Stateless Protocol, Stateful Problem: Durability, Retry, and Retention Policy for MCP Tasks

Ajin Baby

cloudandsre.com

2026-09-03 · v0.1 · cloudandsre.com/research/stateless-protocol-stateful-problem/

Abstract

The 2026-07-28 Model Context Protocol release candidate removed the `initialize`/`initialized` handshake and the `Mcp-Session-Id` header, making the protocol core stateless so that any request can land on any server instance. This is a good trade — servers become deployable on serverless and edge infrastructure — but statelessness does not delete state, it relocates it. The Tasks extension carries what remains: a server may answer a long-running request with a durable task handle that the client drives through `tasks/get`, `tasks/update` and `tasks/cancel`. The extension specifies the lifecycle carefully and the *operational policy* not at all, and the project's own 2026 roadmap says so, naming retry semantics on transient failure and result-retention expiry as lifecycle gaps still to close. This paper is one opinionated, measured set of answers to that gap. We contribute (1) a **reference policy layer** — failure classification into transient/permanent/poison, restart triage for in-flight work, deadline/orphan/expiry reaping, and client-side adaptive polling that treats the server's interval hint as a floor; and (2) a **measured study** (E1–E4, deterministic, offline). The study finds: a non-durable store honours 0% of issued handles across a restart against SQLite's 100%, with in-flight work triaged rather than dropped; the poll cost/staleness frontier is **governed by task duration, not by policy** — backoff to a 30 s ceiling saves under 5% of polls on a one-minute task while costing 5.4× the staleness, and saves 86% of them on a thirty-minute task while costing 1.3% of its duration, which is precisely why the extension's `pollIntervalMs` hint is load-bearing rather than advisory, since the client cannot know its own duration class; defaulting unrecognised errors to transient, as most retry libraries do, doubles attempt volume and manufactures 60 dead-letter entries per 100 failures for *zero* additional recovered work when unknowns are not in fact retryable, and only breaks even once most of them are; and, finally, client backoff and server orphan reclamation are **the same number viewed from two sides** — a client following the extension's own advice to back off is cancelled mid-flight by a server reading silence as abandonment, whenever the client's maximum poll gap exceeds the server's orphan window, a cross-party invariant that neither side can check alone and that no part of the specification states.

Keywords: Model Context Protocol, MCP, agentic infrastructure, durable execution, retry semantics, task lifecycle, reliability, site reliability engineering.

---

1. Introduction

Protocols that go stateless do not abolish state; they hand it to somebody else. The 2026-07-28 MCP release candidate [1] removed the connection-level session — the `initialize`/`initialized` handshake and the `Mcp-Session-Id` header are gone — so that, in the release note's framing, any MCP request can reach any server instance without the sticky routing and shared session stores horizontal deployments previously required. For operators this is straightforwardly good.

What moved is the lifecycle of long-running work. The Tasks extension [2] defines it: when a server decides a request will be long-running it returns a `CreateTaskResult` carrying a `taskId`, an initial status, a `ttlMs` and a `pollIntervalMs`, and the client drives the task through `tasks/get`, `tasks/update` and `tasks/cancel` until it reaches `completed`, `failed` or `cancelled`. The extension is careful about the parts it specifies. It requires that a task be durably created before the response is sent. It tells clients to persist task IDs so polling survives a restart, and to respect the server's poll interval. It states that cancellation is cooperative.

It is equally clear about what it does not settle, and so is the project. The 2026 MCP roadmap [3], describing what early production use surfaced, names as gaps still to close "retry semantics when a task fails transiently" together with expiry policy for how long results are retained after completion.

This paper takes that sentence at face value and works out one set of answers with numbers attached. We are deliberately not claiming novelty against a literature — the gap here is documented, dated, and asserted by the standards body itself, which is a stronger footing than a survey. The contribution is that the answers are *measured*, and that two of the four measurements were surprising to us.

Our contributions:

1. **A reference policy layer** (§4–§5): failure classification, restart triage, a three-sweep reaper, and adaptive polling with the server hint as a floor.
2. **A measured study** (§6): handle durability, the poll cost/staleness frontier, the price of the default failure classification, and a client/server coupling.
3. **Two findings that change advice** (§6.2, §6.4): backoff is only correct in a duration regime the client cannot observe, and a client that backs off correctly can be cancelled by a server that is also behaving correctly.

2. Background

What the extension specifies. Five lifecycle states, three terminal (`completed`, `failed`, `cancelled`) and two not (`working`, `input_required`). Durable creation before response. A TTL and a poll-interval hint on task creation. Cooperative cancellation. Optional push via `notifications/tasks`, with polling as the default. This is a well-drawn boundary and nothing below is a criticism of it: an extension that also mandated retention policy would be mandating operational choices that belong to deployments.

What it leaves open, precisely. Four things, and it is worth being exact because overstating the gap would be the easy error. First, *retry*: a task can reach `failed`, and nothing says which failures deserve another attempt. Second, *retention beyond a number*: `ttlMs` is a value, not a policy — nothing says what a server should do about tasks whose clients never return, or how expiry interacts with terminal state. Third, *restart triage*: durable creation is mandated, but nothing says what a restarting server owes work that was `working` when it died. Fourth, and least visible, the *coupling* in §6.4: the extension advises clients to back off and gives servers a TTL, and never relates the two.

Why this is an SRE problem. Each of these is a control loop with a sensor, an actuator, and a failure mode when its timescales are wrong — the same shape treated in remediation-loop stability analysis [7] and in budget-governed authority [6]. The reaper is a sweep whose period must be short relative to the windows it enforces; the retry classifier is an admission decision whose default sets the system's load amplification under novel errors; poll backoff is a sampling rate traded against detection latency, which is the same trade as alerting on a slow-burning budget.

3. Related Work

Durable execution and retry orchestration. Reliable Actors with Retry Orchestration [4] gives the strong form of the guarantees at issue: failed invocations are retried, completed invocations are never repeated, and a happens-before relationship is preserved across failures. That work operates inside an actor runtime with control over the whole call stack. An MCP server has far less: it sees one request, may be one of many interchangeable instances, and cannot assume its client will ever return. The policy layer here is the weak, deployable version of those guarantees — idempotency-keyed creation rather than exactly-once, triage rather than transparent resumption.

Cost and budget control for agent runtimes. Token Budgets [5] catalogues 63 production budget-overrun incidents and enforces caps through an affine type system, checking before the call rather than observing after. This paper's retry policy is a neighbouring control: an attempt ceiling is a budget on retries, and §6.3 measures what a badly chosen default costs in attempts — the same currency [5] protects, spent in a different place.

Governance of standing authority. Error-budget-governed authority [6] and control-theoretic analysis of remediation loops [7] supply the framing in §2 for why these are loops rather than settings, and specifically the cascade rule invoked in §6.4: a governing loop must run on a slower timescale than the loops it governs.

Executable incident evidence. The companion corpus work [8] supplies the discipline applied to §6: a claim about a control is worth what its measurement is worth.

The gap. None of the above addresses the MCP Tasks lifecycle, and no academic treatment of it exists that we could locate. We make no priority claim on that basis — absence of a search hit is weak evidence — because the paper does not need one. The roadmap [3] names the gaps; this paper measures four candidate answers.

4. Failure Classification and Retry

4.1 Three classes. A failure is **transient** (the call may succeed if repeated), **permanent** (it will fail identically forever), or **poison** (transient in origin, but retried to exhaustion, and therefore operationally permanent). Classification runs off substring markers over the error text and, where the server surfaces them, HTTP-shaped status codes; the attempt ceiling overrides everything, so a transient error that exhausts its budget is reclassified poison and dead-lettered rather than retried forever.

4.2 The default is the design decision. For an *unrecognised* error the classifier defaults to **permanent**, which is the opposite of the common library convention. The reasoning is operational: an error you cannot classify is an error you do not understand, and retrying what you do not understand is how one bad tool call becomes a thundering herd. That argument is only worth as much as its number, so §6.3 prices it.

4.3 Backoff. Capped exponential backoff with full jitter and a hard attempt ceiling, with retry delays advertised to the caller rather than slept through inside the library.

5. Retention, Reclamation and Restart

5.1 Three sweeps, in order. The reaper runs on a timer and sweeps in a fixed order that matters. *Timeouts* first, so a task past its execution deadline becomes terminal and earns a retention window. *Orphans* next, over what is still live: a stateless server has no socket to notice a client leaving, so silence is the only available signal, and a task unpolled beyond `orphan_after_s` is presumed abandoned. Orphans are **cancelled, not deleted**, so a client returning inside the retention window learns what happened instead of receiving a bare not-found. *Expiry* last, the only sweep that deletes rows.

5.2 Restart triage. On startup against a durable store, in-flight work is triaged rather than resumed or dropped: past its deadline it is failed as a timeout; out of attempts it is poisoned and dead-lettered; otherwise it is handed back to the caller as resumable, with an incremented attempt count. Nothing silently drops a handle the client still believes in — the property §6.1 measures.

5.3 Polling. The client scheduler backs off multiplicatively with jitter and treats the server's `pollIntervalMs` as a **floor, never a ceiling**: the server knows how long its own work takes, so it may tell a client to slow down, but it may not talk a backed-off client into speeding up. §6.2 shows why that hint carries more weight than it appears to.

6. Measured Study

All results from `examples/paper_study.py`, computed by `mcp_task_reliability.sim.study` and pinned by 27 tests in `tests/test_paper_study.py`. Every timing decision goes through an injectable clock, so the study runs offline and identically on any machine; jittered cells report seed counts, unjittered cells are exact.

6.1 E1 — does the handle survive the restart? The extension requires durable creation. E1 measures what an implementation that ignores that requirement costs, from the only vantage point that matters: the client is holding twenty handles it was told were durable, and after the server restarts, how many still answer?

Table 1 — handle survival across a restart (20 handles, 12 completed, 8 in flight).

store	resolve after restart	lost	resumable	survival
in-memory	0	20	0	0%
SQLite (WAL, synchronous=FULL)	20	0	8	100%

The in-memory result is total: not merely the eight in-flight tasks but the twelve *completed* ones vanish, because their results lived only in the dead process. A handle the server did not durably write is a handle that goes missing across a deploy, and the client cannot distinguish that from a server that never existed. The eight in-flight tasks come back as *resumable* rather than lost — triaged per §5.2, with the deadline and attempt checks applied — which is the difference between a restart the client can reason about and one that simply eats work.

6.2 E2 — the poll cost/staleness frontier. Stateless MCP removed the connection, so polling is the default path. Fixed-interval polling buys freshness with sustained load; backoff buys quiet with staleness. Neither the extension nor the roadmap says where to sit. Completion time is swept across a ±10% window rather than held fixed, so staleness measures the operational quantity — how stale the client's view is when a task it does not control finishes — rather than where one instant happens to land in one schedule. 220 runs per adaptive cell (20 seeds × 11 phases).

Table 2 — polls issued and staleness, by policy and task duration.

policy	~60 s task		~300 s task		~1800 s task	
	polls	stale p50	polls	stale p50	polls	stale p50
fixed 250 ms	240.4	0.10 s	1200.0	0.00 s	7200.2	0.00 s
fixed 1000 ms	60.4	0.40 s	300.0	0.00 s	1800.2	0.00 s
fixed 5000 ms	12.5	2.40 s	60.4	2.00 s	360.5	2.00 s
adaptive, max 10 s	15.0	3.98 s	46.7	3.97 s	247.1	3.40 s
adaptive, max 30 s	11.9	12.98 s	22.4	11.60 s	89.3	11.60 s
adaptive, max 60 s	11.8	21.65 s	17.3	22.01 s	50.4	22.69 s

Read as absolute numbers this table says backoff is a large win, and that reading is wrong. The operational quantity is staleness **relative to the task's own duration**: twenty seconds is a third of a one-minute task and a rounding error on a half-hour one.

Table 3 — median staleness as a fraction of task duration.

policy	~60 s	~300 s	~1800 s
fixed 5000 ms	4.0%	0.7%	0.1%
adaptive, max 10 s	6.6%	1.3%	0.2%
adaptive, max 30 s	21.6%	3.9%	0.6%
adaptive, max 60 s	36.1%	7.3%	1.3%

The regime, not the policy, decides. On a one-minute task, backing off to a 30 s ceiling saves under 5% of the polls that fixed-5 s issues (11.9 vs 12.5) and pays 5.4× the staleness for it — 21.6% of the task spent not knowing. On a thirty-minute task the identical policy pair inverts: adaptive max-60 s issues 50.4 polls against fixed-5 s's 360.5, a 7.1× reduction, for staleness worth 1.3% of the task. The same configuration is wasteful in one regime and obviously correct in the other, and the ratio between them is 28×.

The consequence is a protocol-design one. **The client cannot know its duration class.** It has issued a call; it does not know whether the server is running a 40-second query or a 40-minute batch job. The one party that does know is the server, and the extension already gives it the channel — `pollIntervalMs` on task creation. E2 is the argument that this field is not a nicety for reducing chatter but the mechanism that puts the client in the right regime, and that a server returning a constant hint regardless of expected duration has silently disabled it.

6.3 E3 — what the default classification costs. §4.2 defaults unrecognised errors to permanent against convention. Workload: 100 failures — 20 recognised-transient, 20 recognised-permanent, 60 unrecognised — with an attempt ceiling of 3. We sweep the fraction of the *unrecognised* that would in fact have succeeded on retry. Recovery is modelled exactly as the classes define it: a truly transient failure succeeds on attempt 2, a permanent one never succeeds. No randomness; the table is exact.

Table 4 — cost of each default, by how retryable the unknowns really are.

truly transient	default	attempts	recovered	wasted retries	lost work	dead-letters
0%	permanent	120	20	0	0	0
0%	transient	240	20	120	0	60
25%	permanent	120	20	0	15	0
25%	transient	225	35	90	0	45
50%	permanent	120	20	0	30	0
50%	transient	210	50	60	0	30
75%	permanent	120	20	0	45	0
75%	transient	195	65	30	0	15
100%	permanent	120	20	0	60	0
100%	transient	180	80	0	0	0

The asymmetry is the result. Defaulting to permanent has a **flat cost**: 120 attempts and zero dead-letters regardless of the error mix, because it never retries what it does not recognise. Its exposure is one-sided and legible — it loses recoverable work in proportion to how retryable the unknowns are, 15 tasks per 25 percentage points.

Defaulting to transient has a cost that is worst exactly where you know least. At 0% retryable it spends 240 attempts against 120 for identical recovered work, and manufactures 60 dead-letter entries out of nothing — a doubling of load and a queue of operator toil, bought for zero benefit. Only at 100% does it become unambiguously right, and even there it buys 60 additional recovered tasks for 60 additional attempts: exactly one attempt per unit of work saved.

The operational reading: an unrecognised error is by definition one whose retryability you have not measured, so choosing the default that is catastrophic when unknowns are unretryable is choosing to be wrong in the case you cannot rule out. Default to permanent, and promote specific errors to transient by naming them — which converts the classifier's marker lists into an artifact that records what you have actually learned about your dependencies.

6.4 E4 — the coupling nobody writes down. §6.2 tells clients to back off. §5.1 tells servers to reclaim tasks whose clients have gone quiet. These are the same number viewed from two sides, and nothing in the extension, the roadmap, or either default relates them. E4 runs a one-hour task with a backing-off client and a reaper sweeping every 30 s, and asks whether the task survives its own client's good behaviour.

Table 5 — task outcome by client backoff ceiling and server orphan window.

client max backoff	orphan window 900 s	orphan window 120 s
10 s	survived	survived
60 s	survived	survived
300 s	survived	**orphaned at 780 s**
900 s	survived	**orphaned at 780 s**

The rule falls straight out and holds across the sweep: **a task is orphaned exactly when the client's maximum poll gap exceeds the server's orphan window.** Both parties are behaving correctly and independently. The client backed off because §6.2 and the extension's own guidance told it to. The server reclaimed because a stateless server has no other signal that a client has walked away. The result is that live work is cancelled underneath a client that is still waiting for it,

and the client discovers this only on its next poll — by which time the server has, correctly per its own policy, stopped doing the work.

This is a cross-party invariant that **neither side can verify alone**. A client does not know the server's orphan window; a server does not know the client's backoff ceiling. It is also, in the cascade framing of [7], a timescale violation: the reclamation loop is running faster than the observation loop it is meant to supervise. The shipped defaults here (10 s client ceiling, 900 s orphan window) clear it by 90×, which is the sort of margin that hides a hazard rather than removing it — the failure appears only when one side is tuned in isolation, which is precisely when nobody is looking at the other.

The clean fix belongs in the protocol rather than in either implementation: the server already returns `ttlMs` and `pollIntervalMs` at task creation, and a client that treated `ttlMs` as a hard upper bound on its own backoff would make the invariant checkable at the one moment both numbers are in the same place. We offer that as a suggestion to the extension, not as a claim to have validated it.

7. Discussion and Threats to Validity

Simulation, not deployment. Every result is from a deterministic offline model with an injectable clock. There is no network, no real MCP server, no live model, and no contention. The claims are therefore about *policy dynamics* — what a classification default costs in attempts, how staleness scales against duration, when an orphan window and a backoff ceiling collide — and these depend on the state machine and the arithmetic rather than on transport behaviour. Numbers that would move under real conditions (absolute poll latency, SQLite throughput, retry success probability) are not claimed as production figures.

E3's recovery model is definitional, not empirical. "Transient means it would succeed on retry" is how the classes are defined, so modelling recovery as success on attempt 2 makes the table exact and also makes it a statement about classification arithmetic rather than about any real dependency's recovery curve. A real system's transient errors succeed at some rate below 1 and after some delay; both effects reduce the value of retrying and therefore *strengthen* the argument for the permanent default rather than weakening it.

The error corpus in E3 is synthetic and small. Three error strings standing in for three classes. The 60/40 recognised-to-unrecognised split is a stipulation, not a measurement. What the sweep establishes is the shape of the two cost curves and the asymmetry between them, which is invariant to the split; the crossover point is not.

E1 compares two backends, not all backends. In-memory versus SQLite is the honest floor and a reasonable single-node ceiling. It says nothing about the case the stateless core actually enables — many interchangeable instances behind a shared store — where the interesting failures are concurrent writes and partial visibility rather than total loss.

E4's sweep is coarse. Four backoff ceilings against two orphan windows, one task duration, one reaper period, one seed for the jittered schedule. The invariant it demonstrates is arithmetic and we believe it general, but the 780 s cancellation instants are properties of this configuration and should not be read as a characteristic time.

The policy layer is one set of choices. Permanent-by-default classification, cancel-don't-delete orphan handling, and triage-don't-resume restart semantics are defensible and defended above, but they are choices. A deployment with cheap idempotent tools and expensive human toil should reasonably invert the first of them — which is why §4.2 exposes it as a parameter and §6.3 prices it rather than asserting it.

A specification in motion. This addresses the 2026-07-28 release candidate and the Tasks extension as of 2026-09-03. The extension's own documentation notes that hosts vary in support, and the roadmap describes the lifecycle as still

settling. Conclusions about specific unspecified behaviour have a shelf life, and the gaps this paper fills are ones the maintainers have said they intend to close.

8. Conclusion and Future Work

Making a protocol core stateless is a deployment win that relocates rather than removes the state, and the MCP Tasks extension is where it landed. The extension draws its boundary deliberately and the project has said, in its own roadmap, which operational questions remain open. This paper answers four of them with measurements rather than assertions, and two of the answers are not what we expected going in.

Poll backoff, the obvious optimisation, is only correct inside a duration regime the client cannot observe — which turns the extension's `pollIntervalMs` from a chatter-reduction hint into the mechanism that places a client in the right regime at all. And a client that backs off exactly as advised will have its live work cancelled by a server that is reclaiming exactly as it should, whenever the client's poll gap crosses the server's orphan window: an invariant spanning two parties, checkable by neither, and stated nowhere in the specification. The other two results are less surprising and more immediately actionable — durability is binary and a non-durable store loses completed results as thoroughly as in-flight ones, and defaulting unrecognised errors to transient doubles attempt volume and manufactures dead-letter toil in exactly the case you cannot rule out.

Future work: validate the frontier against a real server and transport, where poll cost includes connection overhead the model omits; extend E1 to the multi-instance shared-store topology the stateless core exists to enable; and take the §6.4 invariant to the extension maintainers, since the durable fix is a sentence in a specification rather than a feature in any one implementation.

References

(Author lists, titles and identifiers resolved against arXiv metadata and live URLs, 2026-09-03.)

[1] Model Context Protocol project. The 2026-07-28 MCP Specification Release Candidate. https://blog.modelcontextprotocol.io/posts/2026-07-28-release-candidate/ (accessed 2026-09-03). [2] Model Context Protocol project. Tasks — Asynchronous task execution for long-running MCP operations. https://modelcontextprotocol.io/extensions/tasks/overview (accessed 2026-09-03); specification at https://github.com/modelcontextprotocol/ext-tasks [3] Model Context Protocol project. The 2026 MCP Roadmap. https://blog.modelcontextprotocol.io/posts/2026-mcp-roadmap/ (accessed 2026-09-03). [4] O. Tardieu, D. Grove, G.-T. Bercea, P. Castro, J. Cwiklik, and E. Epstein. Reliable Actors with Retry Orchestration. arXiv:2111.11562, 2021. [5] S. Khan. Token Budgets: An Empirical Catalog of 63 LLM-Agent Budget-Overrun Incidents, with an Affine-Typed Rust Mitigation as a Case Study. arXiv:2606.04056, 2026. [6] A. Baby. Error Budgets for Autonomy: SLO-Driven Authority Management for Autonomous AI Operators. Preprint, cloudandsre.com, 2026. https://cloudandsre.com/research/error-budgets-for-autonomy/ [7] A. Baby. Stable by Design: A Control-Theoretic Account of AI-Driven Self-Healing Remediation Loops. Preprint, cloudandsre.com, 2026. https://cloudandsre.com/research/stable-by-design-remediation-loops/ [8] A. Baby. From Incident to Failing Build: Incident-Derived Regression Testing for AI-Agent Guardrails. Preprint, cloudandsre.com, 2026. https://cloudandsre.com/research/incident-derived-regression-testing/ [9] A. Baby. mcp-task-reliability: Durability, Retry and Retention Policy for the MCP Tasks Extension. Companion artifact, 2026. https://github.com/ajinb/mcp-task-reliability (public, Apache-2.0).