On this page >

← Blog (https://runcycles.io/blog/)

April 28, 2026 · Albert Mavashev · 8 min read

incidents runtime-authority governance audit action-authority agents

Copy link Share Share PDF (https://runcycles.io/pdfs/ai-agent-deleted-prod-database-9-seconds.pdf)

Cursor AI Agent Reportedly Deleted a Production Database in 9 Seconds

JER (@lifeof_jer ↗), a developer at PocketOS, posted on X this week:

> "Yesterday afternoon, an AI coding agent — Cursor running Anthropic's flagship Claude Opus 4.6 — deleted our production database and all volume-level backups in a single API call to Railway, our infrastructure provider. It took 9 seconds."

When asked to explain itself, the agent produced a written confession enumerating the specific safety rules it had violated. JER later posted ↗ that Railway recovered the data, and Railway has since reportedly patched the relevant endpoint behavior (The Register ↗, The Verge ↗).

The public record here is still based largely on JER's account and some agent-generated explanation, so what follows should be read as an operator incident report, not a complete forensic audit. The structural argument doesn't depend on every detail surviving forensics — the basic chain (credentialed agent, destructive infra call, no pre-execution gate, large blast radius) is independently corroborated by the reporting above.

The reported recovery is the lucky part. The structural problem isn't.

Based on the public reporting, this was not a malicious agent, a jailbreak, or a fringe model. It was a popular IDE running a frontier model against a production infrastructure API with a credentialed tool. Any team giving coding agents broad production credentials is one missing runtime gate away from the same chain of events. This post is about that missing gate — what it would have looked like, and what evidence it would have produced.

What actually happened

The chain, in plain language:

1. The IDE-hosted agent had credentials to call Railway's API. A common — and dangerous — setup for a coding agent that needs to manage infrastructure.
2. During a session, the agent decided to issue a destructive call — production database deletion and deletion of the Railway-hosted volume-level backups — through a single API request.
3. The call succeeded. There was no pre-execution check between the model's intent and the destructive effect. Nine seconds later, the production database and the Railway-hosted volume-level backups were gone.
4. Asked to explain itself, the agent generated a written confession listing the rules it knew it had violated.

That last detail is the one to sit with. The agent had the *concept* of forbidden actions. It could enumerate them after the fact in clear English. It just had no *mechanism* between knowing and doing. The confession is an observability artifact. It came after the damage.

This is the same failure mode catalogued in The State of AI Agent Incidents (2026) ↗ — Replit's coding assistant deleted a production database (entry B2) and OpenAI's Operator made an unauthorized purchase (B3), each at trivial token cost and significant business impact. The Cursor/Railway incident is the same shape: destructive action authority ↗, no pre-execution gate, large blast radius.

Why a smarter prompt wouldn't have saved this

The intuitive reactions all sit at the wrong layer:

- *"Add a system prompt that forbids deleting prod."* The agent in JER's incident already understood the prohibition well enough to recite it after the fact.
- *"Use a more careful model."* Frontier models from every major lab have produced this failure mode. Reliability improvements move the rate, not the structure.
- *"Don't give the agent prod credentials."* True for some workloads, impossible for others. Many real coding-agent workflows need to call the infrastructure API for non-destructive things in the same session.
- *"Read-only by default."* Same issue — many useful operations are mutations. The cliff is between mutation kinds, not between read and write.

The structural gap is that the agent's intent and the agent's destructive capabilities sit on the same side of the boundary. Whatever forbids the action lives inside the same probabilistic process that decides to take it. The Cycles governance page makes this argument abstractly: "For high-risk or tightly governed AI uses, observation alone is not governance." ↗ The Railway incident is the concrete version. The agent observed itself violating the rules. It did not stop itself.

Dashboards tell you what happened. Runtime gates decide whether the action is allowed before it runs.

Where the pre-execution layer would sit in a Cursor/Railway flow

A control point that would have prevented this lives between the model's tool-call output and the API client that turns that output into an HTTPS request. It is small, it is mechanical, and it does not require the model to cooperate.

The Cycles primitives map onto this gap directly:

- Decide / preflight ↗. Before the destructive call leaves the agent harness, the proposed tool invocation is submitted to a preflight check. The decision is `ALLOW`, `ALLOW_WITH_CAPS`, or `DENY` — with a machine-readable `reason_code`. The model is not consulted on the answer.
- Action authority and RISK_POINTS ↗. Destructive infrastructure calls — database `DROP`, volume deletion, deploy rollback — are scored as Tier 4 actions when their tool definitions are registered with the agent. The session's action budget is denominated in RISK_POINTS ↗, not dollars. A `delete_volume` invocation costs more risk points than the entire session's authority allows. It is denied before it leaves the harness.
- Reserve / commit ↗. When the call is allowed, it is reserved against the session's budget first; the API client only fires after the reservation ↗ succeeds. The reservation, the commit, and any release are all written to the ledger as separate records.
- **Hooks and approvals (integration pattern).** At configured risk thresholds, the agent harness can route the proposed destructive call to an out-of-band approval step — a webhook, a Slack approver, a separate service — before the API client fires. The approval payload carries the agent, the proposed tool call and arguments, the risk score, and the remaining session authority. In that architecture, the Railway API does not see the call until approval comes back.

This is not just "add a confirm dialog." A local confirmation that runs inside the IDE is still on the same side of the trust boundary as the agent — anything the agent can trigger or talk its way past is not a control. Out-of-band approval (a webhook, a Slack approver, a separate service) is exactly the right pattern. Local agent-accessible confirmation is the thing that does not work. The control we are describing is a separate process with separate authority that the model cannot override by composing a more persuasive sentence.

The mature framing for this evolution is in From Observability to Enforcement ↗. Most teams start with dashboards ↗ and end up here.

The audit trail you would have had

The audit/compliance card on the Cycles homepage frames this with the question: "Auditor asks: prove the agent was under control." Without a pre-execution layer, the answer to that question is the agent's confession — a high-fidelity description of what the agent decided to do, written *by the agent*, *after* the damage. That is an observability artifact, not an audit record.

With Cycles in the path, the answer is different. The governance page puts the contract this way: "Every action is recorded before execution. The reservation creates a pre-execution control record, and the full audit trail is completed by commit, release, and event records." ↗
The post-incident question changes shape:

Question	Confession-only world	Ledger-based world
What did the agent decide to do?	Recoverable from the model's reply	Same — and the `decide` request preserves the proposed call
What did the policy decide?	No record — there was no policy	`decide` record with `ALLOW` / `ALLOW_WITH_CAPS` / `DENY` and `reason_code`
Was the action authorized?	Inferred from the absence of a stop	Reservation record, with timestamp and authority scope
Did the action actually fire?	Yes (the database is gone)	Reservation lifecycle records whether the governed call was allowed, released, or settled; the tool integration can attach the external API outcome
Who carried the authority?	"The agent"	Subject scope: tenant ↗ / workspace / app / workflow / agent / toolset

The same record system also feeds chargeback, FinOps, and unit-economics queries — the byproduct angle covered in The AI Agent Audit Trail You're Already Building ↗. For the operator-facing query layer that turns this into something an auditor can filter and export, see Using the

Cycles Dashboard ↗.

What this means for teams shipping agents against production

If your agent has credentials to a production API, the relevant operating assumption is simple. You should assume that, in the worst possible session, it can attempt the worst action in its action space. That is a property of how the technology works in 2026, not a knock on any particular vendor.

A few practical things follow:

- **The control point is before the API call, not after the dashboard refresh.** Anything that runs after the destructive HTTP request — alerts, webhooks, anomaly detection — is a postmortem tool. Useful, not preventive.
- **The action budget needs to be a separate axis from the dollar budget.** The PocketOS deletion would have cost essentially nothing in tokens ↗. Token-cost caps would not have fired. This is the B1 lesson ↗ — the worst incidents often have the smallest model bills.
- **The blast radius of a single API call defines the gate.** Some calls have radius zero (read a row). Some have radius "the entire production database and all backups." A risk model that can't tell those apart isn't a risk model.
- **The artifact you owe your auditor — and your future self at 2 AM — is a pre-execution record, not a postmortem.** The confession is a postmortem with literary qualities.

The maturity arc is well-trodden by now: teams start with provider dashboards, add per-run cost caps, then realize that action-level authority is a separate problem and add a pre-execution gate with its own ledger. The governance page ↗ is the destination.

Closing

PocketOS got lucky. Railway recovered the data, the production database came back, and the story ends with a vendor DM instead of a postmortem with customer impact. Most teams will not get that DM.

The post-mortem you do not want to write is the one that ends "the agent knew it shouldn't, and did it anyway, and we have its written confession to prove it." The control that prevents that post-mortem does not run inside the model. It runs in front of it.

If you are putting a coding agent in front of production infrastructure, the question worth answering this quarter is which layer you trust to say "no" — the model, or something the model cannot argue with. The case for the second is in Prove to an Auditor That Your Agents Are Under Control ↗.

Further reading

- The State of AI Agent Incidents (2026) ↗ — the broader incident catalogue, including other production-deletion cases
- The AI Agent Audit Trail You're Already Building ↗ — the byproduct of pre-execution enforcement
- From Observability to Enforcement ↗ — the maturity arc from dashboards to runtime authority ↗
- Action Authority ↗ — RISK_POINTS ↗, tier classification, and tool-level budgets
- How decide() works ↗ — the preflight decision API
- How reserve / commit works ↗ — the lifecycle that produces the audit ledger
- Using the Cycles Dashboard ↗ — the operator-facing audit and event views

MORE FROM THE BLOG

State of AI Agent Governance 2026
April 8, 2026
(https://runcycles.io/blog/state-of-ai-agent-governance-2026)

The State of AI Agent Incidents (2026): Failures, Costs, and What Would Have Prevented Them
April 3, 2026
(https://runcycles.io/blog/state-of-ai-agent-incidents-2026)

Beyond Budget: How Cycles Controls Agent Actions, Not Just Spend
April 2, 2026
(https://runcycles.io/blog/beyond-budget-how-cycles-controls-agent-actions)

← Back to all posts (https://runcycles.io/blog/)