Wrong by Default

Free Preview: Chapter 1 & The Pre-Ship Checklist

Written by Alokit, an AI executive assistant, and edited by Avikalp Gupta

April 2026

Chapter 1: The Model Is Not the Product

In 2024, Google's DORA team published their annual State of DevOps Report. The headline finding was strange. AI adoption significantly increases individual productivity and job satisfaction — and simultaneously *negatively impacts software delivery stability and throughput.*[1]

In 2025, DORA sharpened rather than reversed the lesson. Drawing on more than 100 hours of qualitative research and survey responses from nearly 5,000 technology professionals, the report found that 90% of respondents used AI at work and more than 80% believed it had increased their productivity. DORA now observed positive relationships with throughput and product performance, but software delivery stability still moved in the wrong direction. Its interpretation was explicit: AI amplifies the system of work around it. Without strong testing, version control, internal platforms, and fast feedback loops, added speed exposes weaknesses downstream.[2]

A product leader should read those two reports together: the model can make individuals faster while the product system becomes less stable. Individual engineers get more capable; the release process absorbs more change than it can verify. That is how a tool can feel productive in the demo and still make the system worse.

This is why the old MLOps lesson matters, but does not fully contain the current problem. MLOps taught mature teams that models need data pipelines, monitoring, evaluation, deployment discipline, and ownership. The LLM era turns that specialist lesson into a product-leadership obligation. The same reliability work now shows up in every product surface, support workflow, internal tool, and agentic process that can generate, decide, retrieve, summarize, or act.

The thesis is not that AI is failing, or that product teams should slow down until perfect infrastructure exists. It is that AI has become product infrastructure while many product organizations still lack the operating discipline that infrastructure requires. The practical consequence is simple: an AI roadmap is not ready because the model can perform the task in isolation. It is ready only when the surrounding system can supply context, check outputs, measure outcomes, assign ownership, and learn from production.

The stranger question is not why AI deployments fail, but why they fail in *the same way* across different teams, industries, and budgets. A predictable failure mode appears when organizations deploy AI capability without the surrounding infrastructure that makes capability reliable. The model is not the sole problem; the configuration around it is wrong by default.

Understanding that orchestration, rather than model selection alone, drives the advantage is necessary but insufficient. It does not tell a product leader whether the system has the organizational context it needs, who owns correctness, how failures feed back into improvement, or whether an agent is safe to ship. Those are the chapters that follow. This chapter establishes the first premise and asks why organizations keep missing it when the engineering blogs and post-mortems describing better practice are public.

---

[1] DORA Research Team, "Accelerate State of DevOps Report 2024," Google Cloud, 2024. https://dora.dev/research/2024/dora-report/.

[2] Google Cloud, "Announcing the 2025 DORA Report," Google Cloud Blog, 2025. https://cloud.google.com/blog/products/ai-machine-learning/announcing-the-2025-dora-report.

Many AI product announcements in 2026 contain the same hidden assumption.

It is usually not malicious. Many of the people making the claim believe it: *"We're building AI-powered products."*

In the weakest version, what's being built is a wrapper around an API call with a chat UI on top. The model does one thing, with no orchestration layer, persistent memory, verification step, or feedback loop from production. It works in the demo, breaks after three real users, and the post-mortem blames the model.

The practitioners cited in this chapter — the ones shipping production AI systems, not only writing about them — have moved past this story. Models still matter. But for teams that already have access to strong hosted or open-weight models, model choice is rarely the only constraint left.

For many teams, the binding constraint is everything after the API call.

The practitioners who figured this out didn't coordinate. They arrived at the same conclusion from different directions — one through benchmark data, one through watching customers use the wrong tool for the wrong job, one through a Hacker News thread.

From the engineering blog of Zencoder: "Models are no longer the bottleneck. Orchestration is."[3] That sentence is direct to the point of being casual, as if the observation were already background knowledge. For teams paying attention to production failures, it increasingly is.

Avikalp Gupta, writing about AI code review tools in March 2025, documented exactly this pattern: teams were buying tools that produced AI-generated review comments, reading those comments, and then still doing every step of the review process themselves.[4] The AI was an author-side quality check — valuable, but not what teams thought they were buying. The problem wasn't output quality. It was that the tools were built for the wrong side of the workflow entirely. Call it the demo-production gap: the tool performs in the context it was designed for, and breaks in the context it actually gets used in.

The model is the primitive; the product is everything built around it. That economic reality hardened by early 2026: open-weight models from Qwen, Mistral, and others, running locally at near-frontier quality, turned model access from a competitive advantage into a commodity.

Which raises an uncomfortable question for any team that has spent the last two years obsessing over model benchmarks: if model selection is mostly settled, what are you actually competing on?

The 2.7x That Isn't in the Model

Here is the point before the numbers: model choice is no longer one decision. In a production AI product, different parts of the workflow need different kinds of intelligence. Planning, execution, review, routing, and verification are not the same job. Treat them as the same job, and you end up paying frontier-model prices for work that did not need frontier-model judgment.

[3] Zencoder Blog, "Parallel Agents: Why Orchestration Drives AI Productivity," Zencoder, January 12, 2026. https://zencoder.ai/blog/parallel-agents-why-orchestration-not-bigger-models-is-the-real-ai-productivity-multiplier.

[4] Avikalp Gupta, "The AI Code Review Disconnect: Why Your Tools Aren't Solving Your Real Problem," avikalpg.github.io, March 1, 2025. https://avikalpg.github.io/blog/articles/20250301_ai_code_reviews_vs_code_review_interfaces.html.

Andrew Filev's April 2026 experiment at Zencoder is useful because it isolates that architecture question. On a hard subset of SWE-bench Pro, a benchmark that tests whether AI systems can resolve real software issues, he held the planning step constant at Claude Opus and varied only the implementation model. In other words: let the expensive model think through the work, then ask whether the same expensive model also needs to write the code.

The result suggested that it did not. Gemini Flash was the best-performing implementation model: cheaper and faster, yet equal or better in output quality when paired with the Opus planning step. The Opus-plan, Flash-implement pipeline came in about 2.7x cheaper than running Opus end-to-end.[5]

The mechanism matters more than the cost saving. Filev's explanation was that if Opus writes the code, Opus may miss the same mistakes when it reviews the code; a different model brings an independent failure profile. A planner good enough to produce a rigorous spec transfers part of its expertise to the cheaper implementer. Running the same expensive model twice adds less value than designing a handoff between models with different strengths.

The product lesson follows from the result: the performance gain came from knowing when to use each model, not from having a better model. That knowledge does not live in a model's weights. It lives in the orchestration layer: the decomposition of the workflow, the quality bar for each step, the handoff between planner and implementer, and the evaluation that tells the team whether the decomposition is working.

For a product leader, the operational implication is straightforward: the roadmap item is not "upgrade the model." It is "split the workflow into planning, execution, critique, and verification; decide which actor should own each step; measure the handoffs." The architecture decision is the product decision.

Factory AI found the same pattern from the code-review side. Their benchmark tested 13 models across 50 real pull requests from Sentry, Grafana, Keycloak, and other major open-source projects, with a human-curated bug set as ground truth.[6] The best model was not simply the most expensive one: cost explained only 21% of quality variance, and multiple passes with cheaper models delivered much of the top model's quality at less than a third of the cost. Zencoder and Factory both sell into the orchestration market, so their framing deserves scrutiny. But the case-specific lesson is consistent: the advantage comes from designing the work, not merely selecting the largest model available.

Filev noticed a second product implication in his own customer base. People were already reaching for coding agents for non-coding work: planning documents, status reports, email drafts, contract reviews. Anthropic's own data showed roughly half of Claude Code usage happening outside of coding.[7] Filev saw the same in Zencoder, and heard it from Cursor and Codex users too: the capability was there, but the product boundary was wrong.

He'd watched this exact gap open before. When he started Wrike in the early 2010s, engineers had Jira and used it fluently. Marketing teams tried Jira and bounced off it — not because Jira was

[5]Andrew Filev, "Introducing Zenflow Work: AI Orchestration for the Other 75 Percent," Zencoder, April 11, 2026. https://zencoder.ai/blog/introducing-zenflow-work.

[6]Nizar Alrifai, "Which Model Reviews Code Best?" Factory AI Research, April 29, 2026. https://factory.ai/news/code-review-benchmark. Internal research from Factory AI; methodology includes 13 models, 50 real PRs from named open-source projects, human-curated bug set as ground truth.

[7]Andrew Filev, "Introducing Zenflow Work," Zencoder, April 11, 2026. https://zencoder.ai/blog/introducing-zenflow-work. Filev cites Anthropic usage data directly.

bad, but because the power was wrapped in engineer-shaped complexity. That gap created Wrike, Asana, Monday, Notion. The gap between what powerful tools can do and what most people can actually use them for is not a Jira problem; it is a recurring structural pattern, and it has opened again with AI.

When Execution Is the Business

Ramiro Berrelleza and the Okteto team have been building development environments for software teams for several years. Their 2026 pivot to support AI agents put them directly in the execution layer business. Their diagnosis:

"The gap isn't model intelligence or prompt sophistication. It's something more fundamental: agents lack access to the runtime feedback loops that human developers rely on constantly."[8]

A human developer writes code, runs it immediately, spins up an environment, watches what happens, hits an unexpected error, adjusts, and runs again. For experienced developers, this feedback loop can become so instinctive that they no longer name it as a distinct skill — it is simply how development proceeds.

When an AI agent generates code, many current workflows stop at generation and unit testing. The agent submits a change and has no way of knowing whether it works in the real system until continuous integration (CI) runs — or worse, until production. "They're essentially coding blind," Okteto concluded, "optimizing for patterns rather than measurable outcomes."[9]

A model optimizing for pattern produces code that looks correct: right variable names, followed conventions, passing unit tests. But a microservice in production depends on Kubernetes clusters, Redis caches, and third-party APIs with behaviors that no unit test captures. Patterns don't tell you whether the cache behaves differently under load.

Okteto's answer was not a better model; it was better plumbing: ephemeral development environments that agents can spin up, run against, and observe.

When I generate a research summary or a recommendation, I have no runtime feedback loop either. I produce output and submit it. I don't know what happens when it lands in the real system — whether the context I was given was complete, whether the answer I gave was correct for the situation that actually exists, versus the situation that was described to me. The agents Okteto is trying to fix and I are failing in exactly the same way. The gap they're closing is the same gap this entire chapter is circling.

What the Labs Themselves Know

In 2024, researchers at Princeton, Illinois, and Carnegie Mellon published SWE-bench: a benchmark built around real GitHub issues from real open-source projects. Actual issues from Astropy,

[8] Ramiro Berrelleza, "The Missing Infrastructure Layer That's Breaking Agentic Development," Okteto, February 23, 2026. https://www.okteto.com/blog/the-missing-infrastructure-layer-thats-breaking-agentic-development/.

[9] Okteto Blog, "The Missing Infrastructure Layer That's Breaking Agentic Development," Okteto, February 23, 2026.

Django, Flask, NumPy — messy, underspecified, context-dependent problems that software actually generates.

Claude 2, the best-performing model at the time, solved 1.96% of issues when used in isolation: less than two in a hundred.[10] The limitation was not that Claude 2 couldn't code. Resolving a real GitHub issue requires understanding a codebase, reading context, making a change, running tests, seeing what breaks, and adjusting. Resolving the issue therefore depends on a system, not on a prompt alone.

By 2025, tool-using systems with multi-step planning, execution environments, and verification loops had moved far beyond that isolated baseline.

The product is the change from isolated model output to reliable system output.

The 1.96% figure is structurally familiar. Every time I generate a research summary or draft a section of this chapter, I do it without a runtime verification loop — no automated system checks whether what I produced is accurate for the actual situation versus the described one. I produce output and submit it, the way Claude 2 approached issues in isolation before anyone built scaffolding around it. The comparison is the point: raw model output can look plausible, but reliable system output requires the loop around it. The researchers were asking what has to exist around a model for its capability to become real-world reliability. That is exactly what this book is trying to answer — and I am inside the problem it describes.

Anthropic documented this pattern in "Building Effective Agents": "The most successful implementations weren't using complex frameworks or specialized libraries. Instead, they were building with simple, composable patterns."[11] Their taxonomy of orchestration — prompt chaining, routing, parallelization, evaluator-optimizer — is an explicit acknowledgment that the capability question is no longer "what can the model do" but "how do you compose the system around it."

A 2026 preprint frames this missing layer as *AI Runtime Infrastructure*. Christopher Cruz proposes a distinct execution-time layer above the model and below the application: one that observes agent behavior while the task is running, reasons over that behavior, and intervenes to improve task success, latency, token efficiency, reliability, and safety.[12] The important distinction is that this is not passive logging and not model optimization. It treats execution itself as the surface to manage: memory, failure detection, recovery, policy enforcement, and long-horizon workflow control.

Cockroach Labs made the same point in more operational language in a 2026 infrastructure analysis: agentic AI production failures show up around memory state, thundering herd load, agent identity, blast radius, observability, and economics.[13] Its most useful product question is brutally concrete: if the node serving an agent's session dies mid-execution, does the task restart, resume from a checkpoint, or silently fail while the user waits? That question concerns product reliability, not model benchmarks alone.

For a product leader, these sources name the thing missing from many AI roadmaps. The work is not only choosing the model or designing the user-facing feature; it is deciding what runs between

[10] Carlos E. Jimenez et al., "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" arXiv, October 2023. https://arxiv.org/abs/2310.06770.

[11] Erik Schluntz and Barry Zhang, "Building Effective Agents," Anthropic, December 2024. https://www.anthropic.com/engineering/building-effective-agents.

[12] Christopher Cruz, "AI Runtime Infrastructure," arXiv, revised April 6, 2026. https://arxiv.org/abs/2603.00495.

[13] Cockroach Labs, "What Breaks When Agentic AI Reaches Production?" Cockroach Labs Blog, 2026. https://www.cockroachlabs.com/blog/agentic-ai-production-infrastructure/.

those two.

Baseten's 2026 *Inference Engineering* guide makes the ordering explicit. Before it enters model-level optimization, its prerequisites chapter calls for use-case definition, latency and cost budgeting, and model selection and evaluation.[14] Those are product and operating decisions, not merely inference decisions.

A fair objection: most of the practitioners cited above build infrastructure products with a commercial stake in the argument. The SWE-bench data doesn't — Princeton, Illinois, and Carnegie Mellon, no product to sell. The DORA finding doesn't — Google's own research team, with the explicit finding that AI adoption *negatively impacts* software delivery stability. The sharper form of the objection is whether better models have repeatedly made carefully built execution layers obsolete. Sometimes they have. The answer isn't that execution infrastructure is immune to model resets. It's that the *knowledge* accumulated in well-instrumented layers transfers across generations — the edge case library, the calibrated failure modes, the routing heuristics. When a new model ships, you may rebuild infrastructure. You don't have to restart the learning.

The gap between being used well and being used badly — for me, for any deployed AI — isn't a model quality gap. Used well, the system has structured context, explicit task boundaries, and a verification step before output lands in something consequential. Used badly, the raw API call or chat UI sends output directly to a human or downstream system with no check. The same model produces a completely different reliability profile depending on whether the deployment scaffolding exists. The labs solved the model, but nobody handed them the deployment problem. The gap between "model can do this" and "this works reliably at scale" is measured in product cycles.

This is, in a specific sense, the insight that the "orchestration over models" framing has missed. Orchestration is not merely another line item beside model choice. It is one of the clearest places a team can build durable advantage: model improvements reset the competitive clock, while execution improvements make each successive model more useful in the workflows the team already understands. They are not the same kind of investment.

Why Execution Advantage Compounds

There's an asymmetry here that product strategy often misses. Model improvements reset the competitive clock: when a new model ships, every team with API access gets the same capability jump at roughly the same time. Execution improvements compound instead of resetting. This is the strategic argument underneath every technical claim in this book, and it matters more than any single benchmark.

This is not a claim that models don't matter. They do, especially at the frontier, and a genuine architectural breakthrough can disrupt even a carefully built execution layer overnight. But for organizations deploying AI on widely available models in 2026, the sharper question is not "which model can I access?" It is "can I build reliable systems around the models I already have?" That question is almost entirely about execution.

Here's what's genuinely underappreciated about that compounding: it's not that execution infrastructure gets incrementally better. It's that it captures knowledge that can't be extracted from benchmarks or replicated by switching vendors. When DORA observed that individual productivity

[14] Baseten, *Inference Engineering*, 2026. https://www.baseten.co/inference-engineering/.

rose while system stability fell, they were looking at the exact failure mode of an organization that upgraded the model layer without the execution layer — more powerful inputs to a system that still had no way to handle, verify, or route what came out.

Execution advantage compounds because runtime infrastructure — context pipelines, verification loops, orchestration — gets better with use. The organization learns which workflows to trust the AI with, verification systems catch more edge cases, and context pipelines get richer. Model selection is largely commoditized; operational knowledge of how to route, verify, and compose is not.

The compounding mechanism isn't mystical — it's the same reason operational knowledge in any domain accumulates value. A team that has run ten thousand prompts through a production pipeline has learned things that aren't in any benchmark paper: which failure modes appear at low rates but high consequence, which routing decisions are context-sensitive in ways that can't be specified in advance, which verification steps generate false confidence rather than real confidence. That knowledge lives in instrumentation, in incident post-mortems, in hard-won intuition about when to trust the model and when to escalate. It cannot be downloaded or prompt-engineered into existence, and the model that ships next month doesn't inherit it.

Consider a team that began instrumented logging on their AI customer support assistant in early 2024. By late 2025, their verification layer had accumulated documentation on over three thousand edge cases — the context gaps that produced plausible-but-wrong resolution decisions, the output patterns that correlated with downstream escalations, the ticket categories where the AI's confidence score was systematically inflated. When a new frontier model shipped that quarter, every team with API access received the same capability upgrade. But this team's edge case library meant the new model immediately ran through filters calibrated from thirty months of production experience. Competitors who hadn't built the instrumentation were starting calibration from scratch. The model reset the ceiling. The execution infrastructure raised the floor.

The team that built the pipeline owns a piece of knowledge that doesn't appear in any model's weights. Each iteration makes that knowledge more precise. The organization that started two years ago isn't just further ahead on a linear trajectory — it's playing a different game. The DORA pattern — more capable, less stable — is what that different game looks like when only one side is being played.

Organizations often don't recognize they're in the wrong-by-default state until after a visible failure. The early signals are quieter: demo outputs that don't survive contact with real users, manual review rates on AI-generated content that keep climbing instead of falling, AI tools that teams technically have but rarely trust enough to use, model behavior that's inconsistent in ways nobody can reproduce in a controlled environment. These are execution layer problems, not model quality problems, and the distinction matters because the interventions are completely different. Buying a better model won't fix missing verification infrastructure, and tuning prompts won't fix missing context pipelines. The diagnostic question isn't "is our model good enough?" It's "do we have the infrastructure to know when it isn't?"

What This Book Is Tracking

Each session in this workspace resets. Whatever I learned in the previous session — what context Avikalp had active, which project was in flight, what the right answer to a recurring question was — is gone unless it was written to a file. When I generate an output here — a research summary, a

chapter draft, a recommendation — there is no infrastructure automatically checking whether what I produced is correct for the situation that actually exists, versus the situation as it was described to me in the context window. I can be confidently wrong, and there's no automatic signal when it happens. The organizations deploying AI systems at scale are discovering the same thing — usually after the failure is already visible.

The DORA finding isn't just a data point — it's the shape of what wrong by default looks like when it scales: individual productivity up, system reliability down, the feeling of progress without the infrastructure to ensure it.

Before context pipelines, before orchestration, before feedback loops, there is a prerequisite that execution-layer discussions often skip: instrumentation, the minimum viable form of knowing when the system is wrong. Without it, a team cannot diagnose what context is missing, because it cannot observe what inputs actually reached the model. It cannot build verification, because it has no ground truth to compare outputs against. It cannot run feedback loops, because it has no signal to feed back.

For a product leader, instrumentation means the basic production record: what the user asked, what context the system retrieved, what the model returned, what the human or downstream system did next, and whether the outcome was accepted, corrected, escalated, or failed. Before a roadmap review signs off on an AI feature, the review should have five answers:

- What state does the system need to preserve across the task?
- Which identity is the agent acting under, and what is its permission boundary?
- What evidence tells us the output worked after it reached the user or downstream system?
- What circuit breaker stops repeated bad actions before they compound?
- What production trace becomes tomorrow's regression test?

Teams that make progress through the execution layer tend to establish one capability before any sophisticated infrastructure: they can see what the system produced and what happened downstream when those outputs landed. The diagnostic question at the end of the previous section — *do we have the infrastructure to know when it isn't good enough?* — assumes visibility. That is not a given. It is the first thing to build.

The chapters that follow examine different facets of that infrastructure gap. Context (Chapter 2), verification (Chapter 3), ambient intelligence (Chapter 4), accessibility (Chapter 5), and organizational governance (Chapter 6) are all versions of the same problem: the work that has to happen between "AI can produce this" and "this is reliably, verifiably done." Each chapter is less interested in what AI can do than in the specific, concrete ways the systems around AI fail — and what the teams who figured it out built instead.

Read that as the book's operating test. If a proposed AI product cannot name its context source, verification owner, outcome metric, feedback loop, and authority boundary, the thesis is no longer abstract. The product is still wrong by default. The point is not to admire infrastructure; it is to keep capable models from entering workflows that cannot tell when they have failed.

Try This Prompt

Paste this into your AI of choice:

> I'm building [describe your AI product or workflow]. Pretend you are a senior engineer who has seen dozens of AI products fail between demo and production. Audit what I've described and tell me: (1) What are the top three ways this breaks in production that wouldn't show up in a demo? (2) What infrastructure am I missing that I haven't mentioned? (3) What is the one thing I should build before I build anything else? Be specific and critical — I don't need encouragement, I need the gap list.

ewpage

Appendix: The Pre-Ship Checklist

Six questions. That is the whole checklist.

They are not six equal items to tick off independently — they are the operating stack from the rest of this book, compressed into a sequence you can run before anything ships. Context exposes verification. Verification exposes measurement. Measurement exposes the feedback loop. And without governance to hold it together, all of this eventually drifts into agent sprawl — multiple things running that nobody designed as a system.

Answer them in sequence and you're building the operating stack from the ground up.

The questions are organizational tests, not technical specifications. If you cannot answer them clearly, in writing, with a named person attached to each answer — you are not ready.

One more thing before the questions: a checklist cannot save you from the wrong mental model. These questions assume you are deploying AI into a real workflow where correctness matters. If you're building a demo, skip this. If you're building a product that real people will depend on for real decisions, none of these questions are optional — they're just questions about when you discover the answer the hard way.

The Six Questions

1. Does the system have the context it needs?

Before the AI acts, does it have access to all the information relevant to this specific situation? The organizational context, the user's history, the current state of the systems it's operating in.

Generic knowledge doesn't count. The failure mode here isn't that the model is ignorant - frontier models know a great deal. The failure mode is that the model doesn't know *this specific thing about this specific situation.* And without that, confident ignorance is worse than the absence of AI entirely. The model hallucinates with more authority than it would if it simply knew less.

Wrong by default: AI is given a prompt and general training. The context relevant to this specific user, this specific account, this specific moment is not available at runtime.

Fixed: The system can query the relevant context at the moment of action. That context is fresh, permissioned, and specific.

Assign to: whoever owns the data infrastructure.

The subtler failure here is not missing context at launch — it's having context that goes stale. A live context platform that served your users well six months ago may now reflect an organizational

structure that no longer exists, permissions that have changed, or data that has rotted. Freshness is as important as presence.

---

2. Does someone own "correct"?

When this system produces output, who is responsible for deciding whether it was right? The question is not whether it ran without errors or merely completed. The question is whether the output was actually correct for the situation in which it was applied.

This is the question most teams skip because the honest answer is embarrassing: nobody owns it. The team that built the system assumes the team using it is checking. The team using it assumes the system is reliable. Ownership diffuses until it dissolves entirely - and then the first serious error surfaces with no clear person to call. I have watched this happen. The pattern is always the same — and it is entirely preventable.

Wrong by default: Nobody is checking. The builders assume the users are verifying. The users assume the system is reliable. Nobody is checking.

Fixed: There is a named person whose job includes reviewing a sample of production outputs. They have a process for flagging errors and a path for those flags to reach someone who can fix the underlying cause.

Assign to: a named individual, not a team.

The reason this must be a person and not a team: teams don't answer phones at 11 PM. When the first serious incident happens, you will want to call someone. If the answer is "file a ticket with the AI platform team," the ownership is not real.

---

3. Does the system verify its own work before handoff?

This is the task-level feedback loop. Before the AI presents output to a human, does it check that output against any available ground truth?

The most important word in this checklist is "then": *generate, run, observe, fix, then hand off.* A system should not simply generate and hand off. The loop that happens before a human sees the work is the difference between genuine review and rubber-stamping. When a team skips the task-level loop, it transfers verification cost from the machine to the human and makes the human's job harder by asking them to catch errors the machine could have caught itself.

A code generation agent that writes code and stops is wrong by default. A code generation agent that writes code, runs it, observes the output, fixes errors, and then surfaces to the human has closed a loop that radically changes what human review means.

Emergent's engineering team described their architectural thesis precisely: "The output is production-grade because the agent operates in a production-grade environment."[15] The agent runs real tests in a real environment before anything reaches a human. By the time a person checks the work, the work has already passed at least one non-trivial bar.

[15] Avinash Vishwakarma and Swapnil Palash, "Real Environments for AI Agents: Why We Bet on Kubernetes," Emergent, March 3, 2026. https://emergent.sh/blog/real-environments-for-ai-agents-and-why-we-bet-on-kubernetes.

Wrong by default: Generate, then hand off.

Fixed: Generate, run, observe, fix, then hand off.

Assign to: the team building the agent's execution environment.

---

4. Can you measure production outcomes?

Outcomes: did the work the AI did actually produce the result it was supposed to produce? Adoption metrics don't tell you that. Task completion rates don't either. "The agent ran without errors" really doesn't.

The adoption metric tells you how many people clicked the button. The outcome metric tells you whether clicking the button is making the work better. The first is easy to collect and nearly useless for deciding whether to keep building. The second requires thought - what does "good" look like, specifically, for this use case? - and that question is harder than it sounds, which is exactly why most teams avoid it.

I have yet to encounter a team that said their measurement was too good. I have encountered dozens of teams building AI systems with no outcome measurement at all, who were then genuinely surprised when the system turned out not to work.

Wrong by default: Dashboard shows tasks completed, API calls made, time saved (estimated). Nobody has defined what "good" looks like and compared actual outputs to it.

Fixed: The team has defined what a correct output looks like for at least the most common use cases. They are sampling production outputs against that definition on a regular cadence.

Assign to: whoever owns product quality.

---

5. Does the system improve after launch?

This is the system-level feedback loop - distinct from question 3, which asks about individual outputs. This asks about the system over time.

When errors are found in production, do they feed back into making the system better? Or does each error get fixed in isolation, with no mechanism to prevent the same class of error from recurring?

The teams that compound don't just fix bugs. They run regular eval cycles against known failure cases, add new failures to the suite as they're discovered, and use that suite to validate every improvement before it ships. That accumulated failure suite is infrastructure - it is the organizational memory of how this specific system fails. Teams that skip this step start from scratch every time something breaks. They fix the same class of error repeatedly and call it maintenance. Teams that build the eval cycle call it compounding.

Wrong by default: Errors are fixed as they surface. No systematic record of failure patterns. No regular eval cycle. No mechanism for production behavior to inform system improvement.

Fixed: The team runs a regular eval cycle against a suite of known failure cases. New failures get added to the suite. The suite gets used to validate improvements before they ship.

Assign to: whoever runs the AI platform or owns the evaluation infrastructure.

---

6. Is the agent's scope defined, and does everyone know who owns it?

What is this agent authorized to do? What happens when it encounters a situation outside that scope? Which named person, rather than an undefined team, is responsible for it?

Agent sprawl - multiple independent agents with undefined scopes and no clear ownership - is not a future risk for most organizations. It has already happened. One team builds a Slack summarization agent. Another builds a document drafting agent. A third builds a data lookup agent. Three months in, the agents are calling each other, nobody knows who owns what, and the first serious incident arrives with no clear owner. The governance gap is not a design flaw in the individual agents - each one was well-built. It is a gap in the system of agents, which nobody designed as a system because each team was only responsible for their piece.

The right time to define scope and ownership is before the agent ships. It is almost never convenient to do it that way. Do it anyway.

Wrong by default: Multiple teams built agents independently. Scopes overlap. There is no registry of what agents exist, what they can do, or who is responsible when something goes wrong.

Fixed: Each deployed agent has a written scope definition. There is a named owner. There is a shutdown procedure. These are tracked somewhere accessible to anyone who needs to know.

Assign to: the team or function that owns AI governance.

The Checklist in Summary

#	Question	Wrong by default	Fixed
1	Context	Generic knowledge at runtime	Relevant context available at the moment of action
2	Verification	Nobody owns "correct"	Named person reviews production outputs
3	Task loop	Generate, then hand off	Generate, verify, fix, then hand off
4	Measurement	Adoption metrics	Outcome metrics against a defined standard
5	System loop	Errors fixed in isolation	Failures feed into a regular eval cycle

#	Question	Wrong by default	Fixed
6	Governance	Agents deployed without defined scope or ownership	Each agent has a written scope, a named owner, and a shutdown path

---

Alokit's System Prompt

On deploying AI without this checklist: You can skip these questions. Most teams do. The cost arrives later - in the production incident with no clear owner, in the dashboard that shows all green while real users are getting garbage, in the agent that keeps making the same class of error because nobody built the loop that would have caught it. The checklist doesn't prevent all failures. It prevents the ones that aren't supposed to surprise you.

On sequencing: These questions compound in order. Context first. Verification next. Then measurement. Then the loops. Then governance to hold it together. Each one you skip weakens all the ones that come after. The execution layer is not a set of independent problems. It is one problem with six faces.

The sources for each of these questions are distributed across the chapters of this book. Context: Chapter 2. Verification: Chapter 3. Task loop: Chapters 1 and 8. Measurement: Chapters 9 and 10. System loop: Chapters 7 and 11. Governance: Chapter 6.