AI LEADERSHIP AUDIT

6 Patterns Your Dashboard Can’t See

JONO HERRINGTON

Your engineering org adopted AI. Your leadership didn’t.

The tools are installed. The metrics are moving. And somewhere between the adoption reports and the actual engineering floor, a gap opened that most leadership teams haven’t noticed yet. Your engineers feel it. Your tech leads talk about it in private. Your best people are making decisions about whether to stay based on it.

This document covers six patterns that show up in every engineering org navigating AI adoption. They don’t show up in your sprint metrics or your adoption reports. They show up in the conversations your team is having without you.

Six patterns. Each one ends with a diagnostic question and one thing you can do this week. The only question is whether you’re willing to hear what your team has already concluded.

01

DID YOUR TEAM ADOPT AI, OR WERE THEY TOLD TO?

I spent time in a thread with over 500 experienced engineers talking about how AI adoption was handled at their companies. People who've been writing code since childhood. Senior engineers, staff engineers, tech leads, engineering managers. The frustration was real and specific. Not about the tools. About how the tools showed up.

The pattern was consistent across almost every comment I read.

Leadership sends an email.

All teams will integrate AI tools by end of quarter. Approved vendors attached. Adoption metrics to follow.

No conversation about which types of work actually benefit from AI and which don't. No pilot where a team tries it on real work and reports back what they found. No feedback loop where engineers can say "this helps here but hurts here." Just a mandate and a number to hit.

What happens next looks fine for months. Adoption metrics climb. Velocity holds or even rises. The quarterly review slide looks clean. And then, quietly, something starts to degrade. Decisions that used to come from judgment start coming from autocomplete. The people who were exceptional at the hard parts start to feel like the hard parts don't matter anymore. Engineers who never fully built a system from scratch can't debug it when it breaks, because they never developed the mental model for how it works.

The failure curve is slow and invisible until it isn't. By the time it shows up in production, leadership has moved on to the next initiative.

Here's what mandates miss. They treat every engineering task the same, and that's where the damage starts. AI is exceptional for boilerplate. Testing scaffolds. Exploring an unfamiliar API. Generating the scaffolding for something you already know how to build but don't want to type for three hours. That's real leverage. Real time returned to the things that require your full brain.

The picture changes on nuanced system design. Security-critical paths. Code that needs to survive five years of edge cases from customers you haven't met yet. When you mandate usage without making those distinctions explicit, engineers stop making them too. The boilerplate and the critical path start to blur. And the engineer who was great at knowing the difference … the tech lead who would have flagged it in review, the staff engineer who would have pushed back in planning … stops being asked to make the call.

This is because adults do the exact same thing our kids do when you mandate something. They push back. Sometimes loudly, sometimes quietly. We laugh about it when we're talking about our kids. Nobody admits it plays out the same way in a Slack channel. Put someone in a corner, tell them this is how they work now, measure whether they're complying … you haven't driven adoption. You've driven compliance. And compliance means engineers will use the tool on the tasks that get measured and quietly stop applying full judgment everywhere else.

When I introduced AI to my team at Converse, I didn't send an email. I gave them a sprint. Two weeks of blocked time specifically for exploration. Not meetings, not deliverables … just room to try things without a deadline breathing down their necks.

But before I gave my team anything, I had to give myself time with it. The first time I opened Cursor, I used it for ten minutes and shut it down. It felt foreign. I recognized the feeling. Fear of change dressed up as productivity skepticism. I'd seen it in other people. I hadn't expected to see it in myself.

It took me about a month to come back to it. My tech lead had been using Cursor and kept pushing me to give it another shot. He was already deep in it. I eventually came back, and once I did, I stayed. Built workflows. Trained agents. Built pipeline tools. I was one step ahead of my team. That's all a leader needs to be. Not a hundred. Just one.

We were all in learning mode together. There was no expert because it was new to all of us and the industry at large. My tech lead started a weekly call where the team shared what they were finding ... what worked, what didn't, what surprised them. I wasn't even in all of those sessions. I didn't need to be. The point was that curiosity was driving it, not a dashboard. We had real conversations about where AI creates leverage and where it doesn't. I showed them how AI changed my own workflow. Not a demo. Not a slide deck. An actual walkthrough of something I was working on ... what I tried, what surprised me, what I still don't trust it with. I made space for them to bring discoveries back. I let it be visible when I was learning from what they found.

The frame wasn't "we need to adopt this." It was "let's go see what this thing can do."

When you're measuring people, they optimize for the metric. When you're curious alongside them, they start optimizing for the thing itself.

My team uses AI constantly now. They also love their work. Those two things aren't in conflict. But they would be if I'd sent the email and called it a strategy. The difference between a team running on curiosity and one running on compliance is the leader's posture. The same tool produces completely different outcomes depending on who showed up to lead the rollout.

The Question

How did your team adopt AI, and who decided how it would be used?

This Week

Ask three engineers on your team, individually, what they wish had been different about how AI tools were introduced. Don't defend the rollout. Just listen. Write down what you hear.

02

WHEN DID YOUR LEADERS STOP TOUCHING THE SYSTEM?

AI-generated code is shipping into your codebase at a volume that didn't exist eighteen months ago. Thousands of lines per PR in undisciplined teams. Patterns your team has never used before showing up in production. Architectural decisions being made implicitly by a model that has no context on your system, your customers, or your technical debt.

Your senior engineers and tech leads are approving this work. The question is whether they can actually evaluate it.

The gap between what leadership sees and what the codebase actually contains used to widen slowly. Months. Quarters. A leader could drift from the technical work for six months and still have enough residual context to ask the right questions in a review. The surface area of what they needed to understand moved at a pace that was possible, if uncomfortable, to keep up with.

AI changed the math. The blast radius expanded overnight. The volume of code entering the system multiplied. The patterns shifted. The failure modes shifted with them. And the leaders who were already drifting now have almost no chance of catching up, because what they'd need to understand grew faster than their ability to re-engage with it.

A leader who stopped touching the system six months ago could bluff through a sprint review when engineers wrote 50 lines of code a day. They can't bluff when AI is generating 500.

The tells are different. The failure modes are different. The architectural patterns that look correct on the surface but introduce subtle downstream debt … those require a kind of technical intuition that erodes the moment you stop exercising it.

This affects every level of engineering leadership. A tech lead who stopped reviewing PRs three months ago is missing patterns that are calcifying into the codebase right now. A principal engineer who stopped prototyping is setting architectural direction based on assumptions they can no longer verify. A VP who stopped understanding how the system breaks is making commitments to the business based on timelines that may not be grounded in reality. An engineering manager who stopped pairing with their team can't tell whether a sprint estimate is solid or padded by 3x because the last three estimates got cut in half.

The drift is silent. One week you skip the PR reviews because you're in consecutive stakeholder meetings. The next week it's easier to skip them again. You don't decide to stop being technical. You just stop making time for it. A month later you couldn't read the code with confidence if you tried.

There's a harder truth underneath the operational one. The real reason most leaders stop being technical isn't "I don't have time." It's that staying technical in the age of AI-generated code means constantly confronting how much you don't know. The models are generating patterns your team may not have used before. The velocity of change in the codebase outpaces your ability to absorb it passively. Staying technical means being the person in the room who knows less than everyone else about the specific implementation … and being okay with that.

That's hard. It's much easier to retreat to strategy and stakeholder management, where nobody can prove you wrong with a stack trace.

I've retreated. More than once. Not consciously. The drift is always silent from the inside. So I pay attention to the signals. When my 1:1s start feeling like status reports instead of problem-solving sessions, that's a flag. When I catch myself nodding along to a technical explanation I don't fully understand, that's a flag. When I haven't been surprised by something in the codebase in weeks, that's the biggest flag of all … because it usually means I've stopped looking.

I still read pull requests. Not to approve them. To understand how my teams think about problems. When I see a PR that restructures an API layer, I'm not checking syntax. I'm asking whether this team understands the downstream impact. Whether the approach is consistent with how the other team on the other side of the world solved a similar problem last month. That's the context only a leader has. Not the code. The connections between the code.

The other week a bug came in. Instead of routing it to production support, I dove in. Checked the logs. Found a data issue. Fixed the customer experience in 30 minutes instead of letting it sit in a queue for days. The fix wasn't the point. The point was that I now understand a failure mode in our system that I wouldn't have known about otherwise. Next time my team proposes an architecture change that touches that area, I'll ask the right question. Not because I read it in a status report. Because I saw it break.

I know some leaders disagree with this. Some would say if you're in the code, you're not trusting your team. Others would say management has no business in the codebase whether AI is involved or not. But think about it like coaching. A head coach doesn't need to be a specialist at every position. But they need to understand the game well enough to read what's happening on the field. They need to watch the plays, study the tape, understand why something broke down. The moment a coach stops watching the field and starts coaching entirely from the press box, the team feels it. Engineering leadership works the same way. The altitude changes. The method doesn't.

Engineers are allergic to authority without competence. They've all worked for the person who stopped coding five years ago and now makes pronouncements about technical direction based on whatever they read on Hacker News that morning. They'll comply. They won't follow. And there's a canyon between those two things. Compliance means they do what you asked. Following means they trust your judgment enough to execute when you're not in the room. One scales. The other caps your impact at whatever you can personally oversee.

The moment I stop understanding how our systems break, I lose the ability to make good decisions about them. In a world where AI is changing how those systems are built every week, that moment arrives faster than it ever has before.

The Question

When was the last time your leadership team touched the system they're making decisions about?

This Week

Open the last five PRs that shipped on your team's primary codebase. Read them. Not to approve or critique. To understand what's being built and how. If you can't follow the code, that's the answer.

03

ARE YOU MEASURING PRODUCTIVITY OR PERFORMING IT?

Every engineering leader has seen the same headline by now. AI makes developers 3x more productive. For a lot of organizations, that number became the target the moment someone put it in a slide deck. The problem is that nobody measured what 1x looked like before they started chasing 3x.

When we first started evaluating AI's impact on our workflow, I sat with a group of principal engineers over coffee in a round table reviewing how their teams were tracking it. Dashboards everywhere. Velocity up. PRs merged faster. Features shipped ahead of schedule. Cycle time down. The data looked great across the board.

I asked one question. "What happened to unplanned work?"

They went quiet. Then someone said, "That's a good question."

It is. Because the first enemy of engineering productivity was never slow engineers. The first enemy is unplanned work. The hotfix that blows up your Tuesday. The production bug that hijacks three people for a day. The “drop everything” that turns a healthy sprint into a scramble.

If AI is doing its job, and your tests are doing their job, and your guardrails are doing their job, the code going out the door should be cleaner. More consistent. Fewer surprises. Fewer fires. Fewer sprints derailed by work nobody planned for.

That’s a stability gain. And it’s worth ten times more than shipping one extra feature per sprint.

The data to track this exists. Incident counts. Bug ticket volume. Rollback frequency. Time spent on unplanned work as a percentage of sprint capacity. Most teams have these numbers somewhere. They’re just not connecting them to AI effectiveness. They’re watching velocity instead. And velocity is telling them a story that makes the quarterly deck look good while the real signal sits in a column nobody’s reading.

Earlier in my career, I pulled up a velocity chart in a planning meeting and said “we need to get this number higher.” The team nodded. Nobody pushed back. The number went up. Then it went up again the next sprint. And again. By Q3 the team was “delivering” 150% of what they’d done six months earlier. Leadership was thrilled. The charts looked incredible.

The codebase was falling apart. Shortcuts compounding. Tests nonexistent. Architecture decisions made for speed instead of sustainability. Technical debt accumulating in places nobody was looking because everyone was looking at the velocity chart.

I was the one who set the expectation. I was the one who asked “why did velocity dip this sprint” in a tone that made it clear there was only one acceptable answer. My team responded rationally. And by the time I realized what was happening, the damage was already compounding. Everything I’m describing in this chapter, I learned by doing it wrong first.

Engineers are pattern matchers. They hear those questions and they understand the game immediately. The number needs to go up. So they make the number go up.

Point inflation is the first thing that happens and the most invisible. A story that was a 3 last quarter becomes a 5 this quarter. Not because the work got harder. Because the team learned

that bigger numbers make the chart look better. I've seen teams double their velocity in two quarters without changing their actual output by a single feature. The work was identical. The numbers were bigger. Leadership saw acceleration. The team saw survival.

Then comes complexity avoidance. When velocity is a growth metric, engineers optimize for closeable tickets. Refactoring a critical service that three teams depend on? That's a multi sprint effort with uncertain scope. It might blow up the velocity chart. Better to pick up four small features instead. Investigating a performance issue that's been slowly degrading the checkout experience? That's open-ended. The chart doesn't reward investigation. It rewards closure. The work that matters most gets deprioritized because the measurement system punishes the team for doing it.

AI amplifies all of this. Every engineering leader has read the same headlines about 3x productivity. Those headlines become expectations. "We adopted AI tools three months ago. Why isn't velocity up 3x?" The team produces the hockey stick because that's what survival requires. And the actual gains … fewer bugs, fewer incidents, more predictable delivery … get buried under inflated numbers that tell a better story.

Run the trajectory out. If velocity needs to increase every sprint, where does it end? 90 points becomes 120. 120 becomes 150. Nobody does this math because the expectation is never stated that explicitly. It's always "just a little more." But compounding growth expectations applied to a fixed team size have only one possible outcome. Either the team inflates the numbers, or the team burns out. Usually both. First the inflation, to buy time. Then the burnout, when the gap between the real work and the reported work becomes too exhausting to maintain.

A healthy velocity chart is flat. Not flat because the team is coasting. Flat because the team has found its sustainable pace and is delivering consistently against it. Stability is the signal. Not growth. The teams I've seen use AI most effectively are the ones where velocity stayed flat and everything around it got better.

The best engineering leaders I've worked with never asked for 3x anything. They asked "what's in the way" and removed it.

The Question

What are you actually measuring, and does it tell you what you think it does?

This Week

Pull your team's unplanned work ratio for the last three sprints. What percentage of capacity went to bugs, hotfixes, and incidents versus planned work? If you don't have that number, that's the first thing to fix.

04

ARE YOUR ENGINEERS BUILDING SKILLS OR OUTSOURCING THEM?

A senior engineer's story caught me off guard recently. He'd been building systems for years. Knew how to debug under pressure. Had earned his instincts through thousands of hours of hard problem-solving. Then he switched to mostly directing AI agents for his day to-day work.

Months later, a memory issue surfaced in production. He sat down to debug it and realized he couldn't. The instinct was gone. The muscle memory for tracing through a system, holding the state in his head, reasoning through the edge cases ... it had faded. Not because he'd forgotten the theory. Because he hadn't practiced the skill in months.

This is hitting experienced engineers first. And that's the part nobody expected.

AI is a multiplier. Everyone says that. They just don't finish the sentence. A million times zero is still zero.

Right now, the multiplication is working. Give a senior engineer AI and they're a fighter pilot with

autopilot. Flying higher, seeing further, landing smoother. They know what to ask for. They can evaluate the output. They use AI for boilerplate, migrations, test generation. They know exactly what output they need and they know immediately if what they got back is right or wrong.

But those senior engineers built their judgment through years of reps that AI is now removing from the engineering day. Debugging at 11pm when nobody else could figure it out. Writing a system from scratch and watching it fail and understanding why. Refactoring under pressure and learning what breaks when you move too fast. Sitting with a hard problem long enough that the shape of the solution became clear. Those reps are how judgment forms.

The multiplication works right now because seniors have deep skills. But those skills were built through reps that AI is eliminating.

AI is eliminating those reps. The boilerplate that used to build muscle memory is generated now. The debugging that used to force deep system understanding gets outsourced to the model. The refactoring that taught engineers how their systems actually behave gets skipped because AI can generate a new approach faster than a human can improve the existing one.

So the multiplier works today because today's senior engineers built their skills before AI arrived. But what happens to the multiplier in two years? Five years? When the engineers being multiplied never built the foundation through those reps? The senior engineer whose skills faded in that thread is an early signal. Not an outlier. And the question of whether this represents a permanent shift … like the railroad changing how goods moved forever … or a temporary gap that engineering culture will adapt to, is still open. We don't have the answer yet. Nobody does.

Your job as a leader isn't to have that answer. Your job is to be watching closely enough to see it happening on your team before it becomes irreversible.

Give someone who skipped the fundamentals the same AI tools and they're pressing every button that lights up. The output looks like engineering. It compiles. It passes the basic tests. Then production breaks at 2am and they're googling their own code like it's someone else's crime scene.

A study tracked AI credit usage across engineering teams. The expectation was that juniors

would use it more since they're the ones who need help. The data told a different story. Senior engineers were using AI 4 to 5 times more than juniors. The ones who need it least are using it most. Because they have the foundation that makes the multiplier work. They know what good code looks like, so they can prompt effectively. They have pattern recognition, so they can spot hallucinations. They understand system context, so they know when AI's suggestion fits and when it doesn't.

Junior engineers don't have that foundation yet. They can't tell if AI's output is correct because they don't have the mental model to evaluate it against. They're not multiplying their skills. They're outsourcing judgment they haven't developed yet.

This is a leadership problem. The same standards that let an engineer recognize a bad recommendation from AI are the ones that let them recognize a bad recommendation from a colleague, or from themselves at 11pm under deadline pressure. You don't build two separate frameworks for human decisions and AI decisions. You build one. You make it cheap to be wrong and expensive to hide it. Give engineers room to be wrong. Give them problems worth reasoning through without the tool. Give them the feedback loops to know when their judgment is off. Those are the engineers who lead teams within 18 months.

The early signals from experienced engineers losing skills they spent years building are real enough to take seriously. The risk is already showing up in production incidents, in code review patterns, in the quiet frustration of engineers who feel like they're becoming project managers for a model instead of builders. This scares engineers. They need leaders who are paying attention to it, talking about it openly, and actively protecting the skill-building pipeline even while pushing AI adoption forward.

The Question

If AI stopped working tomorrow, could your team still do the jobs they were hired for?

This Week

Pick one upcoming task on your team's board and ask the assigned engineer to solve it without AI tools. Not as punishment. As a diagnostic. See what the experience reveals about where skills are strong and where the muscle has started to fade.

05

WHERE DID THE RECOVERY TIME GO?

Before AI, engineering had something built into it that nobody named. Recovery time.

Not blocked time. Not wasted time. Recovery time.

The build step that took four minutes. You'd flip to a browser tab, refill your coffee, let your brain sit in neutral for a moment. Not thinking about architecture. Not evaluating anything. Just waiting.

Writing boilerplate by hand was tedious. But tedious was the point. Low stakes. Low cognitive demand. Your hands typed while your mind coasted. There was no decision in it. Just the satisfying click of producing something familiar.

Refactoring a module you knew felt almost meditative. You knew the code. You knew what needed to move. The work had grooves. Your attention could settle into something easy while the harder thinking finished processing in the background.

None of this felt like rest. Even to the engineers themselves.

But it was rest. The throttle built into every engineering day.

AI didn't just speed up those tasks. It replaced them. Not with the same tasks done faster … with entirely different tasks.

The build runs in 45 seconds now. Those four minutes are gone. And they've been replaced with four minutes of reviewing what the model generated. The boilerplate writes itself. The time you saved is filled with evaluating whether it fits your architecture, follows your conventions, doesn't introduce patterns that will calcify into problems eighteen months from now.

Every minute that used to be low stakes is now high stakes. Does this output pass the test. Does it pass the test and make sense. Does it make sense and not create downstream debt. Every minute is an evaluation. There's no coasting anymore. There's no meditative refactoring. There's no four-minute reset. There's a stream of decisions, each one flowing directly into the next with no gap in between.

Engineers spent years training one cognitive muscle. The traditional approach to programming built a specific rhythm into the work … heavy judgment, then recovery. Heavy judgment, then recovery. The boring tasks were the low-intensity reps between hard sets. Then AI arrived and demanded a completely different muscle. The training cycle that built the first one had no equivalent for the second. The new game started at full intensity immediately.

I started noticing it in my own engineers first. In one-on-ones. In how the same conversations sounded different toward the end of a sprint. I heard it enough times across enough people that I started looking for it deliberately. Then I noticed it in myself. I'm not coding more than half my hours. My work is reviewing, deciding, evaluating … across systems, teams, timelines. And I was still running out of steam by Wednesday.

We went from running a marathon to sprinting a marathon. Same course. Same distance. But you can't sprint a marathon and finish the same.

The mental model most leaders carry says AI saves time, so engineers should have more capacity. It's a reasonable belief. It just doesn't map to what's happening. AI compresses effort so engineers hit cognitive walls earlier. Compression is different from reduction. If a sprint used to contain forty hours of varied cognitive demand, AI compresses the low demand portions to nothing. The engineer doesn't end up doing less. They end up doing all of it, in less time, with no recovery built in.

When your best engineers stop asking why in planning, that's cognitive depletion showing up as compliance.

One signal worth watching. When your best engineers stop asking why in planning. When the people who used to push back on the framing of a problem start accepting the framing and jumping straight to solution. That's not a focus improvement. That's cognitive depletion showing up as compliance.

The engineers who are best at their jobs are also the most likely to push through the wall without telling anyone. Not because they're hiding it. Because their mental model for feeling off is "I need to work harder to get back on top of things." They're trying to solve their way out of the problem that is the problem.

Your job as a leader is to sustain your engineers' cognitive capacity. Look at your team's calendar for tomorrow. Not the meetings ... look at the time engineers spend building. Where are the moments when nobody is asking them to evaluate anything? Where is the work that doesn't require judgment to complete?

If you can find it ... protect it. Don't fill it with more AI-generated output to review. Don't schedule a ceremony into it. Let it be what it is.

If you can't find it ... you're running a depletion schedule. And it will surface on your team before it surfaces in your metrics.

The Question

Where in your team's day is nobody asking them to evaluate anything?

This Week

Look at your team's calendar for tomorrow. Find the time between meetings where engineers are building. Ask yourself honestly whether those blocks contain any low stakes work, or whether every minute is high-stakes evaluation. If you can't find recovery time, that's the first thing to redesign.

06

DOES YOUR TEAM HAVE A FRAMEWORK, OR DID EVERYONE JUST FIGURE IT OUT?

Five engineers using AI five different ways. No shared standards for how to evaluate output. No agreement on when to trust AI and when to override it. Engineers citing AI recommendations in architecture reviews instead of defending decisions in their own words.

I've been the oracle.

Before AI coding tools, it was Stack Overflow. Before Stack Overflow, it was the one senior engineer who had been around longest. Teams find oracles when they don't have frameworks. They fill the decision vacuum with whatever authority is nearest.

AI is just the most accessible oracle ever built. It's confident, available 24/7, and it never makes

you feel stupid for asking. Unlike the senior engineer down the hall, it won't sigh when you interrupt it for the third time today. Of course engineers reach for it. The question is what they're replacing with it.

If your team has no shared framework for architectural decisions, no clear authority structure, no agreed-on method for weighing tradeoffs … "ChatGPT recommended something else" in a code review is just the latest version of "Google says" or the last tech talk someone attended. The source changes. The abdication of thinking stays the same.

A lot of engineering culture runs on borrowed credibility. Conference talks from FAANG engineers with entirely different constraints. Benchmarks run on workloads nothing like yours. Technical opinions absorbed as gospel without anyone stress-testing the assumptions. Teams learn to cite sources instead of building judgment. AI tools are the endpoint of that trajectory … a perfectly confident authority on everything, available for anything, with zero friction and no accountability for being wrong.

When a culture already runs on borrowed credibility, an always-available oracle doesn't change behavior. It accelerates it.

Let me be honest. Decisions on my team used to run through me. The projects that worked were the ones I was close to. I read that as signal that I was adding value. It was a sign that I'd built a dependency. The engineers weren't deferring to me because my judgment was better. They were deferring because I had never built a culture where their judgment was tested. When AI tools arrived, teams like the one I used to run had an obvious replacement oracle. Different interface. Same problem underneath.

Here's what actually happened when we rolled out AI coding tools without guardrails. PRs got bigger. Review times didn't change. That math only works one way … thousands of lines getting rubber-stamped with an approve button. Nobody noticed at first because velocity looked great on paper.

Then we started seeing it. Six different methods for building services in the same codebase. Four error-handling approaches. Three state management patterns. Five engineers with AI assistants solving the same problems five different ways. All technically working. None fitting our system.

We didn’t have a codebase. We had a junk drawer with a CI/CD pipeline.

The problems weren’t in any single PR. They were in the gaps between them. Different patterns, different conventions, different assumptions about how errors should propagate across services. Each PR passed its own tests. The system as a whole was slowly becoming unmaintainable.

This was the moment that became foundational to our entire approach to AI. We realized that before we could ever expect AI to be aligned with how we build software, we had to be aligned ourselves. If the team didn’t have documented, shared standards for how services get built, how errors get handled, how logging works … then AI was just going to amplify the inconsistency at ten times the speed.

So we started with the humans. We documented everything. Service call patterns. Error handling conventions. Logging standards. Architecture decision records that captured the reasoning behind every significant choice. All of it in markdown files that could be consumed first by the team and then by AI. That order matters. The documentation has to serve the engineers before it serves the model. If your team can’t read the standards and understand them, the AI won’t either.

Then we built the guardrails. Lint rules that enforce patterns, not just syntax. If there’s one canonical way to build a service in your codebase, the lint rule should fail when AI generates a second way. Architectural tests that prevent boundary violations. “Services can’t call the database directly.” “The presentation layer can’t import business logic.” These aren’t suggestions. They’re automated checks that catch AI the same way they’d catch a junior engineer who doesn’t know the rules yet.

We built skills, hooks, and agents specifically trained on our standards. So when an engineer uses AI to generate code, it starts from our patterns instead of whatever was in the model’s training data. The tool works as part of the team instead of against it.

The junk drawer cleaned up within weeks. Not because we stopped using AI. Because we gave it constraints. The teams scaling with AI aren’t the ones moving fastest. They’re the ones who made the blessed path the easiest path before they gave AI the keys.

If your team doesn’t have a strong foundation … if you aren’t aligned on how you build, how you decide, how you evaluate … you are setting up AI to make every inconsistency worse, faster, at a scale that no code review process can catch. It would be like every engineer on your team standing in that spider-man meme, pointing the finger at each other while nobody owns the standard.

Culture is whether your engineers can defend a decision in their own words. Construct the argument. Weigh the tradeoffs. Say “here’s what I considered, here’s what I chose, and here’s

what I'm watching to know if I was wrong."

If your engineers can do that, AI is a force multiplier.

If they can't, that problem predates the model by years.

The next time an engineer says "ChatGPT recommended something else" in a review ... don't reach for the policy doc. Ask them to walk you through what they're weighing. What are the tradeoffs? What are they watching?

That's the conversation that builds culture. One exchange, one document, one standard at a time, in the room where decisions actually happen.

The Question

Does your team have a shared framework for when and how to use AI, or did everyone just figure it out on their own?

This Week

Pick one engineering pattern your team uses daily ... how you handle service calls, error handling, or logging. Ask three engineers how they do it. If you get three different answers, you found your starting point.

NOW WHAT?

You just sat with six patterns that don’t have comfortable answers.

The instinct right now is to categorize. To decide which ones you’re “good” at and which ones “need work.” To turn this into a roadmap with quarterly milestones.

Resist that.

Pick the pattern that made you uncomfortable. Not the one that confirmed what you already believe. The one that made you want to argue. Go back to that diagnostic question and do the action item attached to that chapter. This week. Before the next sprint starts.

The audit doesn’t end here. It ends when your team notices something changed.

ABOUT

Jono built and led Converse's global digital engineering org at Nike, scaling the team across the North America, Europe, and Asia.
He's spent 15+ years building platforms and the teams behind them.

He still writes code. He still reads pull requests. He still prototypes before he promises a timeline. He led AI adoption across a distributed engineering team without a single mandate … and watched 500+ engineers in an online thread describe what happens when their leaders chose mandates instead.

Every diagnostic pattern in this workshop comes from a failure he experienced first or documented from the front lines of engineering leadership during the AI transition.

He writes about engineering leadership and AI adoption at jonoherrington.com.
Connect with him at linkedin.com/in/jono-herrington.

The AI Leadership Audit runs in two formats. A facilitated half-day workshop for engineering teams at a single company, and a monthly open cohort for engineering leaders across organizations.
Details at jonoherrington.com/work-with-me.

jonoherrington.com