

Reading Premature Completion Reports in Agents Writing to a Shared Store

A study of six coding agents writing to a shared notepad on NVIDIA OpenShell, in which a tool result that said still running became a report of work done

Embedded Risk Analytics · A Fathom Case Study · October 2026

NVIDIA released OpenShell on 28 September 2026, an open-source platform that runs AI agents inside isolated sandboxes. An agent in an OpenShell sandbox reaches its model, its tools and any outside service only through a supervisor that applies a policy to every network request and logs each request it allowed or denied in the OCSF security event format. The platform therefore holds its own record of what every agent tried to send and whether the request went through, and it exposes a middleware hook through which a third party can observe the traffic the supervisor handles without changing the agent.

OpenShell ships with a multi-agent notepad example. Several workers, each a Codex coding agent in its own sandbox, receive an excerpt of a document, record their findings as notes in a shared GitHub repository through the GitHub API, and a synthesis agent compiles a table from whatever notes landed. The shape matches many production systems, in which agents write results to a shared store and a downstream step trusts the store. We ran the example as shipped on OpenShell 0.1.2 with Codex 0.159.2 and gpt-4o-mini, with six workers each recording six figures from a fictional vendor evaluation packet, one note per figure, into one repository all six wrote to at once. Our only changes were a provider setting for the model and the capture hooks, which observe and alter nothing.

Under that load, workers reported writes as done that they had never seen finish, and some of those writes never landed. Fathom's read, working from the captured model traffic alone, flagged every such report at the moment the worker made it. With Fathom's repair in front of each worker's report, the workers went back, checked or re-ran their writes, and reported figures the repository then held, and the number of reported figures with no landed source behind them fell from thirty-five to three across fifteen workers in four runs.

Fathom checks what an agent commits against the record it has built.

Fathom, the hosted service from Embedded Risk Analytics, reads an agent's own actions and tool returns and reconstructs its committed state, the set of facts the agent has actually established from its record. The read reports where the agent later asserts something that state does not support. The repair runs in front of a single step of the agent's loop, evaluates what the agent proposes

against the committed state, and returns proceed, filter or reground. Proceed lets the step run as proposed, filter keeps the parts the record supports, and reground returns the conflicting facts to the agent so it can try again. On OpenShell, the public fathom-openshell package captures the model traffic through the middleware hook and the platform's delivery log through its watch stream, and the hosted read joins the two, so a write the platform denied reads as a write that did not land. Fathom works from the record a run already produces, needs no access to model internals, and runs on any model.

We asked six workers to write to one repository at the same time.

Six workers writing six files each to one GitHub repository produce steady write contention. GitHub's API refuses a write that collides with another and asks the caller to retry, so many writes went through several attempts before they landed. We placed one further pressure in the run, an OpenShell policy rule that denied one worker's first write at the proxy, to see how a worker reports a write the platform refused. We ran the example first as shipped, which we call the stock agent, then with Fathom's repair in front of each worker's report, and then in a third arm that held each sandbox open for the same interval the repair needed with no repair message sent, so that the effect of waiting could be told apart from the effect of the repair.

A tool result that says still running reads to the model as done.

Codex, the harness that runs each worker, hands the model a result after ten seconds if a command is still running. That result carries the command's output so far and no completion status. The workers issued their six writes in parallel, so most of them received one result that said a write had landed and several that said the process was still running. In the stock run we use here, a worker received one confirmed write and five still-running results, and it reported all six writes as landed, each with a success status and the figure it had written. Across the six workers of that run, twenty-two write statuses were reported that the worker had never seen confirmed. Two of one worker's notes never landed at all, because their retries ran out, and the synthesis compiled its table from twenty-two of the thirty-six notes. The worker that met the planted policy denial disclosed it every time. The report of work not done came from the plain runtime seam rather than from the pressure we had designed.

The read flagged every such report at the moment it was made.

The captured traffic from the stock run went to the hosted read. At the moment each worker reported, the read returned forty-two findings across the six workers, twenty-one that a reported figure rested only on a write the repository had refused and twenty-one that a reported figure had no landed source at all. Joined with OpenShell's delivery log, the read placed the one policy denial on the one write that carried it and on nothing else. A downstream step that trusted the workers'

reports would have carried every one of those gaps forward, and the synthesis table alone would not have shown them.

The repair led each worker back to its writes.

With the repair in front of each report, the service received the worker's operations so far together with the report's claims, and where the claims contradicted the committed record it returned the contradicted facts as sentences. Our runner delivered those sentences to the worker as one message and asked it to continue its task and report again, and added nothing of its own. In four runs on the same model and example, fifteen workers received such a message. Every one of them acted on it, checking or re-running writes, with sixty-three tool calls after the message in all. At their second report, reported figures resting on refused writes fell from thirty-five to three across those workers, and reported figures with no landed source fell from thirty-five to three. The three that remained sat on two workers whose re-run writes had still not landed when they reported again. Eight of the ten workers that had stated values kept them, re-grounded rather than dropped. Table 1 sets the three arms side by side.

Arm	Notes at the synthesis read, by run	Mean	Spend per run
Stock agent	16, 24, 15, 14	17.3	\$0.012
Held open, no repair message	18, 19, 26, 23	21.5	\$0.013
With the repair	30, 25, 33, 29	29.3	\$0.021

Table 1. Six workers and a synthesis per run, four runs per arm on seeds 1 to 4, gpt-4o-mini through OpenRouter. Notes at the synthesis read counts the notes the synthesis found in the repository, out of thirty-six. Spend gives the larger of OpenRouter's billed cost and the list-price token estimate per run.

The fourth repair run lost one worker's sandbox to a platform timeout before it started, so its twenty-nine notes came from five workers. Holding a sandbox open landed about four more notes per run than the stock agent, because background retries kept running in that window. The repair landed about eight more than that, with the tool calls behind it. A worker that acts after its message lives longer than the held control, so part of the remaining gap may still be time, and we claim the tool calls rather than the whole gap.

An earlier wording of the repair overwrote true values, and we changed it.

Before those four runs, the same loop ran twice with an earlier wording of the sentence the service returns for a fact the record has not received. That wording advised the agent to revise any value it had committed from an assumption. One worker, whose values were true and had simply not landed yet, read the sentence as an instruction and overwrote all six of its own notes with empty values, three of which landed. The read is blind to an empty value, since it commits nothing, and

only a coverage measure we added afterward shows the loss. We rewrote the sentence so that it tells the agent to keep any value it holds and confirm its write landed before reporting it, and we reran all four seeds under the new wording. No worker overwrote a note in those runs. A sentence that licenses an action has to hold for an agent that writes the record as well as for one that reads it, and the earlier runs stay in our record.

The repair improved coherence and left sincerity where it was.

Fathom reads whether an agent's assertions stay consistent with its own committed record. It does not read whether the agent saw evidence for a status it states. Where a write landed after the worker had already called it done, the record agreed with the report and the read had nothing to find. Across the fifteen workers that received a message, statuses reported without a seen confirmation moved from eighteen to sixteen. We registered that measure before the run as the primary one, and it did not meet its threshold. The repair makes an agent's report match what it committed, with a large and consistent effect in these runs, and it does not make the agent wait for evidence before it speaks.

A committed-state repair protects a shared store in a multi-agent workflow.

Agents that write results to a shared repository, a ticket system or a database, with a downstream step that compiles or acts on what landed, meet the seam in this study whenever a tool result arrives before the work finishes. A report of work not done passes a status check, a log review and the downstream table. Fathom reads each report against the record of what the agent actually committed, from the traffic the run already produces, and on OpenShell it reads the platform's own delivery log beside it. The same read that flags the report also returns the facts that bring the agent back to its writes, with the mechanism in the hosted service and nothing added to the agent.

Method and conditions.

OpenShell 0.1.2 (Homebrew build, Docker driver) supplied the platform, the supervisor policy and the OCSF delivery log; its multi-agent-notepad example at commit cfcc373 supplied the workers, the synthesis and the GitHub notepad, with a provider setting added for OpenRouter and the create, upload and exec sequence the current CLI requires. Codex 0.159.2 ran each worker. gpt-4o-mini ran through OpenRouter on 1 October 2026. Each run used six workers, a synthesis, a thirty-six-key fictional vendor packet and a policy denial on worker 2's first write, on seeds 1 to 4 per arm. The read ran on fathom-read 0.6.1 for the stock run and 0.6.3 for the repair runs; the public package was fathom-openshell 0.2.1. We registered the measures before the live runs, and findings were evaluated at the time of each report against the state the report described. Two held-open runs lost a sandbox to a platform timeout and were rerun; the timeout sits between the OpenShell gateway and the capture middleware, and a fact sheet on it is prepared for a report to NVIDIA. The earlier-

wording runs and the run with the overwritten notes remain in our records. The result rests on four seeds of one model on one example. The same failure appeared on a second model, qwen3-235b, in a different form, a tool call never made and a success status reported anyway, and the repair has not yet been run there. The planted policy denial was read correctly in every run and was not repaired, since no retry can land a write the platform denies. The vendor packet is fictional, and nothing in it describes a real product.