

Operating AI Agents as Distributed Systems: Reliability Controls and User-Visible State

Exzil Calanza

Independent research preprint. This work is not peer reviewed.

Source commit: `ecd71496f10ec95841123e956c5433e81dc310dd`

The reported counts characterize only the frozen experiments and production sample described here; they are not universal reliability estimates.

Abstract

Production AI agents are increasingly described as assistants, copilots, digital labor, or delegated workers. Those metaphors are useful for product communication, but they are incomplete as an operating model. A production agent is a probabilistic planner embedded in a system that calls tools, reads and writes external state, waits on remote dependencies, retries work, and reports progress to users. That execution pattern is closer to a distributed application than to a junior employee. This paper reconciles captured Gemini Deep Research and GPT Deep Research evidence for the associated Platform article and derives a conservative reliability framework. It deliberately excludes unsupported market-wide failure-rate claims and numeric retry-collision claims. The resulting controls are durable external state, idempotent writes, bounded timeouts and jittered retries, causal traces, circuit breakers and admission control, scoped authorization, and explicit human approval for sensitive side effects. The user experience is treated as part of the safety and reliability boundary because long-running work must expose pending, blocked, retrying, approval-required, failed, and completed states without confusing progress with completion.

Research scope and evidence policy

The frozen research run is `run_agents_distsys_2026_06_27`. It contains two independently captured research streams: Gemini Deep Research and GPT Deep Research. The reconciliation artifact accepts claims only where the evidence supports a bounded architectural statement. The editorial synthesis “distributed system with a UX layer” is not represented as a quotation from any vendor or source.

The research rejected three tempting but weakly supported claims. First, it rejected a universal or market-wide percentage for AI-agent failure in production because the reviewed public evidence did not provide a representative denominator. Second, it rejected a numeric claim for how much jitter reduces retry collisions; the qualitative mechanism is supported, but the specific percentage was not. Third, it rejected an attributed production-database incident whose sourcing risked conflating events. These exclusions matter because reliability guidance should not depend on dramatic but unverified statistics.

The accepted synthesis is narrower. Major vendors use workforce language for agents, while technical agent stacks include models, tools, state, orchestration, guardrails, approvals, and tracing. Production agent workflows can therefore experience partial failure, nondeterministic planning, stale state, duplicated side effects, timeouts, overload, rate limits, authorization mistakes, and user-interface ambiguity. These are ordinary systems problems even when the planning component is powered by a language model.

Why the workforce metaphor is operationally incomplete

A human junior employee brings perception, background judgment, and social context that software does not automatically possess. A person may notice that an attachment is unusually large, that a calendar time appears inconsistent, that two instructions conflict, or that a task should stop because the surrounding situation changed. An autonomous agent does not have that human substrate unless the surrounding system explicitly represents the relevant state, constraints, and stop conditions.

An agent receives a request and may transform it into planning, retrieval, tool selection, API calls, validation, mutation, waiting, and reporting. The user sees one task; the system sees a graph of steps and dependencies. A remote call can time out after the server has accepted it. A worker can restart after a side effect but before recording the receipt. A context window can omit prior detail. A rate limit can block one branch while another has already completed. These are partial failures, not personality problems.

The operational implication is that reliability should be designed around state transitions and side effects rather than around the agent’s conversational confidence. The system should be able to prove what happened even if the model is restarted or replaced.

Durable state instead of transcript state

A chat transcript is useful context, but it is not an authoritative workflow ledger. Long-running work needs a first-class state store containing the current phase, completed steps, pending dependencies, idempotency identifiers, approval decisions, timestamps, side-effect receipts, and recovery information.

This distinction is especially important when a workflow pauses or resumes. If the only record of completion is a sentence in conversational history, a later model can repeat the action or misclassify a pending operation as done. A durable ledger lets the system make completion a machine-verifiable property.

A robust state machine should distinguish at least queued, running, waiting on dependency, needs approval, retrying, failed safely, rolled back where possible, and complete. The interface can then render those states directly rather than translating every intermediate event into natural-language assurances.

Idempotency for external writes

Retries are unavoidable in distributed systems. The danger appears when a retry repeats a side effect. A duplicated write can send an email twice, publish the same post twice, create duplicate records, or overwrite state after the user has moved on.

The control is idempotency or an equivalent deduplication contract. Each externally visible mutation should have a stable request key that survives retries and process restarts. A server or adapter can then return the existing result when it sees the same logical request again.

Idempotency does not remove the need for timeout and retry policy. Each dependency should have a bounded timeout budget. Retries should be limited and should use backoff with jitter so many workers do not synchronize into a retry storm. When the retry budget is exhausted, the workflow should enter a visible recovery state rather than looping indefinitely.

Observability across agent steps

Traditional wrapper logs can show that an agent started and ended while hiding the causal path in between. Operators need step-level evidence: the plan, selected tool, tool input, tool output, validation result, retry reason, approval decision, and final assertion.

Distributed tracing provides a useful model. One session or work item should have a correlation identifier, and every consequential step should produce a span or equivalent structured event. This makes it possible to distinguish a model-planning error from a stale state read, permission denial, dependency outage, malformed tool output, or a user-interface error that displayed “complete” while a write remained unverified.

Receipts are particularly important for mutations. A system that sends, publishes, uploads, charges, schedules, or deletes should preserve enough evidence to answer whether the side effect actually occurred. Progress signals are not completion receipts.

Circuit breakers, queues, and admission control

A healthy agent platform should resist cascading failure. If a dependency is returning errors, retrying every request can increase the load precisely when the dependency is least able to handle it. Circuit breakers stop repeated calls after a defined failure threshold and route work to a recovery state until the dependency is eligible for another probe.

Queues and admission control serve a related purpose. They make overload explicit. Instead of launching more work than the system can safely execute, the platform can bound concurrency, keep pending work visible, and prioritize work according to policy. This is preferable to an invisible backlog of half-started agent sessions competing for browser tabs, API limits, or credentials.

Scoped authorization and human approval

An agent with credentials can become a confused deputy if an untrusted instruction causes it to use authority the user did not intend. The control is not merely a better prompt. Authorization should be scoped at runtime.

A safer design gives the agent only the permissions required for the specific operation, ideally through short-lived credentials. Sensitive mutations should have explicit policy checks and, where appropriate, human approval. Approval should be bound to the exact action and evidence rather than to a vague instruction such as “continue.”

Human approval is not a substitute for least privilege. It is one layer in a system that also limits the reachable resources, validates tool inputs, records the decision, and verifies the result.

The UX layer as a reliability surface

Long-running agents create a user-experience problem that is architectural rather than cosmetic. A simple chat interface can collapse many execution states into reassuring prose. That is dangerous if the system cannot distinguish accepted, in progress, blocked, awaiting approval, failed, and complete.

The interface should expose the next action, already-completed actions, unresolved blockers, supporting evidence, and stop or approval controls. It should also avoid presenting a generated answer as proof that an external mutation occurred.

A useful principle is that the user-facing state should be derivable from the durable execution ledger. If the ledger says a publication is pending independent verification, the interface should not say it is published. If the ledger says a retry is scheduled, the interface should show waiting or retrying rather than complete.

Operator checklist

- Store workflow state, approvals, idempotency keys, and side-effect receipts outside the transcript.
- Require deduplication for every external mutation that may be retried.
- Define end-to-end and per-dependency timeout budgets.
- Use bounded retries with jitter and stop conditions.
- Add circuit breakers and admission control so dependency failures do not cascade.
- Trace planning, tool use, validation, retries, approvals, and completion under one correlation ID.
- Issue scoped, short-lived authority for sensitive operations where supported.
- Bind approvals to specific actions and evidence.
- Expose pending, blocked, retrying, approval-required, failed, and completed states in the user interface.
- Treat completion as a verified postcondition rather than a language-model assertion.

Conclusion

The useful production model is not “an AI employee that sometimes makes mistakes.” It is a probabilistic planner inside a bounded distributed execution system. The planner can remain flexible while the wrapper is intentionally conservative: durable, observable, scoped, retry-safe, interruptible, and explicit about uncertainty.

That model changes the reliability question. Instead of asking whether the agent sounds competent, ask whether the surrounding system can survive a partial failure, stale state, duplicated retry, dependency outage, hostile instruction, cancellation, or restart without producing a duplicate side effect or falsely claiming completion. That is a systems-engineering question, and it can be tested.