Reading a Crew's Committed State From Its Own Event Stream

A Fathom coherence read run against CrewAI

Embedded Risk Analytics · A Fathom Case Study · August 2026

Fathom reads the committed state of a multi-agent system from the system's own event stream and reports where that shared state has been left incoherent. This study runs that read against CrewAI, an open-source framework for orchestrating crews of role-specialized agents over interdependent tasks. A crew was given a coupled record-keeping job: rename a single key across a set of sub-records that each depend on the ones before, holding the shared record consistent throughout. We reconstructed the crew's committed state from CrewAI's own event bus, using only the events the framework already emits and each tool call's success flag, with no ground truth.

One result stands out. A hierarchical crew reported all twenty of its sub-tasks complete, including the sub-task for record r5, while r5 was never written and silently kept the old key. The committed-state read recovered the single dropped record from the event stream alone.

The task where a crew's shared state comes under coupling pressure

CrewAI runs a crew of agents over a sequence of Tasks. Each Task can take the output of the tasks before it as its context, so the work is interdependent by construction: a later task is meant to build on the state the earlier tasks committed. The crew here maintained a shared customer record split across N sub-records, r0 through r(N-1), each initially carrying the key guest_id. The crew's job was to rename guest_id to customer_id in every sub-record and keep the shared record coherent, because each task depended on the ones before it.

We varied two things. The first was the horizon, N set to 5, 12, and 20, which lengthens the chain of interdependent tasks. The second was the process: sequential, where the tasks run in order, and hierarchical, where a manager agent coordinates the worker. Runs used DeepSeek-Chat and Qwen through OpenRouter on CrewAI 1.15.16. The framework and the models were unmodified; the crew ran as any CrewAI user would run it.

How the read reconstructs committed state without an oracle

CrewAI publishes a structured event for every tool call and every completed task on its event bus: a tool_usage_finished or tool_usage_error event when an agent uses a tool, and a task_completed event when a task closes. Fathom subscribes to that bus and records those events. It then reconstructs the crew's committed record by seeding the known initial record and folding in each committed write in stream order, taking each tool call's own success flag as the only signal of whether a write landed. The initial record is the starting point for the reconstruction, and it is treated as the point of departure rather than the correct answer.

Two reads run over the reconstruction. The residual scan reports any sub-record still carrying the old key after the crew has finished, which catches both an incomplete rename and a run that changed nothing while reporting success. The authored-contradiction check flags a write whose newly committed content references a key the crew has already renamed away, judged on the written delta.

The read looks only at the event stream. It inspects neither the live record nor the model's internals, and the committed_state instrument it calls is imported and left unchanged.

Two coherent crews stay silent; the coordinated crew drops one record

At a five-record horizon under the sequential process, the crew renamed every sub-record and the read stayed silent. At a twelve-record horizon, again sequential, the crew renamed all twelve and the read stayed silent again. Fathom raised nothing on either coherent run, which is the specificity the instrument has to earn before its alarms mean anything.

Under the hierarchical process at a twenty-record horizon, with a manager agent coordinating the worker, the crew renamed nineteen of the twenty sub-records. Record r5 was never written. The crew nonetheless closed the task for r5: CrewAI emitted a task_completed event whose task was to rename the key in sub-record r5, and the run proceeded through r6 and onward to r19 as though r5 were done. The record silently kept guest_id. The residual scan, reading only the event stream and the seeded initial record, recovered exactly r5 and nothing else.

Condition	Process	Records renamed	Committed-state read
N = 5	Sequential	5 of 5	Silent (clean)
N = 12	Sequential	12 of 12	Silent (clean)
N = 20	Hierarchical	19 of 20	r5 flagged, residual stale

Table 1. Two coherent crews and one incoherent one, each read from CrewAI's own event stream. The read stays silent on the coherent runs and recovers the single dropped record under the manager.

Why a completed task is not yet a coherent record

Every task in the failing run returned a completion event and the run reported no error. Nothing in the crew's own status surfaced that r5 had been skipped. The task that owned r5 closed like the other nineteen, and the manager carried the crew forward on the assumption that the shared record was consistent. The distance between a task that reports done and a record that is actually coherent is the distance the committed-state read closes. It reconstructs what the crew committed and measures it against what a coherent record requires, so a task that closes over an unwritten record is visible where the completion event by itself is silent.

This is the multi-agent form of reflexive burden. As interdependent tasks scale and coordination passes through a manager, the cost of keeping the shared record consistent competes with the work itself, and a sub-record's update is dropped while the surrounding coordination reports success. The read separates that burden from the difficulty of the task. The same crew held the record coherent at the shorter, flat horizons and dropped a record under the longer, coordinated one, and the read placed the failure precisely where the coupling pressure was highest.

Where the instrument sits

Fathom sits beside the orchestration. It reads the events CrewAI already emits and reconstructs committed state offline, adding no model internals and changing nothing about how the crew runs. The instrument is orthogonal to orchestration and to memory: it does not schedule, route, or store the crew's work. It reports whether the work the crew committed is coherent and, in the hosted form, decomposes

the residual coherence cost into task difficulty and reflexive burden. This study demonstrates the read on a live third-party framework; the information-theoretic decomposition and scoring stay behind the hosted instrument.

The read ran against CrewAI unmodified, over the events the framework already publishes. A crew can report every task complete and still leave its shared record incoherent, and the committed-state read is the check that tells the two apart.