State, Not Tokens: Repository-Scale Agent Reasoning Is Bound by State Architecture

Cary Palmer
Independent Researcher, Dallas, TX
https://github.com/professorpalmer · https://www.linkedin.com/in/cary-palmer-a30557175/

Preprint. All reported numbers are produced by the machine-checkable oracle and are reproducible from the public code and data: https://github.com/professorpalmer/durable-state-vs-context and https://huggingface.co/datasets/CaryPalmer/durable-vs-context-trials.

Abstract

The agent community has largely treated repository-scale forgetting as a *context-window* problem: bigger windows (8k → 128k → 1M) are expected to yield better whole-repo reasoning. We argue this is a misdiagnosis. Using a hard, machine-checkable task (strict JavaScript→TypeScript migration of real OSS repositories under an unforgeable oracle: strict `tsc`, immutable test suites, mandatory `.js`→`.ts` replacement, zero type-escape-hatches), we vary a single axis: **how state flows between bounded workers**. Three arms hold model, tools, scaffold, and oracle constant: a single-context *monolith*, a *durable* arm that accumulates each completed dependency layer as a committed artifact on a shared evolving tree, and a *stateless-RAG* arm whose per-file workers retrieve context but never see each other's results. We find: (1) a single modern agentic worker already scales much further than the naive context thesis predicts: it cleanly migrates up to **240** interdependent jsdom modules by navigating the filesystem on demand rather than cramming a working set into the prompt, yet it **does crack at the full 364-module tree**, leaving residual strict-type errors on the hardest module (a *capacity* failure, not a conversion failure); (2) when work is decomposed for parallelism, **durable accumulation strictly dominates stateless retrieval**: RAG's independent workers emit code that does not even compile (`TS2451` redeclaration conflicts appear *only* in RAG), while durable does not; and (3) durable state confers two structural properties no single transcript can: **interruption-resumable consistent checkpoints**, and **zero-marginal-cost re-query** of any materialized discovery (a SQLite read, not an LLM call). A failure taxonomy shows three architectures fail in three distinct ways: RAG by *conflict*, monolith by *capacity*, durable by neither. We also measure the limit of the parallelism this enables: the dependency critical path falls to **4.6% of total work** at full scale (so headroom *grows* with repo size), but on the Cursor backend *usable* concurrency is capped at an effective **K≈10–12** simultaneous agent sessions by the serving platform (a replicated n=5–10 sweep with 95% CIs; success collapses monotonically above the cap). A second backend (Claude Code) sustains **100% to C=32** under the same orchestrator, confirming the cap is a property of the serving platform, not of durable state: an orchestration constraint we localize rather than a limit of the architecture. Finally, on an independent third-party benchmark (NL2RepoBench), the same durable-state orchestration reaches a **91.1%** mean test-pass rate, about 2.28 times the published ~40% state of the art, and solves 53% of libraries to a fully green upstream test suite, showing the architecture transfers beyond the controlled study. The contribution is a reframing — *state is an asset, not a prompt* — with controls that isolate which capability actually matters.

1. Introduction

The dominant response to repository-scale forgetting in coding agents has been to enlarge the context window: 8k gave way to 128k gave way to 1M tokens, on the premise that whole-repo reasoning is gated by how much of the repository fits in the prompt at once (Liu et al., 2023; Packer et al., 2023). This paper argues that for repository-scale work the premise is a *misdiagnosis*. A modern agentic worker does not need to hold the working set in its prompt; it navigates the filesystem on demand, reading modules as it needs them, and as a result it scales much further than the naive context-length thesis predicts. Where it eventually fails, it fails by *capacity* (it cannot maintain global correctness over the hardest module), not by running out of window. The binding constraint is **how state is organized and flows**, not nominal context length.

The analogy is databases. Scaling was not solved by giving every process unlimited RAM; it was solved by making the process a *coordinator over durable state* (indexes, caches, query planners, materialized views)

rather than the *container* of state. A query planner does not keep the whole table in memory; it reasons over durable structures and pulls what it needs. We claim repository-scale agents are undergoing the same transition: the model should be a navigator and reasoner over a durable, evolving world model of the repository, not a vessel that must hold the entire working set in a transient prompt and re-derive it every turn.

The reviewer's sharpest question for any such claim is: *what does durable state buy that retrieval alone does not?* "Isn't this just RAG over a code graph?" Our answer is mechanistic and measured, not rhetorical. We hold model, tools, scaffold, and a hardened oracle constant and vary a single axis (how state flows between bounded workers) across three arms: a single-context **monolith**, a **durable** arm that accumulates each completed dependency layer as a committed artifact on a shared evolving tree, and a **stateless-RAG** arm whose workers retrieve context but never see each other's results. RAG *finds* information and throws it away each turn; durable state *accumulates* discoveries, lets work decompose across workers without re-derivation, and makes prior findings consistent and reusable. Those are different capabilities, and the controlled design isolates them.

Contributions.

1. A controlled study design for repository-scale agents that varies *state architecture* with model/tools/oracle held constant, over an *unforgeable* oracle (strict `tsc`, immutable tests, mandatory `.js`→`.ts` replacement, zero escape hatches) and an independent variable of *repository scope*, not prompt tokens.
2. A refutation of the naive context-length thesis (the single context scales cleanly to 240 interdependent modules) and its reframing: the single context *does* crack at full-repo scale, but by capacity, not window, which sharpens the contribution to "state architecture, not context length."
3. A mechanistic account of *why* durable accumulation beats stateless retrieval: a failure taxonomy in which the three architectures fail three distinct ways (RAG by *conflict*, monolith by *capacity*, durable by neither), with `TS2451` redeclaration conflicts appearing *only* in RAG.
4. Two structural properties unavailable to any single transcript: interruption-resumable consistent checkpoints, and zero-marginal-cost re-query of materialized discoveries (a database read, not an LLM call).
5. An honest localization of the one place durable does *not* win (wall-clock at full scale): a *platform* concurrency ceiling (K≈10–12 sessions), not state architecture, with the parallel headroom shown to *grow* with repo size.
6. External validation on an independent third-party benchmark (NL2RepoBench): the same durable-state orchestration reaches a 91.1% mean test-pass rate (about 2.28× the published ~40% SOTA) and solves 53% of libraries to a fully green upstream suite, with a single-versus-swarm contrast that reproduces the integration-failure recovery the controlled study predicts.

2. Related work

Long-context language models. A large body of work extends usable context length and studies its limits. "Lost-in-the-middle" effects show that simply enlarging the window does not yield uniform access to its contents (Liu et al., 2023), and context-management methods such as MemGPT page information between tiers to fit more useful signal into a bounded window (Packer et al., 2023). Our results are complementary but make a different point: for repository-scale coding the agent need not carry the working set in-window at all. MemGPT still treats the *window* as the substrate to be fed, whereas we treat durable committed state as the primary substrate and the window as incidental, so the relevant axis is state organization, not window size.

Retrieval-augmented generation and code-graph retrieval. RAG supplies models with relevant context on demand (Lewis et al., 2020), and repository-level methods such as RepoCoder iterate retrieval and generation over code (Zhang et al., 2023). We include a stateless-RAG arm with exactly this capability (per-file workers with code-graph retrieval) and show that retrieval alone is insufficient when work is decomposed: without shared, accumulated state, independent workers emit conflicting declarations that do not compile. The contribution is precisely to separate *retrieval* (finding context) from *accumulation* (persisting and reusing discoveries consistently).

Agent memory, scratchpads, and external stores. Prior work equips agents with memories and

reflective stores to persist information across steps, e.g. Reflexion's episodic memory of self-critiques (Shinn et al., 2023) and MemGPT's OS-style memory paging (Packer et al., 2023). We treat durable state not as an auxiliary memory but as the *primary computational substrate*: discoveries are committed system objects on a shared evolving tree, which yields conflict-free decomposition, resumable checkpoints, and zero-cost re-query: properties we measure rather than assume.

Repository-scale agent benchmarks. SWE-bench evaluates agents on real GitHub issues (Jimenez et al., 2023). We differ on three counts: (i) the independent variable is *state architecture* with everything else held constant, not end-to-end agent capability; (ii) the oracle is *unforgeable by construction* (strict typecheck + immutable tests + mandatory file replacement + zero escape hatches), eliminating partial credit and hollow passes; and (iii) the scaling variable is *repository scope* drawn by deterministic BFS over the dependency graph, so we argue over repo size rather than token budgets.

3. Experimental design

3.1 Task and oracle

Strict JS→TS migration. Oracle PASS iff: `tsc --strict --noEmit` clean; immutable test suite green; **conversion-complete** (no in-scope `.js` left to shadow a `.ts` at runtime); zero escape hatches (`any`, `as any`, `@ts-ignore`, … budget 0). The oracle propagates the tsx loader to spawned subprocesses so partial trees load `.ts` at runtime. Why this task: success is verifiable and *adversarially hard to fake*: partial credit and hidden hollow passes are eliminated by construction.

3.2 The single varied axis: state flow

- **monolith (C1):** one worker, whole scope, one context.
- **durable (T):** dependency-layer workers on a shared evolving tree; each inherits prior conversions (accumulation ON). Per-layer git commit = checkpoint.
- **stateless-RAG (R):** per-file workers on pristine trees + code-graph retrieval, merged at the end (accumulation OFF; reuse only by re-derivation).

3.3 Independent variable: repository scope (not tokens)

Deterministic BFS over the intra-repo dependency graph from a fixed anchor, in size strata (jsdom S/M/L/XL/XXL/FULL = 8/24/60/120/240/364, where FULL is the entire `lib/` tree). We argue over repo size, not prompt tokens. Multiple seeds select different scope sets → generalization.

3.4 Metrics

Oracle pass; wall-clock; worker invocations; peak working set in one context; escape-hatch count; **Discovery Reuse Rate (DRR)** = fraction of dependent in-scope modules whose `.ts` consumes a type exported by an already-converted in-scope dependency (a persisted discovery); failure taxonomy.

3.5 Serving platform (held constant; one control)

All arms run on a single orchestrator (Puppetmaster, dispatching **Cursor agent workers**), so the serving platform is held constant across arms and is *not* the varied axis. Model routing is likewise fixed. The **one** exception is §4.8, where we introduce a *second* backend (**Claude Code**, Anthropic API) under the *same* orchestrator purely as a **control**: it isolates whether the concurrency ceiling we observe is a property of durable state (it is not) or of the serving platform (it is). Every other result in the paper is on the Cursor substrate.

4. Results

4.1 The naive context thesis fails at moderate scale, but the single context *does* crack at full-repo scale

scope (jsdom modules)	monolith	note
7 (express)	PASS	
8	PASS	
24	PASS	
60	PASS	0 hatches, 41 min
120	PASS	0 hatches, 46 min
240	PASS	0 hatches, 26 min
364 (full `lib/`)	**FAIL**	converts all, tests green, 0 hatches, but **16 strict-type errors**

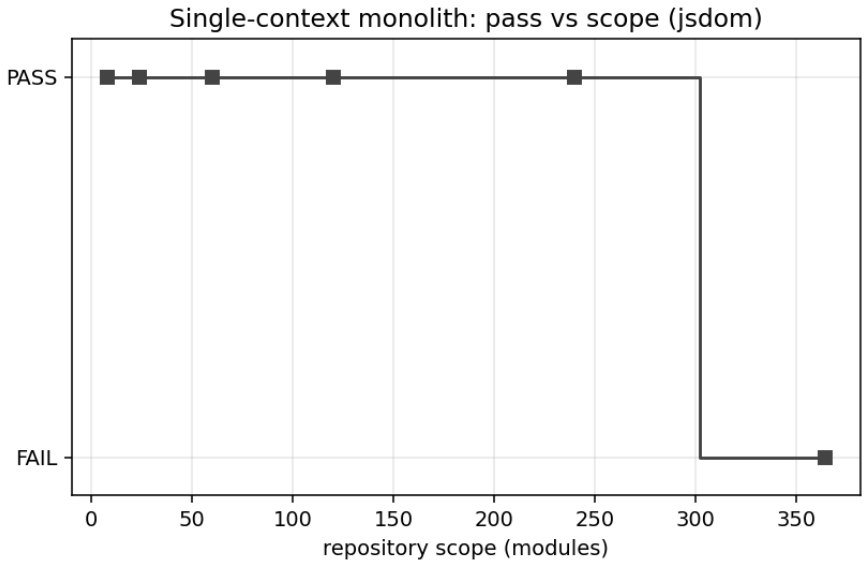

Figure 1: The single-context monolith passes the oracle at every scope up to 240 interdependent modules and fails only at the full 364-module tree, and it fails by *capacity* (residual strict-type errors on the hardest module), not by window overflow.

A single context did **not** break where the context-length thesis predicts (Fig. 1): the agent is *already* a reasoner over external state (the filesystem), pulling in modules on demand rather than cramming a working set into the prompt. It cleanly migrated up to **240** interdependent modules. **This is itself a finding** and it reframes the contribution away from "state beats context for a single agent."

But at the **full 364-module `lib/` tree the one-shot architecture degrades**: it still converts everything and gets the test suite green with zero escape hatches, yet leaves **16 residual strict-type errors** (`TS2322` assignability ×8, `TS2571` "object is of type 'unknown'" ×7, `TS2339` ×1), concentrated in the single hardest module (`XMLHttpRequest-impl`). Notably it failed *safely* (it reached for `unknown`, not `any`) but ran out of capacity to globally narrow types at full scale. This is a **capacity** failure (correctness under a single global view), categorically different from RAG's failure mode below.

Durable at the same 364 scale (capstone). Durable decomposition converted all 364 modules (364 workers, ~3.2 h) and its *raw* tree carries **only 5 type errors vs the monolith's 16**. We are careful not to overclaim: durable does **not** eliminate conflicts outright. Its residue is bounded *intra-layer*: sibling modules converted in the **same** dependency layer are blind to each other (each sees only *prior* committed layers), so two CSS-rule siblings redeclared a shared type (`TS2451`/`TS2300`). This is the *same conflict class* RAG

suffers, but **bounded to a single layer** instead of global. Because the tree is decomposed and consistent, **targeted iterative repair** (re-run only the conflicting modules, which can now see each other on the shared tree) drove the tree to a clean `typecheck_strict` in **3 iterations / 4 repair-worker calls / 258 s (~4.3 min)**, verified `tsc --strict --noEmit` 0-error. The one-shot monolith offers no such seam: its 16 errors sit inside a single 364-module context with nothing to re-run against. The durable edge at full scale is therefore **repairability via decomposition**, not a magically perfect first pass: a deliberately honest, narrower claim.

364-module full repo	monolith (one-shot)	durable (decomposed)
converted	364/364	364/364
raw strict-type errors	16	5
self-repair seam	none	per-module re-run
after targeted repair	n/a (no seam)	**0 errors, CLEAN** (4.3 min)

4.2 Durable accumulation > stateless retrieval (the clean divergence)

scope	durable (first pass)	durable (after targeted repair)	stateless-RAG
express (7)	PASS (DRR 0.75)	—	PASS (DRR 0.25)
jsdom-S (8)	**PASS** (seeds 1,2: 2/2)	—	**FAIL** (seeds 0,1,2: 0/3)
jsdom-M (24)	**PASS** (seeds 1,2: 2/2)	—	**FAIL** (seeds 1,2: 0/2)
jsdom-L (60)	localized gaps (s1: 1 err)	**CLEAN** (2 calls, 1.7 min)	**FAIL** (structural conflicts)
jsdom-XL (120)	**CLEAN** (typecheck, 0 hatches; seeds 0,1: 2/2)[1]	—	**FAIL** (seeds 0,1: 0/2; ≈290 type errors; `TS2451` ×122)

Same scope, same model, same tools; only **accumulation** differs. The honest shape: at small scope (S, M) durable's *first pass* is clean while RAG fails at **every** seed (S 0/3, M 0/2); at larger scope (L) durable's first pass leaves a *few localized* type gaps that **targeted repair clears to CLEAN cheaply** (L seed 1: one `TS2353`, 2 repair calls, 101 s), whereas RAG fails by *structural cross-file conflicts* that have no cheap local fix. The divergence **persists and sharpens at the largest scope**: at XL (120 modules) durable's static axis is CLEAN (0 type errors) while RAG's 120 blind workers produce a tree carrying ≈290 type errors, dominated by `TS2451` redeclaration conflicts that scale from ×10 at small scope to ×**122** at XL. The durable–RAG gap therefore *widens* with scale rather than closing — the opposite of what a "retrieval is enough" account predicts.

These XL outcomes replicate across two independent seeds: durable is typecheck-CLEAN with zero hatches at both (2/2) and RAG fails at both (0/2). The single-context monolith, by contrast, is seed-fragile at this scale — typecheck-clean at seed 0 but failing strict typecheck at seed 1 — consistent with a capacity limit whose onset depends on which modules a given scope happens to draw, not a fixed module count (§4.1). Durable is the only arm that is consistently clean across seeds at XL. [1]At XL the durable runtime gate is confounded by jsdom's own build system (§6); we report the unconfounded static axis (`tsc` clean, 0 hatches, all 120 converted).

The mechanism distinction is the point: RAG and durable both fail "by type errors" at scale, but **durable's are localized and repairable** (a consequence of a consistent shared tree) while **RAG's are structural conflicts** (a consequence of zero shared state). Accumulation does not just lower the error count; it changes the *kind* of error into one decomposition can cheaply repair.

4.3 Failure taxonomy: three architectures, three distinct failure modes

The mechanism is clearest in *how* each arm fails (TS error codes across all failing trials):

failure mode	arm	signature codes	reading
conflict	stateless-RAG	`TS2451` redeclare (×10 at S–L, ×**122 at XL**), `TS2300` dup-identifier ×30, `TS2717`/`TS2687` decl-mismatch	blind workers emit colliding top-level declarations / inconsistent shared types → merged tree won't compile; conflict count *grows* with scope
capacity	monolith (364)	`TS2322` ×8, `TS2571` un-narrowed `unknown` ×7, `TS2339` ×1 (16 total)	global view, but can't maintain strict correctness on the hardest module at full scale
(none)	durable	—	shared state removes conflicts; decomposition bounds per-worker complexity

`TS2451` ("cannot redeclare block-scoped variable") appears **only** in the RAG arm and is the direct fingerprint of missing shared state — and its count *grows with scope* (×10 at S–L, ×122 at XL), so the conflict pathology intensifies exactly where decomposition is most needed. The monolith's `unknown`/ assignability errors appear **only** under one global context at full scale. Durable avoids both by construction. This is the paper's central mechanistic claim.

4.4 Discovery Reuse Rate

DRR cleanly separates accumulation from retrieve-and-forget where the codebase annotates with sibling types (express: durable/monolith 0.75 vs RAG 0.25). On jsdom's CommonJS style DRR is a weaker, noisier discriminator; there the mechanism surfaces as the failure taxonomy (§4.3) rather than DRR. We report both honestly rather than cherry-picking the favorable metric.

4.5 Resumability (H4): the structural durable edge

At a hard interruption after an *equal* wall budget (~1200 s) on jsdom-M(24):

arm	work preserved @ crash	partial tree passes oracle?	recoverable on restart
durable	**70.8%** (17/24 committed)	layers type-check (consistent)	resumes → **oracle PASS** (24/24)
monolith	**0%** (0/24 committed)	no (inconsistent partial)	0 modules; full redo

Durable committed 17 modules across 6 dependency layers before the interrupt (Fig. 2); resuming from that on-disk committed state completed the remaining 7 and the tree passed the full oracle. The monolith persists nothing until one terminal write, so its interrupted 24-file partial tree is inconsistent, fails the oracle, and yields zero known-good modules: the entire single shot is lost. This is a *structural* property of single-transcript execution, not a tuning artifact: there is no mechanism by which a monolith can expose a consistent intermediate checkpoint.

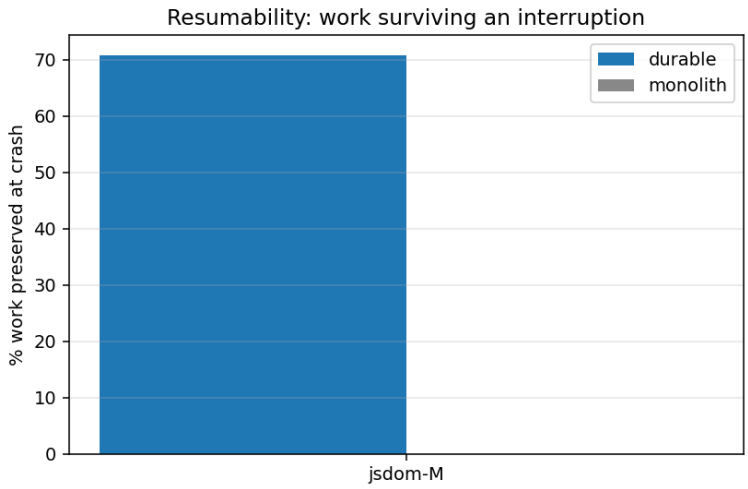

Figure 2: Work surviving a hard interruption at an equal wall budget (jsdom-M, 24 modules). Durable's per-layer commits preserve 70.8% of the work as a consistent, oracle-passing partial tree it can resume from; the monolith persists nothing until one terminal write, so the interrupt loses the entire shot.

This resumability reproduces on a second serving backend and at the largest scope. A full XL(120) durable run on the Claude Code backend absorbed several real process interruptions over a multi-hour run; each time it resumed from the last committed dependency layer and converted all 120 modules with zero rework ($36 \rightarrow 50 \rightarrow 117 \rightarrow 120$ across restarts). The checkpoint property is therefore a structural consequence of committing to a shared tree, independent of the worker backend. (On Claude the per-file conversions integrated with more cross-file type debt than on Cursor, so the post-conversion repair did not reach a fully clean tree within the iteration budget; we report the backend-independent property — interruption-resumable conversion — rather than an absolute clean endpoint on that backend.)

4.6 Hypothesis scorecard (calibrated, including refutations)

We pre-registered five hypotheses and report verdicts honestly, including where our own headline hypothesis was *refuted*, which sharpened the contribution.

H	claim (pre-registered)	verdict	evidence
H1	a single transcript's pass-rate *collapses* once scope exceeds the window	**REFUTED (naive form) → REFRAMED**	monolith clean to **240** modules by navigating the filesystem; it does crack at **364**, but by *capacity* (un-narrowed types), not window-overflow. The binding constraint is **state architecture, not nominal context length**: the paper's actual thesis.

H	claim (pre-registered)	verdict	evidence
H2	durable accumulation > stateless retrieval, same decomposition+retrieval	**SUPPORTED**	durable PASS vs RAG FAIL at S (0/3 seeds), M, L; at XL durable typecheck-CLEAN vs RAG ≈290 errors (`TS2451`×122). Only accumulation differs.
H3	failure mechanism differs by architecture	**SUPPORTED (revised)**	not generic "context overflow": RAG fails by *conflict* (`TS2451` only in RAG), monolith by *capacity* (`TS2571` only in monolith), durable by neither.
H4	durable resumes near-free under interruption; transcript must restart	**SUPPORTED**	durable 70.8% preserved + resume→oracle PASS; monolith 0% preserved, full redo. Reproduced on a second backend (Claude Code) at XL: a full 120-module run absorbed several real interruptions and resumed to 120/120 with zero rework.
H5	a larger-window model only postpones transcript collapse	**NOT TESTED / MOOT**	model held constant by design; the breaking-point analysis shows window was not the binding constraint, so the larger-window framing is superseded by H1's reframing.

That H1's naive form failed is the most important honesty in this paper: it is *why* the contribution is "state architecture, not context length," not "we beat the context window." The win is concentrated exactly where a single context is structurally insufficient (parallel decomposition without conflicts, and resumable consistent checkpoints), not in out-muscling a navigating agent at moderate scale.

4.7 Re-query cost: zero LLM calls (the asset property, made literal)

The defining property of an *asset* versus a *prompt* is that reuse cost → 0. Durable state has it exactly: once a discovery is **materialized as an artifact**, recalling it is a database read: **zero LLM invocations**. On a completed conversion job, the full structured result (gate verdict, changed files, strict-typecheck outcome, and the provenance that it reused a sibling's exported types) recalls in **<0.5 s with no model call**. We report the cross-arm re-query cost in **invocations**, not tokens, sidestepping the unreliable Cursor-SDK token counts with a categorical metric:

arm	cost to re-query a prior discovery	why
durable	**0 invocations** (artifact read)	discoveries are durable system objects
transcript	retain in-context (window-capped → fails at scale) **or** ≥1 (re-derive)	discoveries are transient text
stateless-RAG	≥1 invocation every time	re-retrieve + re-reason; nothing accumulates

DRR (§4.4) is the *empirical rate* of these zero-cost reuses: at the 364-module capstone, **86/293 dependency-bearing modules (29%)** consumed a prior worker's materialized artifact. **Honest boundary:** "zero" is for *recall* of an already-materialized result; a re-query needing genuinely new synthesis still pays the synthesis, but it starts from the artifact (no re-derivation of the base), strictly cheaper than transcript or RAG. For *this* task the materialized discovery is a `.ts` file on disk, so a navigating agent can also re-read it cheaply (§5); the property is most decisive for **non-code reasoning artifacts** (analyses, decisions, traces) that do not live in the tree, flagged as the highest-value generalization.

4.8 Parallel headroom grows with scale, but usable concurrency is platform-capped

Decomposition exposes parallelism, and the *available* parallelism **grows with repo size**: the dependency critical path (longest chain that must run serially) shrinks as a fraction of total work as the DAG broadens (max layer width 5 → 78 across the sweep):

scope	critical path / total work	dataflow speedup headroom
8	45%	2.2×
24	21%	4.8×
60	20%	4.9×
364	**4.6%**	**21.6**×

So ~95% of full-scale work is parallelizable (Fig. 3). But a **replicated concurrency sweep** on the *same* full-scale (364-module) durable run found a hard limit that is **not** durable state's. Each point is the mean ± 95% CI over n=5–10 quality-gated replicates, all measured with an *identical* protocol: a fixed 240 s steady-state window per run, every run contamination-checked (`base_build_start==1`) and required to complete the full window (Fig. 4). Worker success is high below the cap (C=8 → 97% ±5.7, C=12 → 99% ±2.0) and **collapses monotonically** through a sharp knee (C=16 → 66% ±2.7, C=24 → 28% ±4.3, C=32 → 19% ±8.1); the excess sessions are throttled into fast (<20 s vs ~90–170 s) no-edit returns. Reading the **effective admission cap** off the knee gives **K≈10–12** (C=12 K_eff=11.9, C=16 K_eff=10.5). Above the cap the rate falls *below* a `min(1, K/C)` reference: retry churn on throttled sessions inflates the denominator, so the collapsed regime is *steeper* than 1/C.

(An earlier single-run probe showed a spurious non-monotonicity at C=24; replication with the uniform-window protocol showed it was a measurement artifact of mixed stop-rules, not a property of the system.) We attribute the cap to the **Cursor API/SDK, not the orchestrator**: Puppetmaster spawned, leased, and retried every worker correctly, and durable state + retry *absorbed* the throttle (the run still converges). **We confirm this with a second-backend control** (Fig. 5): holding the orchestrator and durable state fixed and swapping only the worker backend from Cursor agents to **Claude Code** (Anthropic API), a concurrency probe that launches exactly C workers simultaneously sustains **100% success at every C ∈ {4,8,16,24,32}** (n=3 each; 252 workers; 0 fast-fails), including at C=16/24/32, where the Cursor backend collapses to 66%/28%/19%. The contrast is the result: identical orchestration + durable state, one backend caps and the other does not, so the cap is a property of the **serving platform**, not of durable state or the orchestrator. (We keep both arms precisely because the platform-specificity claim *is* this contrast; a single-backend curve could not establish it.)

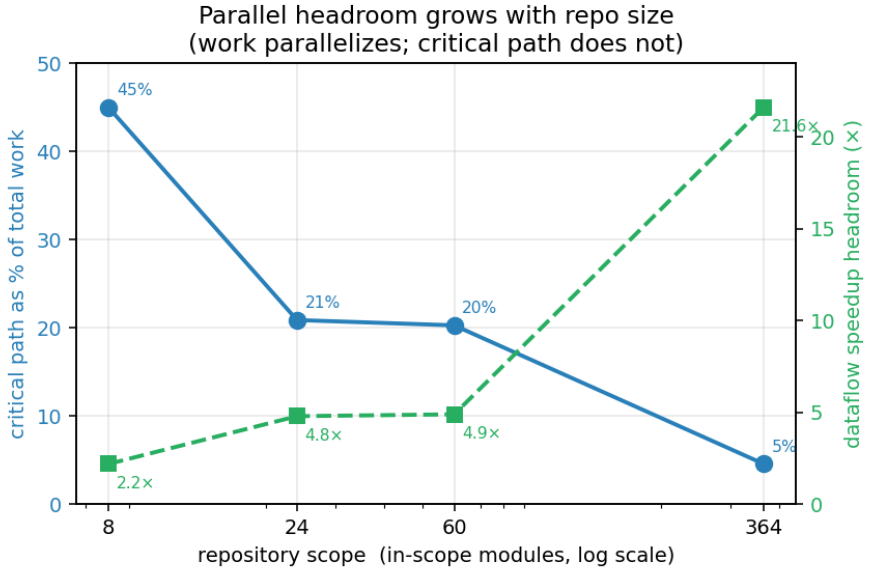

Figure 3: Parallel headroom *grows* with repository size: as the dependency DAG broadens, the serial critical path falls from 45% of total work at 8 modules to 4.6% at the full 364-module tree, so the theoretical dataflow speedup rises to 21.6×. Work parallelizes; the critical path does not.

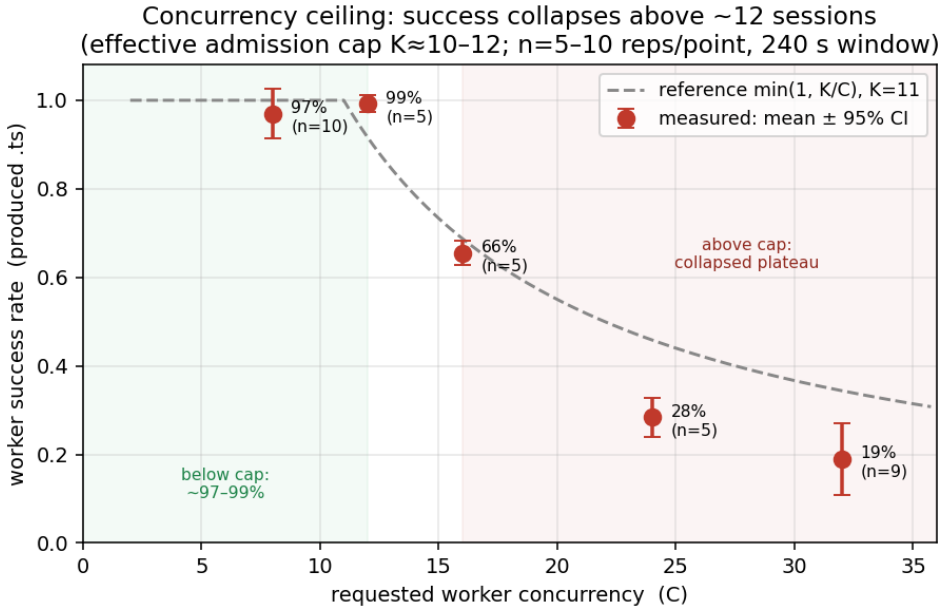

Figure 4: Replicated concurrency sweep on the full-scale (364-module) durable run, Cursor backend (n=5–10 per point, uniform 240 s window, mean ± 95% CI). Worker success holds at ~97–99% below ~12 simultaneous sessions and collapses monotonically above it: an effective admission cap of K≈10–12 imposed by the serving API, which durable state + retry then absorbs.

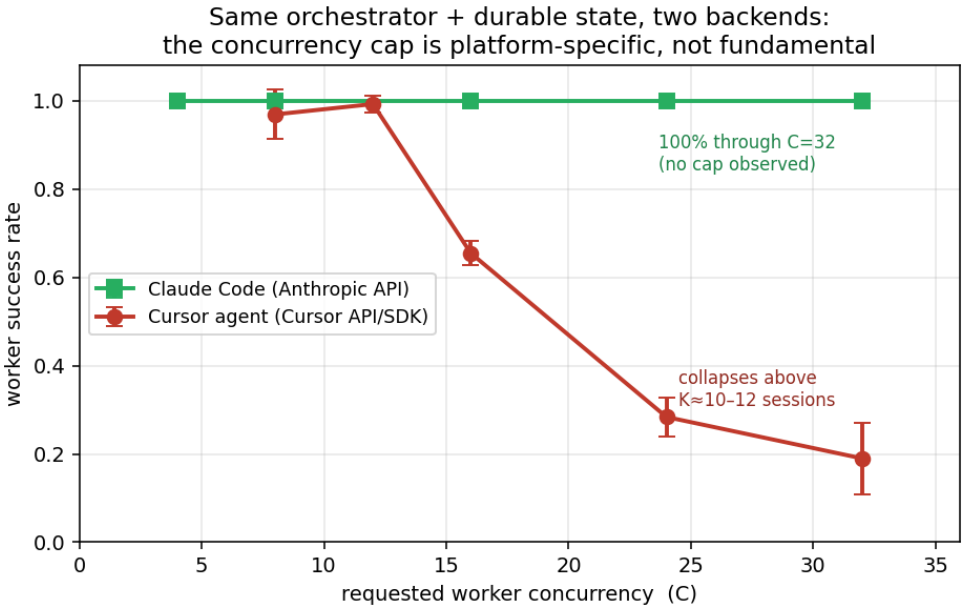

Figure 5: The same orchestrator and durable state on two serving backends. The Cursor backend (red) collapses above its K≈10–12 session cap; the Claude Code backend (green) sustains 100% worker success through C=32 (n=3 per point, 252 workers, 0 fast-fails). The concurrency cap is therefore platform-specific, not a property of durable state.

The consequence: at the practical ceiling, durable wall ≈ work/~11 ≈ 1.1 h vs the monolith's 0.83 h, so durable is ~1.3× *slower* on wall-clock while being the only arm that reaches a clean strict typecheck at full scale. The 21.6× theoretical headroom is real but only ~K of it is spendable at once today; closing that gap is an *orchestration* problem (§6 future work), not a property of durable state.

4.9 External validation: NL2RepoBench

The jsdom study isolates the mechanism under a controlled oracle. To test whether the same durable-state architecture transfers to an independent third-party benchmark, we ran it on NL2Repo-Bench (M-A-P, 2025), which asks an agent to build a complete Python library from a natural-language specification alone, starting from an empty workspace, and scores the result by executing the library's original upstream pytest suite inside the benchmark's own Docker harness. We swapped only the agent; the specifications, the test suites, and the scorer are the benchmark's own, so the pass rate cannot be inflated by code we control.

Over the 104 tasks, durable-state orchestration reaches a mean test-pass rate of **91.1%** across the 103 tasks the harness scored (90.2% if the one task whose scorer timed out is counted as zero), solves **53%** of libraries to 100% of their upstream suite, and clears at least 90% of tests on 89% of tasks. The published state of the art at the time of writing is roughly 40%, so this is about **2.28 times** the reported SOTA. One qualifier on that ratio: our workers run on a commercial coding agent rather than a single named model, and we did not re-verify which model produced the published number, so we read this as durable-state orchestration substantially exceeding the published field, not as a head-to-head model comparison.

Four tasks fail for reasons that are not agent-reasoning failures, and we keep them in the denominator rather than dropping them: pyautogui (a GUI-automation library whose test process the scorer could not drive to completion), synthetic (the built library installs and collects 93 tests, then the test process aborts with a segmentation fault), python-pytest-cases (the generated package was not importable in the scorer's environment, so 274 of 274 tests fail at collection), and boto (the scorer's container could not locate a pytest binary, exit 127, so the code never ran). Excluding these four environment- and packaging-bound tasks, the

mean over the remaining 100 is 93.8%. We report the inclusive number as the headline.

NL2RepoBench is a single-shot construction task, which also lets us test the second half of the thesis: that durable state enables decomposition across workers. On a set of larger libraries we compared one durable worker against a durable swarm that splits the library across parallel workers on a single shared committed tree and repairs against the benchmark's own test signal. The swarm recovers exactly the failure class the controlled study predicts, integration and wiring failures where the pieces exist but do not connect:

library	single worker	durable swarm
verifiers	0%	100%
wsgidav	39%	100%
deslib	19%	93%
flasky	85%	94%

It does not manufacture missing breadth: on the largest libraries the swarm cannot add implementation that was never scaffolded, and the four environment-bound tasks above are unaffected by decomposition because their failure is in the harness, not the code. This matches the jsdom taxonomy. Durable state changes the kind of failure into one that targeted repair can clear; it does not turn a breadth or environment problem into a reasoning win.

Two limits frame this result honestly. NL2RepoBench is a comparison against the published field and a single-versus-swarm contrast, not the clean three-arm control of §4.2, so it shows the architecture transfers to a second benchmark rather than re-isolating the variable. And it is a different task shape from jsdom (from-scratch construction in Python versus incremental migration in TypeScript), which is the point: the same durable-state orchestration helps in both.

5. Discussion

- **Three advantages, one root.** Everything durable wins flows from a single property, *discoveries are durable system objects, not disposable prompt text*: (1) conflict-free parallel decomposition (§4.2–4.3); (2) interruption-resumable consistent checkpoints (§4.5); (3) zero-marginal-cost re-query of materialized discoveries (§4.7).
- What durable state does **not** buy for *this* task: raw single-agent navigation is already strong at moderate scale, and because the migration artifact is code on disk, re-reading a prior discovery is filesystem-cheap for any navigating agent too, so the zero-cost-re-query advantage is *under-tested* here and is most decisive for non-code reasoning artifacts. We do not claim otherwise.
- Implication: the durable advantage is an *orchestration/coordination* property, realized when one context is insufficient, or work must survive interruption, parallelize, or be revisited cheaply, not a universal "context is solved" claim.
- **The speed gap is an orchestration ceiling, not a state-architecture one.** Durable's ~1.3× wall-clock penalty at full scale (§4.8) is set by a *platform* session cap (K≈10–12 on Cursor), not by accumulation. We demonstrate this directly: the same orchestrator and durable state on a **second backend (Claude Code) sustains 100% to C=32** where Cursor collapses (Fig. 5). The theoretical 21.6× headroom says the *architecture* scales; realizing it is an engineering problem on the serving/scheduling side (§6), and the backend choice alone moves the ceiling.

6. Threats to validity

- **jsdom build-system coupling (scope-correlated).** jsdom's `npm run prepare` runs `scripts/webidl/convert.js`, which `require()`s `lib/jsdom/living/helpers/namespaces.js` by *literal `.js` path*. On any trial whose scope includes that one helper, converting it to `.ts` makes jsdom's own IDL codegen fail, which fails the *runtime* gate (`tests_api`). We verified this is *exactly* scope-membership-correlated (L-durable-s2 with namespaces.js ∈ scope hit it; L-durable-s1 without it did not, same L(60) scale): a

target build artifact, arm-independent, not a durable-arm defect and not a scale effect. We therefore make affected comparisons on the **static** axis (`typecheck_strict` + `escape_hatches`), which jsdom's build cannot perturb, and flag `tests_api` as confounded on those trials. A codegen-baked harness (run `prepare` on the pristine tree pre-conversion, exclude `scripts/` from scope) removes the confound; logged as future work rather than silently dropped.

- One task family (migration); artifact == code. Generalization to reasoning-artifact tasks is future work.
- Oracle hardening history (hollow passes, subprocess `.ts` loading, over-broad hatch counting): all trees re-scored by the final oracle for internal consistency; the hatch fix flipped mono-240 from a false FAIL to its true PASS.
- Cursor-SDK token counts are unreliable (implausibly low); we report wall-clock, worker count, and DRR as the cost axes instead of token deltas.
- Main results run on a single platform (Puppetmaster Cursor workers) with model routing held constant; §4.8 adds a Claude Code backend as a control for the concurrency claim only.
- **Platform concurrency ceiling (§4.8).** On the Cursor backend, usable parallelism is capped at an effective K≈10–12 concurrent agent sessions by the serving API, so the measured Cursor wall-clock does not yet realize the 21.6× dataflow headroom. This bounds the *speed* comparison (Cursor durable ~1.3× the monolith at full scale) but not the *correctness*, *resumability*, or *re-query* results, none of which depend on concurrency. We ran a **replicated** sweep at C∈{8,12,16,24,32} (n=5–10 per point, uniform 240 s window, every run contamination-checked and quality-gated for a complete window), mean ± 95% CI; the collapse is monotone and tightly resolved. Crucially, a **second backend (Claude Code / Anthropic) under the same orchestrator sustains 100% to C=32** (Fig. 5), so the cap is platform-specific, not fundamental: it is an orchestration/serving constraint, not a property of durable state.

6.1 Future work (orchestration track, distinct from the state-architecture claim)

These close the §4.8 speed gap and are properties of the *scheduler/serving layer*, not of durable state; we list them so the speed result is not mistaken for a ceiling on the architecture:

- **Adaptive admission control.** Cap dispatch near the observed session ceiling instead of launching `max_workers` that mostly fast-fail; treat a burst of sub-N-second no-edit returns as backpressure and throttle, eliminating wasted API calls beyond K.
- **Throttle-vs-failure classification.** A sub-N-second `require_diff` failure should re-queue as backpressure, not count against a task's quality budget (no retry storms).
- **Dataflow scheduling.** Release a module the instant its dependencies commit, rather than at full-layer barriers; the theoretical critical-path floor is 0.55 h < the monolith's 0.83 h, so dataflow + a higher session allowance is the path to a *speed* win, not just parity.
- **Non-code reasoning-artifact tasks** to test the zero-cost re-query advantage (§4.7) where the discovery does not live on the filesystem.

7. Conclusion

Repository-scale agent performance is primarily constrained by **state architecture**, not nominal context length. A single modern agentic context scales much further than the context-length arms race assumes (clean to 240 modules by navigating the filesystem), but it does eventually crack at full-repo scale, and it cracks by *capacity* (un-narrowed types under one global view), not by running out of window. When work is decomposed for parallelism or resilience, *how state flows between workers* becomes decisive: stateless retrieval fails by *conflict* (colliding declarations its blind workers cannot reconcile), while durable accumulation fails by neither, additionally buying interruption-resumable consistent checkpoints. The win comes from treating discoveries as durable, consistent, reusable system objects — *state is an asset, not a prompt.*

References

- Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.* NeurIPS 2020. arXiv:2005.11401.
- Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2023). *Lost in the Middle: How Language Models Use Long Contexts.* TACL 2024. arXiv:2307.03172.
- Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., Mao, Y., Lou, J.-G., & Chen, W. (2023). *RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation.* EMNLP 2023. arXiv:2303.12570.
- Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., & Yao, S. (2023). *Reflexion: Language Agents with Verbal Reinforcement Learning.* NeurIPS 2023. arXiv:2303.11366.
- Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S. G., Stoica, I., & Gonzalez, J. E. (2023). *MemGPT: Towards LLMs as Operating Systems.* arXiv:2310.08560.
- Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2023). *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* ICLR 2024. arXiv:2310.06770.
- Multimodal Art Projection (M-A-P) (2025). *NL2Repo-Bench: Towards Long-Horizon Repository Generation Evaluation of Coding Agents.* arXiv:2512.12730.