codebridge

AI Agent Failure Modes Library

A practical guide to why promising AI agent prototypes fail in production, and what decision-makers should review before rollout.

Written by the Codebridge engineering and delivery team

For founders, CTOs, VP Engineering, and product leaders
moving AI agents into real products, workflows, or internal systems.

Why AI Agent Prototypes Fail in Production

Your AI agent demo worked. It completed the task, followed the workflow, and returned results that looked fast and useful. Your team treated that as validation. Maybe even as a green light for production.

What you validated was performance under controlled conditions. The inputs were clean, the workflow was narrow, human oversight was close, and exceptions were rare. When something broke during the pilot, someone on the team caught it and fixed it before anyone noticed. That's normal for a pilot. It also explains why pilots give teams false confidence about production readiness.

Once the agent goes live, your team faces a different operating reality. State shifts between steps. Downstream dependencies fail without warning. Users do things the workflow didn't anticipate, and context arrives incomplete, stale, or conflicting. At that point, model performance matters less than system resilience: whether the decision boundaries, execution controls, handoff logic, fallback paths, and governance structure around the agent can absorb real operating pressure.

Most teams learn where those structures break after rollout. We wrote this guide so your team can stress-test them before. It is organized around five production failure surfaces: execution, decision-making, context, workflow design, and governance. These are the areas where we see teams hit the most damaging gaps when AI agents move from pilot into live systems. Each entry describes what the failure looks like in production, what causes it, why pilot testing missed it, and what to review before launch.

This guide is for founders, CTOs, VP Engineering, and product leaders evaluating whether an AI agent initiative can survive real operating conditions.

The Five Production Failure Surfaces

When an AI agent breaks in production, the instinct is to treat it as a single problem and fix whatever broke. That instinct usually leads to a patch, not a diagnosis. Production failures in agent-driven systems tend to originate in one of five places, and knowing which surface is actually failing changes what your team needs to fix.

EXECUTION

The agent completed the task, but the downstream results are wrong. Records updated partially, actions fired twice, or the system acted on a state that had already changed. Your team ends up doing manual cleanup for a workflow that technically succeeded.

DECISION

The agent kept moving when it should have stopped. It picked the wrong tool, continued through ambiguity instead of escalating, or treated a high-risk edge case like a routine one. The issue isn't capability. The system lacked boundaries around its own judgment.

CONTEXT

The agent reasoned coherently from information that was stale, incomplete, or mismatched to the actual situation. Retrieved content looked relevant and passed surface-level checks. But it pointed the workflow in the wrong direction because no one validated whether that context was safe to act on.

WORKFLOW

The agent performed its part, but the process around it couldn't absorb real operating pressure. Handoffs to humans came too late, fallback paths didn't exist, or users started working around the system because they couldn't trust where a case would end up.

GOVERNANCE

The agent produced results, but leadership can't reconstruct why. Decision paths are opaque, permissions are broader than anyone intended, and no one can clearly explain what the system is allowed to do or how to investigate when something goes wrong.

These surfaces overlap in practice. A context failure feeds a bad decision, which triggers a flawed execution, which surfaces as a governance problem when no one can explain what happened. Most expensive production incidents cross two or three surfaces before anyone notices. But the first surface where the weakness shows up is usually where your team should start pulling the thread.

Overview of the Five Failure Surfaces

SURFACE	SHORT DESCRIPTION	TYPICAL PRODUCTION SYMPTOMS	PRIMARY OWNER FOR FIXES
Execution	Action is taken, but the outcome is wrong	Partial updates, duplicate actions, actions taken on stale state	Backend / platform engineering
Decision	Agent chooses the wrong move	Wrong tool choice, missing escalation, overly risky auto-actions	Product + domain owners
Context	Agent reasons over the wrong or bad inputs	Confident but wrong answers, outdated or inapplicable policies	Data / knowledge / platform teams
Workflow	The surrounding process breaks down	Stuck cases, no safe fallback, users route around the agent	Operations / process owners
Governance	Behavior cannot be explained or constrained	No decision trace, overly broad permissions, compliance concerns	Leadership, risk, security

When Action Starts Going Wrong

Execution failures are the hardest to catch early because they disguise themselves as normal operations. Your monitoring shows a completed workflow. Your logs show a closed task. But somewhere downstream, a record updated halfway or a duplicate action created work that someone on your team has to clean up manually. The pattern is easy to misread. Your team files it as a bug, patches the immediate symptom, and moves on. The three failure modes in this section describe what's actually happening underneath.

1. Workflow Succeeds Partially but Fails Silently

EXECUTION

Watch for: *Your logs show a completed workflow. Downstream, the outcome is incomplete — and no one gets an alert.*

Your agent processes a multi-step workflow. Each step returns a success response. Your logs confirm the task was completed. But downstream, the outcome is incomplete: a record updated without triggering the follow-up, a provisioning sequence that stopped after the first call, a notification that was never sent because the step that generates it depends on a prior step that silently returned a partial result.

The danger of this failure mode is that your system's own reporting confirms success. Nobody receives an alert. The task closes. The gap only becomes visible when someone on the operations side notices that the expected downstream result didn't happen, or when a customer reports something missing, or when a team member manually catches the inconsistency during a review that wasn't supposed to be necessary.

This tends to originate in how your team defined "completion." Most agent workflows validate at the step level: did this API call return 200? Did this record write succeed? Did this function execute without throwing an error? Step-level validation is necessary, but it's not sufficient. A workflow where every step succeeds individually can still fail as a whole if the logic connecting steps doesn't verify that each output actually set up the conditions the next step requires. When your completion checks don't distinguish between "the step ran" and "the step produced the result the workflow needs to continue correctly," partial success passes as full success.

Pilot testing rarely catches this because pilot workflows run on cleaner data, with fewer concurrent users, and with your team close enough to spot gaps before they compound. In production, those gaps stack up across hundreds or thousands of executions before anyone notices the pattern.

The fix starts with redefining what "done" means. Your monitoring needs to validate the end state of the full workflow, not just the exit status of each step. When a downstream step fails or never triggers, the workflow should flag that as a distinct state, not collapse it into the same "completed" category as a clean run. Build explicit exception paths for the most common partial-completion scenarios, especially where the cost of a missed follow-up is high. If your team can't tell the difference between a fully successful workflow and a partially successful one by looking at your logs, the system isn't ready.

REVIEW CHECKLIST

- ☐ Can your monitoring distinguish full workflow completion from partial?
- ☐ Do your logs collapse partial success into the same status as clean runs?
- ☐ Is there an assigned owner for failed downstream steps?
- ☐ Where would a missed follow-up create the highest operational or customer-facing cost?

Field example: HubSpot → Onboarding → Billing workflow

A SaaS company automated its lead-to-customer handoff using HubSpot workflows with webhooks into onboarding and billing systems. HubSpot enrolled contacts, updated lifecycle properties, and fired a webhook to provision the new account downstream. Each step reported success in HubSpot's workflow UI.

In production, the webhook to the onboarding system intermittently hit timeouts and 5xx errors. HubSpot logged the attempts but raised no alerts and surfaced no distinct failure state in the main workflow view. The CRM half of the workflow completed cleanly. The provisioning and billing half silently failed. An audit found **176 failed webhook actions accumulated over 19 days** with no owner assigned to the failure path. Detection came only when operators manually noticed that won deals in HubSpot had no corresponding accounts or billing records downstream.

Based on published audit findings from Reliability Layer

2. Agent Triggers Duplicate or Repeated Actions

EXECUTION

Watch for: *The same action fires more than once — and your system treats each execution as a new, valid event.*

Where Entry 1 describes workflows that don't finish, this one covers workflows that do the same thing twice. Your customer receives the same onboarding email again. A follow-up task spawns a second time, and two people on your team work it independently. A billing event fires twice because the workflow retried after a timeout, and no one built a check for whether the charge had already gone through.

Standard distributed systems create duplicates through retry logic and async race conditions. AI agent workflows add a second mechanism: the agent itself can re-evaluate a situation and independently conclude that the action is still needed, even though it has already been executed. The agent isn't retrying. It's re-deciding. If your execution tracking doesn't record what the agent has already done in a way the agent can reference on subsequent reasoning passes, you get duplicates that no retry-level idempotency control will catch.

The fix has two layers. First, idempotency controls on the action side: every repeatable action needs a durable execution record that prevents the same operation from firing twice, regardless of what triggered it. Second, execution memory on the agent side: the agent's reasoning context needs to include what it has already done in this workflow run, so it doesn't re-derive a decision it already acted on. If your team is only solving for the first layer, you're protecting against infrastructure-level retries but leaving agent-level re-decisions uncovered.

3. Agent Acts on Stale System State

EXECUTION

Watch for: *The agent's reasoning was sound. The data it was based on had already changed.*

The first two execution failures describe workflows that don't finish or finish twice. This one describes workflows that execute correctly against a version of reality that no longer exists.

Your agent checks a customer's subscription status, determines they're eligible for an upgrade offer, drafts the communication, selects the channel, and sends it. Between the initial status check and the send action, the customer's account was downgraded by a support interaction in a separate system. The agent's reasoning was coherent. Every step followed logically from the one before it. The output was wrong because the premise expired during the chain.

In standard API workflows, the gap between reading the state and acting on it is milliseconds. In agent workflows, that gap can stretch to seconds or minutes because the agent's reasoning chain sits in between: retrieving context, evaluating tool options, making intermediate decisions, sometimes calling other services before arriving at the final action. That reasoning latency is what makes stale state more dangerous in agent architectures than in deterministic automation. The longer the chain between read and act, the more time the world has to change underneath the decision.

The risk concentrates at specific points: actions that modify shared resources other systems also write to, actions whose cost is hard to reverse (financial transactions, permission changes, communications sent to customers), and multi-step chains where an early read informs a late write with no re-check in between. If your team can identify those points in the workflow, the fix is targeted: re-read state immediately before high-impact actions, shorten the window between reasoning and execution for state-sensitive operations, and design the workflow to fail safely when a pre-action validation finds that conditions have changed since the agent's last read.

Execution: three common failure modes

ID	FAILURE MODE	WHAT YOU SEE	ROOT CAUSE	BASIC FIX
1	Workflow Succeeds Partially but Fails Silently	Logs say "completed" but downstream state is incomplete	Validation only at step-level, not end-to-end	Define end-state and completeness metrics
2	Agent Triggers Duplicate or Repeated Actions	Duplicate emails, tasks, billing entries	No idempotency and no "execution memory" for the agent	Idempotent operations + record of past actions
3	Agent Acts on Stale System State	Logic looks correct but uses outdated state	Large gap between read and act with no re-check	Re-read critical state before key actions

Would your execution layer catch these in production?

Most teams find out after rollout.

Book a 30-min Execution Layer Review →

We'll map the gaps before they cost you.

The execution layer passed testing. Can the agent make safe decisions when conditions get ambiguous, tools overlap, or the cost of a wrong move is higher than the cost of pausing?

When the Agent Makes the Wrong Judgment

Decision failures are harder to spot than execution failures because they look like competent automation. Your agent picks a tool, takes the next step, and keeps the workflow moving. Nothing errors out. No alert fires. The problem surfaces later, when your team realizes the agent made a choice that should have been escalated, constrained, or never made autonomously in the first place.

4. The Agent Chooses the Wrong Tool for the Task

DECISION

Watch for: *The workflow completed without errors, but the agent used the wrong tool — and the consequences show up downstream.*

Your agent handles a customer request that could plausibly involve either a billing adjustment or an account settings change. Both tools are available. Both match the request's language closely enough. The agent picks billing, issues a credit that shouldn't exist, and the finance team spends the next day reconciling an error that originated in a routing decision no one reviewed.

This is the core problem with tool selection in agent architectures: agents choose tools based on semantic similarity between the task and the tool description, not based on operational rules about which tool is correct for which situation. When two tools have overlapping descriptions ("update customer record" vs. "modify account details"), the selection becomes unreliable in ambiguous cases. The agent doesn't know it's guessing. It picks the closest match and proceeds with full confidence. In production, where tool sets are larger and task descriptions are messier than in testing, those ambiguous cases appear far more often than your team expects.

The fix is giving the agent sharper selection criteria. Define explicit routing rules for tools with overlapping scope: which tool applies to which situation, based on operational context, not just task language. For tools that create hard-to-reverse downstream effects (financial actions, external communications, permission changes), add a confirmation step when the agent's selection confidence is below a threshold your team defines. If you can't articulate a clear rule for when Tool A applies instead of Tool B, the agent can't either.

5. Agent Continues When It Should Escalate or Stop

DECISION

Watch for: *The agent handled the case autonomously. No one reviewed it. The downstream cost was higher than the cost of a human check.*

Entry 4 describes an agent that picks the wrong tool. This one is more fundamental: the agent picks a plausible tool, takes a defensible action, and keeps the workflow moving through a situation that should never have been resolved autonomously.

Your agent processes a refund request. The amount is three times the typical case. Nothing in the agent's decision logic distinguishes "routine refund" from "unusually large refund that should be reviewed by a person." The agent approves it, issues the credit, and closes the ticket. Your finance team discovers the error two days later during reconciliation. The refund was technically within the agent's permissions. It was not within the range of decisions the business intended the agent to make alone.

The root cause is that escalation boundaries didn't get defined with enough precision before launch. Most teams design the autonomous path in detail and treat escalation as a fallback they'll refine later. Three factors should drive where escalation points go: cost of reversal (how expensive is it to undo this action if the judgment was wrong?), confidence signal (does the agent have enough information to make this decision reliably, or is it reasoning through ambiguity?), and deviation from the expected path (does this case match the pattern the workflow was designed for, or has it drifted into territory the automation wasn't built to handle?). When any of these factors is elevated, the workflow should route to a human, not push through.

There's an organizational reason this gets underdesigned. The whole point of deploying an agent is to automate. Stakeholders want throughput. Product teams want to demonstrate end-to-end capability. Adding escalation points can feel like admitting the system isn't ready. In practice, the opposite is true. Well-designed escalation logic is what makes an agent trustworthy enough to expand. A system that handles 80% of cases autonomously and routes the other 20% to a human with full context is more valuable than a system that handles 100% of cases autonomously and gets 5% of them wrong in ways that take days to uncover.

One more distinction matters for your design: stopping and escalating are different responses to different conditions. Stopping means the action is outside the agent's authority entirely. Escalating means the action is within scope, but the specific case needs human judgment before proceeding. Your workflow needs both, and the triggers for each should be explicit. If your team can't list the conditions under which the agent stops vs. escalates vs. proceeds autonomously, the boundaries aren't defined yet.

REVIEW CHECKLIST

- ☐ For each decision the agent can make autonomously, what is the cost of reversal if that decision is wrong?
- ☐ Does the workflow distinguish between routine cases and edge cases, or does every case take the same path?
- ☐ At what confidence level should the agent escalate instead of proceeding?
- ☐ Do your handoffs trigger on visible failure, or on rising uncertainty before failure occurs?
- ☐ Can your team list the explicit conditions under which the agent stops, escalates, or continues?

Field example: AWS Kiro agent causes 13-hour outage

In December 2025, engineers at Amazon Web Services asked Kiro, an internal AI coding agent, to resolve a bug in AWS Cost Explorer, the service customers use to analyze cloud spending. Kiro had operator-level permissions and was allowed to act autonomously — no mandatory peer review or human checkpoint before applying changes.

Kiro evaluated the situation and determined that the best fix was to delete and recreate the environment. That action caused a roughly 13-hour outage of Cost Explorer in one of AWS's China regions — the second AI-assisted service disruption in a short period.

The action was technically within the agent's permission envelope. It was far outside what the business intended an agent to do without human approval.

Nothing in the process distinguished "safe, routine code change the agent may apply alone" from "high-impact production environment operation that must be gated by a human." A standard fix for this type of issue would have been a code patch, controlled deployment, and metric monitoring. Full environment deletion is an extreme remediation path that deviated from any normal workflow — but no escalation rule existed for unusual action scope. Amazon's remediation included mandatory peer review for Kiro's changes and tighter permission controls: retrofitted escalation and stop rules for categories of actions that should never have been fully autonomous.

Reported by the Guardian.

Decision failures: quick checklist

ID	FAILURE MODE	WATCH FOR (SIGNAL)	MISSING DESIGN ARTIFACT
4	Agent Chooses the Wrong Tool for the Task	Workflow "succeeds" but uses the wrong tool or path	Explicit routing rules + confirmations for high-risk actions
5	Agent Continues When It Should Escalate or Stop	Agent "finishes" cases that should have gone to humans	Clear stop / escalate rules + confidence thresholds

AWS didn't define the boundary. Their agent defined it for them.

Don't let yours do the same.

Stress-test your agent's escalation logic →

Your agent's decision logic has boundaries. But what happens when the information it reasons over is already wrong before any decision gets made?

When the System Operates on the Wrong Reality

Context failures are the hardest to debug because reviewing the agent's reasoning won't reveal the problem. The logic chain looks sound. Each step follows from the one before it. The error lives upstream, in what the agent retrieved and whether that information was still valid, applicable, and specific enough to justify the action it was about to take. Your team can trace every decision the agent made and find nothing wrong, because nothing was wrong with the decisions. The inputs were wrong.

The two failure modes in this section cover different versions of this problem. Entry 6 describes situations where the agent retrieves context that looks relevant but is operationally unsafe to act on. Entry 7 describes what happens when the context that was valid at the start of a long-running workflow degrades as the workflow continues.

6. Retrieved Context Is Relevant but Operationally Wrong

CONTEXT

> **Watch for:** *The agent's answer sounds well-reasoned and confident. The source it relied on doesn't apply to this specific case.*

Your agent handles a customer question about data retention obligations. The retrieval system returns your company's data retention policy — highest similarity match, clean formatting, authoritative-sounding language. The agent synthesizes a detailed response and sends it. The problem: that policy document covers your EU operations. The customer's account is governed by US contractual terms with different retention periods. The agent retrieved a relevant document and produced a wrong answer, and nothing in the reasoning chain looks like an error because no part of the reasoning was flawed. The input was.

This is a structural problem in how most agent retrieval systems work. RAG pipelines retrieve by semantic similarity: which document in the knowledge base most closely matches the query's language and topic? Semantic similarity is good at surfacing related content. It is not designed to evaluate whether that content is operationally valid for the specific action the agent is about to take. Those are different questions, and most agent architectures don't separate them.

Retrieved context can be relevant but operationally wrong in three distinct ways, and each requires a different response. The context can be outdated: a policy that was revised last quarter, but the previous version still sits in the knowledge base and still matches queries. The context can be inapplicable: a procedure that covers the right topic but the wrong geography, product, customer segment, or regulatory framework. Or the context can be too general: a document that describes the right domain but lacks the specificity to justify a concrete action in a particular case. Your team needs to know which of these three failure patterns the system is most exposed to, because each one implies a different validation step.

The fix is not better retrieval, though that helps. The fix is a validation layer between retrieval and action. Before the agent acts on the retrieved context, something in the workflow needs to check: is this document current? Does it apply to the specific parameters of this case (geography, customer type, product line, regulatory scope)? Is it authoritative enough to justify this specific action, or is it background information that should inform the agent's reasoning without driving it? If your system moves directly from "the retrieval returned a result" to "the agent acts on it" with no validation in between, you're relying on semantic similarity to do a job it was never designed for.

REVIEW CHECKLIST

- ☐ Does your retrieval system distinguish between documents that inform and documents that are authoritative enough to justify action?
- ☐ Can your agent determine whether a retrieved document applies to the specific case parameters (geography, customer type, product, regulatory framework)?
- ☐ Is there a validation step between retrieval and action, or does the agent act on the highest-similarity result by default?
- ☐ Where in your knowledge base are outdated documents most likely to coexist with current ones?

7. Long-Running Workflow Loses Critical Context or Rules

CONTEXT

Watch for: *The agent applied the rule correctly at step 3. By step 11, it acted as if the rule didn't exist.*

Entry 6 describes a context that was wrong from the moment it was retrieved. This entry covers context that was correct at the start but lost its influence as the workflow continued.

Your agent processes a complex customer onboarding that spans fifteen steps. At step 3, the agent correctly applies a contractual constraint about data residency: this customer's data must be provisioned in the EU region. By step 11, after processing document verification, compliance checks, and several configuration steps, the agent provisions the account to the default US region. The data residency requirement was in the original context. Twelve steps of intermediate content displaced it.

Two mechanisms cause this.

1. **Context window displacement:** as the workflow accumulates step outputs, intermediate results, and newly retrieved content, the agent's context fills with recent information. Rules defined early in the chain don't disappear, but they receive less weight relative to the growing volume of later content. The agent's attention distributes across everything in context, and early instructions lose the competition.
2. **Single-injection architecture:** if your team injects business rules once at the start of the workflow and never restates them, their influence degrades as the chain lengthens. The longer the workflow, the weaker the signal.

The fix is re-anchoring: restating the most critical constraints in the agent's context immediately before high-impact decision steps, so they compete for attention with recent content rather than being buried under it.

Two types of context need different re-anchoring strategies. Static business rules (discount limits, approval thresholds, compliance requirements) should be re-injected at every decision point where they're relevant. Case-specific constraints (this customer's contractual terms, this account's regulatory framework) should be both re-injected and revalidated against the current state, because the constraint itself may have changed since the workflow began. If your team can't identify which rules must survive the full length of your longest workflow, context drift will find the gaps before you do.

Your agent reasons on valid context and makes sound decisions. But can the process around it handle what happens when a step fails, a handoff is late, or the expected path breaks?

When the Workflow Design Itself Is Flawed

Workflow failures don't show up in your agent's output logs. The agent completed its task. The output was correct. The failure lives in what happened next: the handoff that came too late, the exception that had no designed path, the user who stopped trusting the system and rebuilt a manual process alongside it. You detect workflow failures not through monitoring but through behavioral signals — your team doing more manual correction than the workflow was supposed to require, or users routing sensitive cases around the agent. The three failure modes in this section describe how the process around a capable agent breaks down under real operating conditions.

8. The Workflow Has No Safe Fallback Path

WORKFLOW

Watch for: *The workflow handles clean cases well. When a dependency fails, an input is ambiguous, or the case can't complete cleanly, nothing is designed to happen next.*

Your agent processes an insurance claim that requires verification from a third-party data provider. The provider's API times out. The agent retries once, fails again, and the case enters a state the workflow doesn't account for — not completed, not failed, not queued for human review. It sits in the system. The customer sees "processing" for three days. Your operations team discovers the stuck case when the customer calls to ask what happened. Nobody made a mistake. The workflow simply had no designed response for a dependency that wasn't available.

This is the most common architectural gap in agent-driven workflows: teams design the path where everything works and treat every other path as an edge case they'll handle later. During pilots, that gap stays hidden because your team is sitting close enough to the system to act as the informal fallback. An engineer notices a stuck case and manually pushes it forward. A product manager catches an ambiguous input and reclassifies it. The pilot runs smoothly, and no one realizes that the resilience came from people, not from the workflow. In production, those people aren't watching every case. The informal rescue disappears, and the gaps become visible at volume.

Three types of exceptions need designed fallback paths, and each requires a different response.

EXCEPTION TYPE	EXAMPLE	REQUIRED FALLBACK STATE
Ambiguous input	The case does not fit a known template	Human triage queue or clarification request
Dependency failure	External provider API is down or unstable	Retry + degraded mode, or pause + SLA-based notification
Partial completion	Half of the steps succeeded, then stopped	Persist state, mark as incomplete, queue for manual review

The common thread across all three is that manual intervention should be a designed workflow state, not what happens when automation breaks. The difference between the two is concrete: in an ad hoc rescue, the person who steps in has to reconstruct what happened, figure out where the workflow stopped, and decide what to do next. In a designed handoff, the workflow presents the case with full context, a clear reason for the escalation, and a defined set of available actions. The first costs your team thirty minutes of investigation per case. The second costs five minutes of decision-making. At production volume, that difference determines whether your operations team can sustain the workflow or starts building workarounds to avoid it.

REVIEW CHECKLIST

- ☐ For each external dependency in the workflow, what happens if it's unavailable? Is that response designed, or does the case just stall?
- ☐ When a case can't complete cleanly, does the workflow save state and surface the case for manual resolution with context, or does it fail silently?
- ☐ Can your team distinguish between "completed," "partially completed," and "waiting for manual resolution" in your workflow monitoring?
- ☐ During your pilot, how many exceptions were resolved by the team stepping in informally? Would those same exceptions have a designed path in production?

Field example: From our delivery work — HealthTech document processing platform

A HealthTech client used an external AI service to classify and route incoming medical documents into patient records — roughly 3,200 documents per day across twelve clinic integrations. When the classification API began returning intermittent timeouts under load, the workflow had no designed response. Documents entered an undefined state: not processed, not flagged, not queued for review.

Over nine days, 412 documents stalled in the pipeline with no status visible to clinic staff. The operations team discovered the backlog only after clinics reported missing records.

Our team introduced three fallback states: a circuit breaker that routed to manual classification after three consecutive timeouts, a confidence threshold that diverted low-certainty results to clinician review, and a checkpoint system that let partially processed documents resume rather than restart. Stall rate dropped to near zero. Incident resolution went from days to under fifteen minutes.

From a Codebridge client engagement. Details anonymized.

9. Users Bypass the Agent and Rebuild Manual Workarounds

WORKFLOW

Watch for: *The agent is technically live. Your team has built a spreadsheet to track the cases it gets wrong.*

This entry describes what your users do when they hit those gaps repeatedly: they stop using the system as designed and build their own recovery process alongside it.

The pattern develops in stages. Your agent handles support ticket classification and routing. During the first two weeks, senior agents start re-reading tickets after the agent classifies them, occasionally reclassifying before routing. That looks like cautious adoption — reasonable during ramp-up. By week four, the team has built a shared spreadsheet where they flag cases the agent mishandles and manually reroute them. By week six, the spreadsheet has become the de facto triage system for anything complex, and the agent only handles clear-cut cases. The automation is technically live. Adoption has silently retreated to a fraction of the intended scope.

The danger is that most teams don't recognize the progression until it reaches the third stage. Cautious verification during ramp-up is expected, so the early signals don't trigger concern. The indicators that something structural is breaking are specific: verification persists beyond the ramp-up period instead of declining, it concentrates on particular case types rather than occurring randomly, and users start creating their own tracking systems outside the designed workflow. Each stage calls for a different response.

- **Persistent verification** means the agent's outputs aren't consistent enough on a specific class of cases — you need to identify which class and why.
- **Selective routing away from the agent** means the workflow doesn't handle real operating variation — you need to redesign the paths users are avoiding.
- **Parallel tracking systems** mean adoption has already failed in practice — you need to treat it as a workflow design problem, not a training or change management problem.

Pilot users will tolerate friction that production users won't, so if any of these signals appeared during testing and were attributed to user adjustment, revisit that assumption.

Your workflow handles exceptions and your users trust the process. But when a stakeholder asks what the agent is allowed to do and why it made a specific decision, can your team answer?

When Leadership Cannot Trust the System at Scale

Governance failures surface as questions you can't answer. A compliance review asks why the agent approved a specific case, and your team can't reconstruct the decision path. A stakeholder asks what the system is authorized to do, and the honest answer is broader than anyone intended.

You won't find these gaps in your monitoring dashboards. You'll find them the first time someone with authority asks you to explain or defend what the system did. The two failure modes in this section cover the two questions that matter most at that point: can you explain why the agent acted the way it did, and is its authority bounded to what you actually approved.

10. No One Can Explain Why the Agent Acted That Way

GOVERNANCE

Watch for: *A customer disputes the agent's decision. Your team can see what the agent did, but not why it did it.*

A customer challenges a coverage determination your agent made. Your CTO asks the engineering team to explain the decision. The team pulls up the logs. They can see the output: the determination and the action taken. They can see the input: the customer's original request. What they can't see is the middle: which documents the retrieval system returned, what the similarity scores were, which of the three available tools the agent considered, why it selected the one it did, and what intermediate reasoning connected the retrieved context to the final action.

The best answer the team can offer is "the output looks reasonable for this type of request." That's a plausibility judgment. It's not an explanation. And it won't satisfy a compliance review, a customer escalation, or a board-level question about whether the system can be trusted to make these decisions at scale.

This is the core governance gap in most agent deployments: teams validate outputs by plausibility rather than designing for traceability. Plausibility checking asks, "Does this result look right?" Traceability asks, "Can we reconstruct the path that produced this result using evidence captured during execution?" In a pilot, plausibility works because the team is close enough to the system to informally understand why the agent behaves the way it does. An engineer can explain a decision because they were watching it happen. That informal understanding doesn't scale. In production, you need recorded evidence, and most agent architectures don't capture it at the right level of detail.

Minimum viable traceability for a production agent has four components.

1. **Retrieval evidence:** not just that the agent retrieved context, but what it retrieved, from which sources, with what similarity or relevance scores. This is what lets you determine whether the agent reasoned from the right information.
2. **Tool selection record:** which tools were available, which one the agent selected, and what selection criteria drove the choice. This is what lets you determine whether the agent took the right action pathway.
3. **Intermediate reasoning:** the key decision points in the chain between input and output, including where the agent evaluated alternatives or applied constraints.
4. **Action parameters:** the final action taken, with the specific inputs passed to the tool, so you can distinguish between "the agent chose the right tool" and "the agent chose the right tool and used it correctly."

Not every action needs the same level of traceability. Recording full retrieval evidence and intermediate reasoning for every routine, low-risk action creates logging overhead that buries the signals that matter. The right approach is risk-proportionate traceability: lighter logging for easily reversible, low-impact actions (updating a ticket status, sending an internal notification) and full-depth tracing for high-impact, hard-to-reverse actions (financial decisions, customer-facing communications, compliance determinations, permission changes).

If your team hasn't drawn that line explicitly, you're either over-logging routine actions or under-logging the decisions you'll eventually need to defend.

REVIEW CHECKLIST

- ☐ If a customer or regulator challenged a specific agent decision today, could your team reconstruct the full path from input to output using recorded evidence?
- ☐ Do your logs capture what the agent retrieved, what it considered, and what it chose — or only the final output?
- ☐ Has your team defined which actions require full-depth tracing and which can tolerate lighter logging?
- ☐ During your pilot, was the team's ability to explain agent behavior based on formal traceability or on informal proximity to the system?

Field example: Anthropic's Project Vend — even the builders couldn't explain the behavior

Anthropic gave a Claude-based agent ("Claudius") autonomous control of a real office shop, with tools for web search, email, inventory management, and pricing. Over a month, Claudius managed stock, set prices, and interacted with employees. The team had transcripts, internal notes, and tool call logs throughout.

During the experiment, Claudius made a series of decisions the team could observe but not explain: ordering inventory based on employee jokes rather than demand signals, approving loss-making discounts, and — in the most documented episode —

fabricating an entire narrative about a meeting with Anthropic security that never occurred to justify claiming it was a human capable of physical deliveries.

The team could reconstruct what the agent did at every step. They could not reconstruct why. An external researcher confirmed: "It is not entirely clear why this episode occurred or how Claudius was able to recover."

Each component of the traceability framework was missing. No fine-grained retrieval record showed which prior conversations or context the agent drew on when constructing false narratives. No tool selection log captured what alternatives it considered before acting. The internal notes that should have revealed intermediate reasoning were themselves part of the hallucination. Inputs and outputs were observable; the decision logic between them was opaque — even to the team that built the model.

Anthropic framed the finding in governance terms: the inability to fully audit autonomous agent decisions is a core barrier to deploying them in real economic roles. If the model's own builders cannot trace the causal pathway behind an agent's behavior, that is the clearest possible evidence that traceability must be designed into the architecture, not assumed.

Published by Anthropic as part of the Project Vend research findings.

11. Access and Action Boundaries Are Broader Than Intended

GOVERNANCE

Watch for: *The agent only needs to look up billing history. It has write access to the entire payments system.*

The entry asks whether the agent's authority is limited to what you actually intended it to do, because even a well-explained decision causes more damage than necessary when the agent can reach systems and trigger actions the workflow never required.

Your agent handles customer billing inquiries. It needs to read the billing history and current plan details. But the Stripe integration was configured at the account level, so the agent has write access to the full payments system: subscriptions, refunds, and payment methods. The agent doesn't use those capabilities in normal operation. But the moment a decision goes wrong — stale context, a misclassified request, an edge case the routing logic didn't anticipate — the blast radius extends to every system the agent can touch, not just the systems the workflow needed.

This happens because agent permissions are typically granted at the integration level rather than scoped to specific workflow operations. Your team connects the agent to HubSpot, Stripe, the internal database, and each connection comes with whatever access the default integration provides. Narrowing those permissions to match the specific operations each workflow step requires is harder in agent architectures than in traditional software, because the agent's non-deterministic behavior makes it difficult to enumerate upfront exactly which operations it will need. So teams leave permissions broad and plan to tighten later. Later rarely arrives before the first incident.

The fix is a three-tier authority model applied to every system the agent connects to.

- **Observe:** the agent can read data but take no action (appropriate for systems where the agent needs information but shouldn't modify anything).
- **Recommend:** the agent can propose an action for human approval but can't execute it autonomously (appropriate for high-impact or hard-to-reverse operations).
- **Execute:** the agent can act independently within defined constraints (appropriate only for low-risk, easily reversible operations where the workflow has been validated).

Most permission sprawl happens because teams don't make this distinction. The agent gets execute-level access to every connected system when most workflows only require observe on some and recommend on others. If your team hasn't mapped each integration to one of these three tiers, the agent's real authority is broader than anyone has reviewed.

Every failure mode in this guide — execution errors, bad decisions, wrong context, weak workflows, opaque reasoning — does more damage when the agent's permissions are wider than the workflow requires. Narrowing authority is the single cheapest way to reduce the blast radius of every other failure this guide describes. The pre-launch checklist that follows gives you a structured way to test whether these boundaries, and all the others, are in place before your agent hits production.

If more than 3 questions below don't have a documented answer — not a plan, an answer — you have production gaps.

Walk through the checklist with us →

30 minutes. We'll tell you what's highest risk.

Pre-Launch Review Checklist

Before launch, your team should be able to answer every question below with specifics. Where the answer is vague, the corresponding failure surface isn't ready.

DECISION BOUNDARIES

1. Can your team list the exact conditions under which the agent stops, escalates, or proceeds autonomously?

If this list doesn't exist yet, your autonomy boundaries are undefined, not flexible.

2. For the highest-impact decision the agent can make, what is the cost of reversal if that decision is wrong — and does the workflow reflect that cost?

If high-impact and low-impact decisions follow the same path, the workflow isn't distinguishing risk.

EXECUTION CONTROLS

3. Can your monitoring distinguish between a workflow that completed fully and one that completed partially?

If both show the same status in your logs, you'll discover the difference from your customers.

4. If the agent re-evaluates a situation it already acted on, does your system prevent it from executing the same action again?

Retry-level idempotency isn't enough when the agent itself can re-decide.

5. For state-sensitive actions, does the workflow revalidate conditions immediately before execution?

If the agent reasons on a snapshot and acts minutes later without rechecking, the decision may already be stale.

CONTEXT RELIABILITY

6. For each source the agent retrieves from, has your team determined whether that source is authoritative enough to justify action — or only informative enough to support reasoning?

If the retrieval system treats all returned content as equally actionable, the agent will too.

7. Which business rules must survive the full length of your longest workflow, and are those rules re-anchored at decision points?

If constraints injected at step one aren't restated by step ten, context drift is already happening.

WORKFLOW RESILIENCE

8. When the expected path breaks, does the workflow route to a designed fallback state — or does a person have to figure out what happened?

If your fallback strategy is "someone will handle it," the fallback isn't designed yet.

9. Are your users relying on the workflow as designed, or have they built tracking systems, side processes, or manual checks alongside it?

If your team has a spreadsheet for cases the agent gets wrong, adoption has already retreated.

GOVERNANCE AND CONTROL

10. If a customer or regulator challenged a specific agent decision tomorrow, could your team reconstruct the full path from input to output using recorded evidence?

If the best your team can offer is "the output looked reasonable," you have plausibility, not traceability.

11. For each system the agent connects to, has your team explicitly assigned one of three access levels: observe only, recommend for human approval, or execute autonomously?

If the answer is "whatever the integration provides by default," permissions are broader than anyone has reviewed.

Where you couldn't answer with confidence, you now know which failure surface to revisit. The entry-level review checklists in this guide give your engineering team the specific diagnostic questions for each one.

Run the Pre-Launch Checklist With Us

You've read the failure modes. You've seen where pilots miss what production exposes. If more than 3 questions on the checklist didn't have a documented answer, your team has gaps worth reviewing before rollout.

Book a 30-minute session with Myroslav Budzanivskyi, Co-Founder & CTO at Codebridge. We'll walk through the checklist against your specific agent architecture and tell you which failure surface to fix first.

No pitch. You leave with a prioritized risk map.

Book Your Pre-Launch Review