simplicity. As a tech lead, the same decision now has a sequencing question attached — who's blocked on this, who should do it, does it fit the skill level of the person likely to pick it up, and what happens to morale if two people want the same piece of work. You're not just deciding what's technically correct anymore; you're deciding what's correct given the five humans who have to execute it.

The Habits That Made the Transition Work

Real 1:1s, with an actual agenda

I used to think 1:1s were a formality — a status check I could get from a Slack message. They're not. A good 1:1 is where you find out someone's been quietly stuck for three days, or that they're bored and about to start job hunting, or that they disagree with a technical direction but didn't want to raise it in front of the group. None of that surfaces in standup. I keep a running doc per person with open threads from the last conversation, so the 1:1 starts with "last time you mentioned X, how did that go" instead of an awkward silence.

A 1:1 agenda that actually works

- Follow up on whatever was open last time — don't let threads silently disappear
- Ask what's actually blocking them right now, not what's on the sprint board
- Leave room for career conversation, even if it feels premature — it's never premature to them
- Ask directly whether anything I did last week made their job harder

Write decisions down

As an individual contributor, I could hold most context in my head. As a tech lead, if a decision only exists in a conversation two people had, it effectively didn't happen — it

won't survive the next person joining the team, the next re-org, or even just three weeks passing. I started writing short decision records for anything that would be expensive to re-litigate: why we picked an approach, what we considered instead, and what would make us revisit it. It's saved us from the same architecture debate resurfacing from scratch more times than I can count.

Absorb interruptions so the team doesn't have to

One of the most concrete things a tech lead can do is stand between the team and the noise — the urgent Slack pings, the "can someone jump on a call" requests, the cross-team asks that arrive with no warning. Every one of those you take yourself is one less context-switch for someone who's trying to hold a hard problem in their head. This is invisible work. Nobody thanks you for the interruption that didn't happen. Do it anyway.

Delegate ownership, not just tasks

The lazy version of delegation is handing off the boring tickets while keeping the interesting architecture work for yourself. That's not delegation, that's just task distribution, and your best people will notice and resent it. Real delegation means giving away a piece of ownership you'd actually enjoy keeping — letting someone else own the design of a system, make the calls, and get the credit, even when you're confident you'd do it faster yourself. It is slower in the short term. It's the only way the team's capability grows instead of just your own.

Keep shipping something, on purpose

I make a point of taking on one small, well-scoped piece of hands-on work every few weeks — not because the team needs me to, but because it keeps my technical opinions grounded in what the codebase currently feels like to work in, not what it felt like a year ago. It also means code review conversations come from someone who's recently felt the same friction, not someone lecturing from a distance.

⚠ Traps I Watched Myself (and Others) Fall Into

- **Becoming the bottleneck:** insisting on reviewing every PR yourself feels responsible; in practice it just means everyone waits on you, and nobody else builds the judgment to review well.

- **Avoiding the hard conversation:** when someone's underperforming, it is always easier to quietly route work around them than to name the problem directly. It's also unfair to everyone else absorbing the slack.
- **Solving the technical problem and ignoring the people problem:** a recurring incident is often a symptom of an unclear owner, not a missing retry policy. Fix the retry policy and the ownership gap either way, or it'll surface again in a different shape.
- **Optimizing for being liked over being useful:** a tech lead who never pushes back on a bad plan to avoid conflict isn't being kind, they're outsourcing the consequences to the team six months later.

The First 90 Days, Concretely

A checklist for the first 90 days as a new tech lead

- **Weeks 1-2:** set up recurring 1:1s with every direct report and stakeholder you depend on — don't wait for a reason to talk to them.
- **Weeks 1-2:** ask each teammate directly what's currently slowing them down — you'll hear the same answer from more than one person, and that's your first real signal.
- **Weeks 3-4:** resist the urge to change anything structural yet. Understand why things are the way they are before you decide they're wrong.
- **Month 2:** pick one visible, low-risk improvement and delegate it fully to someone who wouldn't normally get the opportunity.
- **Month 2:** start writing decision records for anything you find yourself explaining verbally more than once.
- **Month 3:** have the conversation you've been putting off — the underperformance issue, the unclear ownership, the process nobody likes but nobody's changed.
- **Month 3:** honestly assess your own coding time. If it's zero, find a small way to get hands-on again before you lose the thread entirely.

♡ Closing Thought

I'd love to tell you this becomes a solved problem after the first year. It doesn't. Every new team, every new mix of personalities, every new stage of company growth resets part of the equation. What changes is that you get faster at noticing when you've slipped back into old habits — reviewing everything yourself, avoiding a hard conversation, measuring your week by what you personally shipped. The job isn't to arrive at a fixed way of leading. It's to keep noticing, and keep adjusting, faster each time.