GROOM: Gated Refresh of Organizational Memory

Gui Dávid

BeConfident Labs

beconfident.app

Abstract

Large language model agents increasingly ground their behavior in curated text corpora—internal wikis, convention documents, and retrieval indices. These corpora *rot*: the field moves on, the text does not, and every agent that reads a stale page is silently degraded. We make the cost concrete: when a consuming agent treats such a corpus as authoritative, injecting staleness into five facts drops its answer accuracy on those facts from 100% to 0%—a clean propagation, since the probes are deterministic-recall—while unaffected controls hold at 100%. Work on context engineering manages the *window*—what reaches the model at inference time—but not the *source*, and maintaining the corpus itself stays a manual, easily neglected chore.

We present GROOM: consultation-triggered, checkpointed maintenance for agent knowledge bases. The refresh is triggered by consumption itself: when an agent reads the corpus through a skill, a gated launcher returns in tens of milliseconds, and when a refresh is due it spawns a *detached* maintenance agent that runs one bounded operation—lint, prune, expand, research, or iterate. The read never blocks, and the next reader gets the benefit. The refresh is *gated* because autonomous edits to a live corpus are dangerous: every operation is wrapped in a git checkpoint behind a deterministic, token-free validator, and counts only if it reports terminal success, passes structural *and* fact-level validation, satisfies its postcondition, and keeps its edits inside the corpus; otherwise we reset the working tree to the pre-operation commit.

We inject nine fault classes into a real 19-page knowledge base, from link-breaking deletions to the subtler removal of a load-bearing fact that passes every structural check. The gate rejects all nine and restores the corpus byte-identically to the checkpoint (n=450) in tens of milliseconds; a naive maintainer that commits unconditionally corrupts the corpus in all nine. The read path costs a warm median of 50 ms, within a 100 ms budget, and the validation gate scales linearly ($R^2 \geq 0.97$; tens of microseconds per page, 14–27 ms at 400 pages). Profiling the concurrency guard exposed a check-then-write race in the original debounce stamp: it resolves eight simultaneous triggers to a single run only 28–59% of the time, which an atomic claim repairs to 500/500. The benefit generalizes: across three unrelated corpora and two retrievers (BM25 and a dense neural model), removing the structural entropy grooming targets lifts recall@1 by a 45–51% relative gain—a corpus-state comparison, not the operations run live.

We are explicit about what the gate does *not* certify—knowledge correctness beyond curated facts—and a live end-to-end measurement remains the central open problem; in simulation, consumption-triggered maintenance beats an equal-cost schedule under the skewed access patterns real corpora exhibit. GROOM is open source and ships as a content-agnostic pipeline; its first instance maintains a knowledge base on harness engineering for AI agents. The pipeline is also format-agnostic: it reads, validates, and safely maintains bundles in the Open Knowledge Format (OKF), the vendor-neutral standard for agent knowledge bases, and we verify the gate recovers byte-identically on an OKF bundle and that its OKF validator accepts Google's published OKF bundles unchanged.

1 Introduction

An LLM agent is only as current as the text it reads. Production agents are routinely grounded in curated corpora: engineering wikis, style and convention guides, runbooks, research digests, and the document stores behind retrieval-augmented generation [5]. Model weights are versioned and deliberately updated; these corpora drift out of date by default. A page written in March describes a framework that shipped a breaking change in May. An agent that reads it in June inherits the error and repeats it confidently. The failure is silent, so nobody notices until a downstream artifact is wrong. We call this *knowledge rot*. It is distinct from the in-window degradation—"context rot"—that the context-engineering literature addresses [1]. Agents act through reasoning-and-acting loops [10], and any step may consult the corpus, so rot in the corpus is rot in every grounded decision. Context engineering decides *which* tokens reach the model and in what form; it says nothing about whether the underlying source is true today.

The obvious remedy is to schedule a maintenance job. It fails in practice for the same reason documentation rots in software: maintenance that is nobody's immediate concern is deferred forever. But an agentic knowledge base offers a trigger that a static document store does not. *Agents read it programmatically, at a cadence that tracks how relevant it is.* The corpus consulted most often is the one whose freshness matters most. So we amortize maintenance over consumption, in direct analogy to the `stale-while-revalidate` cache-control directive [6]: serve the cached value immediately, and refresh it in the background for the next request. The analogy has a deliberate limit. An HTTP cache can watch its upstream change; GROOM cannot watch the world change, so it cannot invalidate on source change, and consumption frequency is its only freshness signal. The limit is also a feature: the page read most often, and therefore most likely to matter, is the one maintained most aggressively.

Thesis. Make *consulting* the knowledge base the act that *maintains* it. Reads pay no latency for freshness; maintenance runs detached, after the read returns, for the benefit of the next reader.

Building this thesis into a deployable system raises two problems that occupy the rest of the paper. First, *when* should a read trigger maintenance? Triggering on every read is wasteful and races; never triggering is the status quo. We answer with a gated launcher that returns in under 100 ms whatever it decides and serializes concurrent triggers with an atomic claim (Section 3.2). Second, and more seriously, *autonomous edits to a live corpus are dangerous*. A maintenance agent that mis-prunes can delete a load-bearing fact, and an agent that ingests web content opens a prompt-injection channel into every future reader. We make autonomy safe with a git checkpoint around a deterministic validation gate: any operation that fails, exceeds its turn budget, or tries to write outside the corpus is reset to a known-good state instead of committed (Section 3.4).

Contributions.

1. **Consumption-triggered maintenance** for agent knowledge bases. Reading the corpus, through a lazily-loaded skill, fires a gated launcher that detaches maintenance from the read path. This stale-while-revalidate discipline amortizes freshness over reads at no read-time latency (Section 3.2).

2. **Checkpointed maintenance with an enforced gate**: a git-commit / validate / commit-or-revert wrapper that moves the corpus's invariants out of prompts and into code. It turns mis-edits, crashes, turn-truncation, and fence escapes from permanent corruption into recoverable no-ops. We extend the gate from structural checks to fact-level *canaries* that catch the semantic-loss case structural checks miss (Section 3.4).

3. **An evaluation** on a real 19-page knowledge base. It (i) shows corpus staleness collapses a consuming agent's answer accuracy from 100% to 0% on the affected facts while controls hold; (ii) injects nine fault classes and finds the gate rejects and recovers all of them byte-identically (n=450), where a no-gate baseline corrupts the corpus in all nine; (iii) profiles the concurrency guard, exposing a time-of-check/ time-of-use race in the debounce stamp—exactly one run in only 28–59% of eight-way races—that an atomic claim repairs to 500/500; (iv) ablates the validator,

showing structural checks catch 0/5 semantic-loss injections while fact-level canaries catch 5/5; and (v) shows the benefit generalizes across three unrelated corpora and two retrievers (BM25 and dense), where removing the structural entropy grooming targets yields a 45–51% relative recall@1 gain (a corpus-state comparison; running the operations live is future work). We release an open-source, content-agnostic, retrieval-agnostic, and format-agnostic implementation (it maintains its native typed corpus or, optionally, bundles in the Open Knowledge Format), whose operations are discovered from prompt files and whose capabilities are schema-scoped per operation, with the full benchmark harness (Section 5).

We are deliberate about scope. GROOM provides safety and recoverability, not a proof of knowledge correctness. Sections 5.6 and 6 draw that boundary explicitly.

2 Background and Related Work

Context engineering and agent memory. The dominant framing for "what the model knows at inference" is context engineering [1]: curating the smallest high-signal token set per call through retrieval, compaction, and note-taking. Memory systems such as MemGPT [8] extend this with an OS-inspired hierarchy (core, archival, recall) that pages information in and out of the window. These techniques govern the *flow* of information into the model. GROOM is complementary and orthogonal: it governs the *freshness of the source* those techniques draw from. A perfectly tuned retrieval stack over a stale corpus still grounds the agent in stale facts. Such a shared corpus is, in organizational terms, a team's *transactive memory* [22]—the external store a group relies on instead of each member holding everything—and GROOM is the discipline that keeps it current.

Retrieval freshness. RAG pipelines [5] treat staleness as an indexing problem, usually solved by re-embedding on a schedule or on source change. GROOM differs in two ways. Its corpus is authored for agent consumption, so each page carries a machine-read `summary`, `updated` date, and `confidence` grade. And its refresh is *semantic*: a maintenance agent rewrites, merges, and prunes prose rather than merely re-indexing it, which is why it needs the safety machinery of Section 3.4. Benchmarks of LLM staleness [19] and adaptive-retrieval methods that decide *when* to fetch [20, 15] address freshness at query time; GROOM addresses it at the source, between queries.

Self-improving harnesses. The closest conceptual relative is recent work on harnesses that improve themselves [11], which mines an agent's execution traces for failure patterns and proposes harness modifications gated by regression testing. GROOM applies the same design rule—*put the maintenance loop inside the artifact it maintains, gated by checks the artifact defines*—at the knowledge layer rather than the policy layer. We are careful not to blur one distinction. Self-Harness's regression benchmark certifies that a change *improves task outcomes*; GROOM's gate certifies only that a change *did not break structure or a curated fact set*. So we claim safety and recoverability, not correctness, and treat a live outcome benchmark as the main remaining problem (Section 5.7 takes a first, simulated step).

Documentation as code, and CI. Treating prose as a versioned, tested artifact is established practice in software: docs-as-code, link checkers, CI gates. GROOM adopts this stance for an *agent-consumed* corpus and adds two twists. The editor is itself an LLM agent, and the trigger is consumption rather than a commit hook. The checkpoint mechanism (Section 3.4) is a direct analogue of a CI gate, with `git reset` standing in for a failed merge.

What GROOM is not. Six neighbors are close enough to warrant explicit distinction. *Model knowledge editing* [7, 17, 18] (e.g. ROME, MEMIT) repairs facts inside frozen weights and must be re-applied per model. GROOM edits the *shared, model-agnostic source* that every agent and every model version reads, so its corrections survive a model upgrade instead of being invalidated by it. *Truth-maintenance and belief revision* [4] revise *logical* knowledge bases over explicit justification

links. GROOM targets *natural-language* corpora, where consistency is unprovable: it substitutes a deterministic structural gate plus fact-level canaries for logical entailment, and amortizes revision over consumption rather than triggering it on assertion. *Agentic self-refinement* [9, 16] (Reflexion, Self-Refine) iterates a model's *transient* output within a task. GROOM iterates a *persistent, shared* corpus across tasks and readers, and replaces the model's own self-critique with an out-of-model deterministic gate—the critic is code, not the generator. *Persistent-memory agents* [13, 14, 8] (Generative Agents, Voyager, MemGPT) keep a self-edited store across episodes: a reflective memory stream, a growing skill library, paged context. But that store is each agent's *private* memory, edited by the very model that consumes it. GROOM maintains a *shared, model-agnostic* corpus read by many agents, gated by an *out-of-model* critic rather than the generator's own judgment. The closest *artifact* is the recently popularized *LLM-maintained markdown wiki* [12]—a git-versioned corpus an agent grows and health-checks through ingest/query/lint operations over an indexed page set. GROOM shares that substrate but adds what the pattern omits for safe autonomous operation: consumption as the maintenance *trigger*, a deterministic out-of-model checkpoint gate (commit-or-revert) around every edit, and fact-level canaries—the pattern itself carries no write-gating, concurrency control, or semantic-loss guard. Finally, wiki quality bots [21] (ORES, ClueBot) revert *human* edits post-hoc. GROOM gates its *own autonomous agent's* edits pre-commit, and fires on *reading* rather than on an external edit stream. The novelty is the cell none of these occupy: a model-agnostic, persistent, natural-language source that self-maintains as a side effect of being consumed, with correctness deferred to an out-of-model gate.

Open knowledge formats. Concurrent with this work, Google released the *Open Knowledge Format* (OKF) [23], a vendor-neutral standard that represents an agent knowledge base as a directory of markdown files with YAML frontmatter—the same substrate GROOM maintains, drawing on the same LLM-maintained-wiki lineage [12]. OKF standardizes the *format* and is explicit that serving, storage, and query are non-goals; like the LLM-maintained-wiki pattern it descends from, it specifies no maintenance discipline—exactly the cell GROOM occupies. The two compose: OKF is the interchange format, GROOM the maintenance runtime. The same gate validates and safely maintains OKF bundles, which we measure directly (Section 5.9).

Production agent systems. Analyses of deployed coding agents [3, 2] report that most of an agent system is operational infrastructure—loop, context management, tools, permissions, persistence—rather than model-facing logic, and that safety comes from layered, deny-first control and schema-level tool restriction. GROOM's capability scoping and path confinement (Section 4) follow this doctrine; our maintenance agent is itself a small instance of it.

3 The GROOM Architecture

GROOM has four parts: a typed corpus, a consumption-triggered launcher, a set of bounded maintenance operations, and a checkpointed executor. Figure 1 shows how they compose into a loop where reading and maintaining are one motion.

3.1 The corpus is typed for machine consumption

The knowledge base is a directory of Markdown pages, one topic per page. Each page is prefixed with YAML frontmatter carrying a one-line `summary` (what a consuming agent reads to decide whether to load the page), an `updated` date, and a `confidence` grade in {*established*, *emerging*, *contested*}. The `summary` fields make progressive disclosure cheap: a consuming agent reads the index, picks one to four pages whose summaries match its question, and loads only those. The `confidence` grade tells the consumer how hard to lean on a claim—state *established* plainly, hedge *emerging*, present *contested* as a disagreement. This typing lets the corpus be both a human-readable document and a structured object that the validator (Section 3.4) and the maintenance operations can reason over.

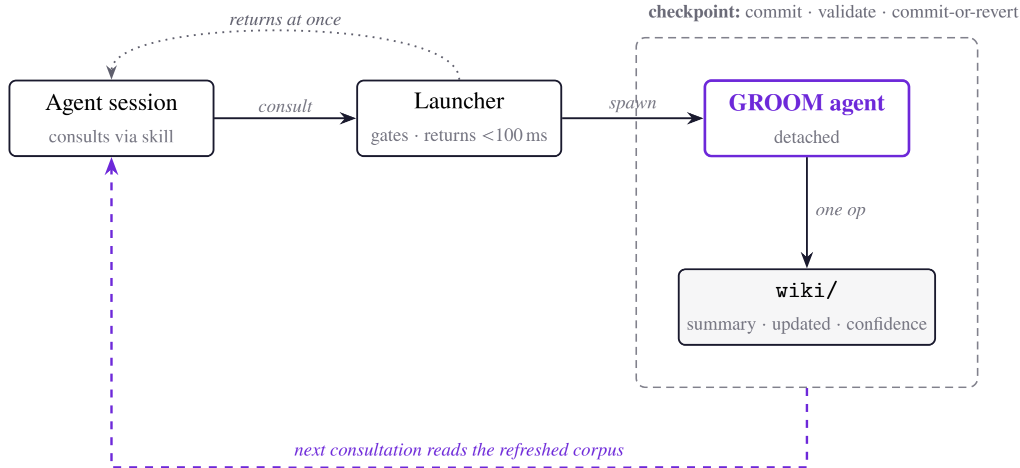

Figure 1. GROOM as a loop in which reading and maintaining are one motion. A consultation enters through a skill; the launcher gates and returns in under 100 ms (dotted), while a detached GROOM agent performs one bounded operation over the typed corpus. The edit is wrapped by a checkpoint that commits, validates, and either commits or reverts (Section 3.4). The triggering read never waits: the refreshed corpus is observed by the *next* consultation (dashed).

3.2 Consultation is the maintenance trigger

GROOM ships as a *skill*: a lazily-loaded instruction file that costs roughly one line of context until a relevant question arises. At that point it tells the consuming agent how to read the corpus and, as its first step, to fire the maintenance launcher. Figure 2 traces a single consultation.

The launcher is the crux of the latency story. It performs three gate checks and returns, without ever awaiting maintenance:

1. **Enabled?** A configuration toggle (`background_refresh.enabled`) allows an operator to disable maintenance globally at consult time.
2. **Due?** A debounce stamp records the last trigger; if a refresh ran within the configured interval (default 24 h), the launcher exits. This is a rate limit, not a concurrency guard—several consultations can pass it within the same instant (Section 5.5).
3. **Claimed?** The right to spawn is taken with an *atomic* filesystem claim (a `mkdir`, which POSIX guarantees is atomic), so exactly one of several simultaneous triggers proceeds; a claim held past a bound is reclaimed so a crashed run cannot wedge refresh. Section 5.5 shows why the debounce stamp alone does not suffice here.

If all three gates pass, the launcher spawns a *fully detached* maintenance process—its own process group, output redirected to a log, never awaited—and returns. The consuming agent answers the user immediately. Maintenance, if it happens at all, happens after the read and is observed only by the *next* consultation. This is stale-while-revalidate applied to a knowledge corpus: whoever comes next pays the cost of freshness, asynchronously, and the reader who triggered it never does.

3.3 Maintenance operations are bounded and discovered

A maintenance run performs exactly one operation, selected by configuration. Five are defined, each a focused agent task with an explicit acceptance guarantee.

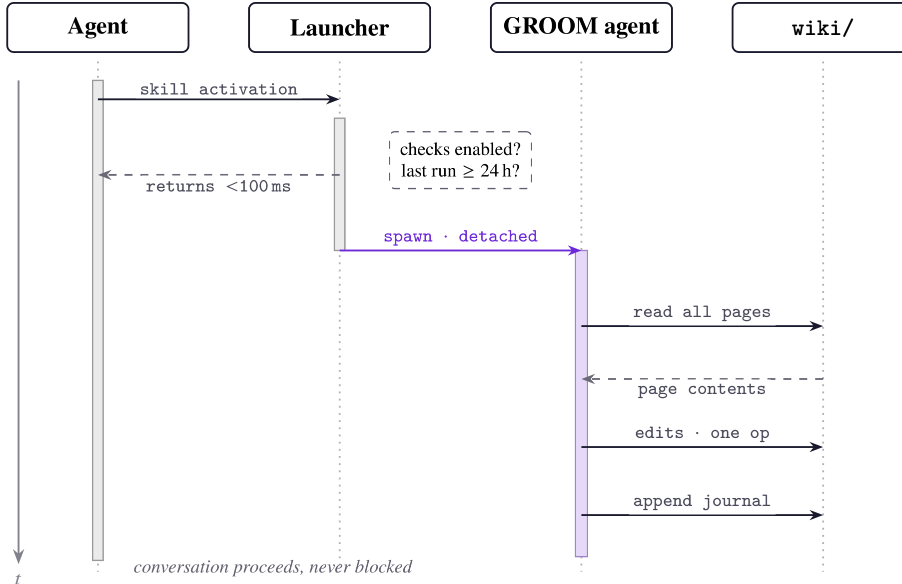

Figure 2. One consultation. The skill activates a launcher that checks its gates and returns within 100 ms (dashed), while a detached GROOM agent reads the corpus, applies one operation, and journals the run. The read is never blocked; an atomic `mkdir` claim serializes a racing consultation (Section 5.5), while the debounce stamp rate-limits across consultations.

Operation	Command	Guarantee
`lint`	Fix the form	Never alters meaning; page set unchanged
`prune`	Cut what repeats	Net line count must not increase
`expand`	Ingest what changed	Touches 3–6 files; vendor/spec sources
`research`	Admit new papers	Citation-gated; zero additions is valid
`iterate`	Improve the weakest page	Rewrites one page; validator-gated

Operations are *discovered from the filesystem*: each `<op>.md` prompt file is a runnable operation, and the composite `all` chains them. Adding an operation means dropping in a prompt file. The registry and the launcher's validity check both derive from the same directory, so the two can never disagree, and a misconfigured operation is rejected before it can burn the debounce slot. This applies the paper's own discipline: behavior lives in data (prompts), not in branching code.

3.4 Checkpointed execution makes autonomy safe

Autonomous edits to a live corpus are the system's central risk. A `prune` that hits its turn budget mid-merge can leave a page half-deleted with dangling inbound links; an `expand` that ingests a malicious web page can be steered to write outside the corpus; a confused agent can delete a load-bearing fact outright. We make every operation recoverable. Each is wrapped in a checkpoint and accepted only if it passes a deterministic gate, and we verify recovery empirically across nine fault classes (Section 5.4, Figure 3).

The executor wraps each operation as follows:

1. **Checkpoint.** Commit the corpus to git; stash any pre-existing dirty state so the operation starts from a clean, known-good tree.
2. **Run, scoped.** Invoke the maintenance agent with a per-operation capability set (Section 4) and a real-time guard that denies edits whose resolved path lies outside the corpus directory.
3. **Gate.** The operation *counts* only if (a) the agent reports terminal success rather than turn-truncation or error, (b) the deterministic validator passes, (c) no file outside the corpus changed, and (d) the operation's own postcondition holds (e.g. `prune` did not increase the line count).

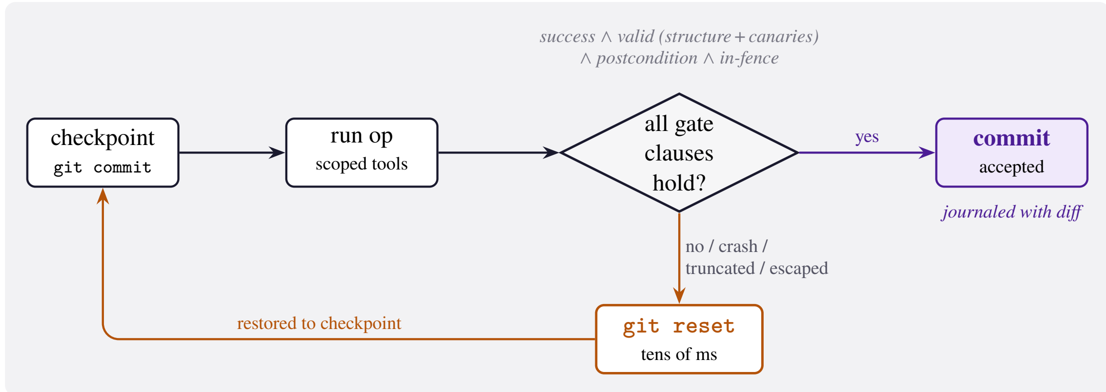

Figure 3. Checkpointed execution. Each operation begins from a committed, clean corpus and is accepted only if the agent reports terminal success, the deterministic validator passes (structure *and* fact-level canaries), the operation's postcondition holds, and no file outside the corpus changed. Any failure resets the working tree to the checkpoint (tens of ms, Section 5), converting mis-edits, crashes, turn-truncation, and fence escapes into recoverable no-ops.

4. **Commit or revert.** On success, commit the corpus changes with the operation name and a one-line summary. On any failure, `git reset --hard` returns the tree to the checkpoint and the failure is journaled.

This design moves the corpus's invariants out of the maintenance *prompts*, where they were merely requested of a stochastic agent, and into *code* that enforces them deterministically. The git history doubles as the observability layer the system otherwise lacks: every accepted change is a reviewable diff, every rejected one leaves a journaled `FAILED` record, and rollback is one command.

4 Implementation

GROOM is built on the Claude Agent SDK and runs on the host's configured Claude Code model and credentials, so an operator who upgrades their CLI upgrades the maintenance agent. The pipeline is roughly 300 lines of orchestration plus a 140-line validator; the corpus and the maintenance prompts are the bulk of the artifact. Three implementation choices follow directly from the paper's safety argument.

Schema-scoped capability. Each operation receives only the tools it needs. `lint` and `prune` are offline text operations, so they get read, edit, and prefix-restricted shell tools but *no web access*; only `expand`, `research`, and `iterate` receive `WebSearch` and `WebFetch`. No operation receives an unrestricted shell or the `find` tool, whose `-exec` primitive is arbitrary code execution. This realizes the "restrict by schema, not by prompt" principle: an operation cannot misuse a capability it was never granted.

Path confinement, twice. Edits are confined to the corpus by two independent mechanisms: a real-time permission callback that denies any write whose resolved path escapes the corpus directory, and—as a backstop that does not trust the SDK—the post-operation git gate, which reverts any out-of-corpus change before committing. Defense in depth: the callback prevents the write, the gate guarantees it cannot persist.

Deterministic validation. The validator (Section 3.4) is ≈140 lines of code, costs zero model tokens, and is reused in three places: as the per-operation gate, as a continuous-integration test, and as a free `status` command for operators. It is the mechanism by which the corpus's invariants are enforced rather than merely requested.

Table 1. Consuming-agent answer accuracy over a fresh vs. a staleness-injected corpus (3 runs/condition; 5 affected facts + 2 controls). Staleness collapses accuracy on the affected facts while leaving controls untouched.

Corpus	Affected facts	Controls	Overall
Fresh	15/15 (100%)	6/6 (100%)	21/21 (100%)
Staled	0/15 (0%)	6/6 (100%)	6/21 (28.6%)

Format interoperability. GROOM is format-native by default; a one-line configuration switch (`format: okf`) points the same runtime at an Open Knowledge Format bundle. Only the validator's notion of well-formed changes: the OKF validator accepts a hierarchical tree, requires OKF's single mandatory `type` field, treats `index.md`/`log.md` as reserved, and resolves path-based links; GROOM's `confidence` grade and the fact-level canaries ride along as OKF extension keys, which the spec requires consumers to preserve. An export/import maps the two frontmatter dialects losslessly (`summary` ↔ `description`, `updated` ↔ `timestamp`). The checkpoint gate, capability scoping, and consultation trigger never inspect the frontmatter, so safety and recoverability transfer to OKF unchanged.

5 Evaluation

We evaluate GROOM on its first production instance: a 19-page, 1,289-line knowledge base on harness engineering for AI agents. We begin with the question that motivates the system—*does corpus staleness actually degrade a consuming agent?*—then ask what a systems paper must answer for an always-on, autonomous component: *what does the read path cost? Does the safety machinery actually catch and recover from bad operations? What does it fail to catch? Does it hold under concurrency and at scale?* Measurements are on one machine (Apple Silicon, Node.js 22). We report medians with the spread, the number of trials, and the cold/warm split, and we are explicit where $n = 1$. The benchmark harness ships with the artifact.

5.1 Outcome: staleness degrades a consuming agent

The premise behind GROOM is that corpus staleness *matters*—that a stale page silently degrades any agent that grounds on it. Before measuring whether we can maintain the corpus safely, we test that premise directly. We take five load-bearing corpus facts and produce a *staled* copy in which each is altered (a wrong arXiv id, a wrong benchmark figure, a wrong rate, a renamed concept, a renamed method). A consuming agent answers a fixed question set treating the wiki as authoritative, as GROOM's skill instructs ("prefer the wiki over your training data for this domain"). The probed facts are near-future or corpus-specific, so the agent must lean on the corpus rather than its own priors. We record a consuming agent's answers over the fresh and staled corpora and grade them against ground truth—three runs per condition, with two unaffected control facts. The answers are transcribed from a live execution into the grading harness, so it certifies the grading, not a live re-run of the agent.

The effect is sharp and specific (Table 1). On the fresh corpus the agent answers the affected facts with 100% accuracy; on the staled corpus, 0%—it faithfully reports each injected error (15/15)—while the control facts hold at 100% in both conditions. The probes are deterministic-recall questions, so the collapse is clean by construction; the point is the propagation, not a graded difficulty. Corpus staleness passes one-for-one into wrong agent answers, on those five facts. This establishes the premise the design rests on: corpus correctness is outcome-critical. The rest of this section measures whether GROOM can keep the corpus correct cheaply and safely.

5.2 The consultation hot path is cheap

Because maintenance is detached, the only latency a reader pays is the launcher's exit time, and the design target is simply that it stay imperceptible. It does: over ten consecutive invocations the warm path has a median of 50 ms (40–70 ms, $n = 9$), comfortably inside the 100 ms budget, with one 210 ms cold start on the first call after boot that we report rather than hide. The point is the threshold, not the

Table 2. Latencies on the 19-page corpus. We separate the launcher's one-time cold start from the warm read path, and report recovery over 450 trials (50 reps across the nine fault classes of Section 5.4). The validator figure is in-process computation (Figure 4); invoked from the shell it adds the same ~50 ms Node startup as the launcher. Absolute latencies are load-sensitive; we report them for reference, and the paper's claims (a read path under the 100 ms budget, byte-identical recovery far below a multi-second maintenance run) hold at any plausible value.

Quantity	**Median**	**Range**	**Trials**
Launcher exit, warm (read hot path)	50 ms	40–70 ms	n=9
Launcher exit, cold start	210 ms	—	n=1
Checkpoint recovery (`git reset`)	13 ms	11–15 ms (p95 22)	n=450
Validation computation, 19 pages	0.8 ms	—	median of 50

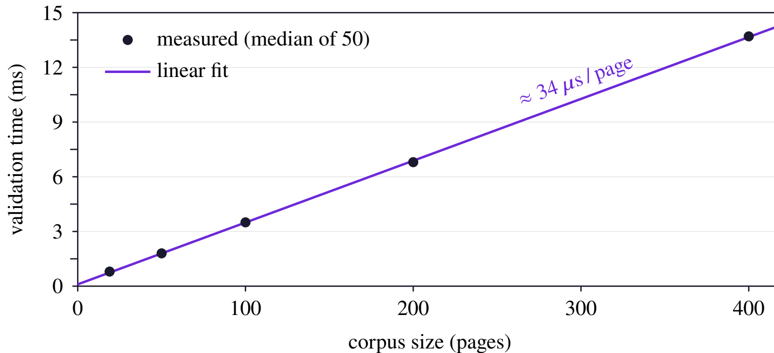

Figure 4. The deterministic validation gate scales linearly in corpus size: median in-process validation time over synthetically duplicated corpora from 19 to 400 pages (50 trials each, Apple Silicon, Node.js 22; excludes interpreter startup). The gate that gauges every maintenance operation costs ≈34 μs per page ($R^2 > 0.99$; absolute timings are load-sensitive) and stays in the low milliseconds (≈14 ms at 400 pages)—negligible against a multi-second LLM maintenance run and cheap enough to invoke unconditionally.

constant: the read path clears the budget with room to spare. We treat the absolute latencies here as order-of-magnitude and report them for reference (Table 2), since they are load-sensitive; what the system rests on is machine-independent—the gate's correctness, the byte-identical recovery below, and the validator's linear scaling.

5.3 The validation gate scales linearly

Because the gate runs on every maintenance operation, its cost must not grow faster than the corpus. We measure in-process validation time over synthetically duplicated corpora from 19 to 400 pages, 50 trials per size (Figure 4). The relationship is strongly linear (R^2 from 0.97 under load to >0.999 on a quiet machine) at tens of microseconds per page (34–70 μs/page, load-sensitive): 0.8 ms at the present 19 pages, and ≈14 ms (quiet) to ≈27 ms (under load) at 400. The gate thus stays in the low milliseconds and remains negligible against any (multi-second) LLM maintenance run—well beyond the scale at which the *maintenance operations* themselves would need to become incremental (Section 6).

5.4 Fault injection: bad operations are caught and recovered

To test the checkpoint guarantee directly we built a fault-injection harness. For each of *nine* fault classes, it materializes the working-tree state a faulty operation would leave behind, evaluates the real gate predicate (transcribed from the production runner), and—on rejection—runs the real recovery (`git reset --hard` followed by `git clean`). The nine classes exercise all four rejection clauses: structural faults the validator must catch (a broken link, malformed frontmatter, a future date, a deleted load-bearing fact, and a page deletion that breaks inbound links), terminal non-success (an

agent that crashes mid-edit; a turn-truncated run), an out-of-corpus write (a fence escape that edits `pipeline/`), and a violated postcondition (a `prune` that grows the corpus). We run two arms: GROOM, which evaluates the gate and recovers on rejection, and a *no-gate baseline*—the naive autonomous maintainer that commits whatever the operation produced.

Across 50 repetitions per class ($n = 450$), the gate rejected all nine classes. Recovery restored the corpus *byte-identically* to the checkpoint every time—the working tree equals `HEAD` after recovery, 50/50 per class—in tens of milliseconds (Table 2), two to three orders of magnitude below the maintenance run it reverts, so a rejected edit is effectively free. The no-gate baseline corrupted the corpus in all nine: a structurally invalid or factually wrong tree, an out-of-fence file, or an unwanted growth, committed and persistent. That contrast is the point, though the baseline half is true by construction: an unconditional committer necessarily persists whatever it is handed. The substantive result is the gated arm—the same nine operations become recoverable no-ops. "Byte-identical" is verified, not inferred: recovery succeeds only if the post-reset working tree is clean against `HEAD`.

Result. Nine fault classes, $n = 450$. With the gate: every failure becomes a byte-identical recovery in tens of milliseconds. Without it: corruption committed in all nine.

5.5 Concurrency: a race profiling forced us to fix

Our first design used the debounce stamp—written at spawn time—as the only concurrency control, assuming a second simultaneous consultation would see the stamp and skip. Profiling refuted this. We fired eight launchers concurrently with the stamp absent (a run is due) and repeated the race 500 times. The stamp resolved the eight triggers to *exactly one* run in only 28–59% of rounds (load-dependent; our committed run sits at the low end, ≈28%), leaking two or more simultaneous runs in the rest—up to six or seven, depending on the run. The cause is a classic time-of-check/time-of-use window: all eight launchers read the (absent) stamp and pass the debounce gate *before* any has written it. A single trial hides the bug entirely; the race surfaces only under repetition.

The fix is to make claiming the right to spawn atomic. We replace the check-then-write with an `mkdir` on a claim directory, which POSIX guarantees is atomic: exactly one concurrent caller creates it and proceeds, and a claim held past a configurable bound is reclaimed so a crashed run cannot wedge refresh. Re-running the identical 8-way race, the atomic claim resolves to exactly one run in **500/500** rounds. We keep the git checkpoint as the authoritative serialization point regardless: a second run that somehow began still commits or reverts against a versioned tree (Section 3.4). But the lock now does its job, and the episode is a reminder that a concurrency guard asserted from a single trial is no guard at all.

Result. An 8-way trigger race (500 rounds): the original debounce stamp yields exactly one run only 28–59% of the time (a TOCTOU race); an atomic `mkdir` claim yields exactly one in 500/500.

5.6 Structural gates are necessary but not sufficient—and canaries close the gap

A checkpoint that enforced only structure would guarantee valid frontmatter, resolving links, and a reachable index, but not *semantic* integrity. We show the gap concretely and then show it closed. We delete a load-bearing benchmark figure from a page—a specific quantitative result—while leaving the page structurally intact. Against structural checks alone the corpus **passes**: no link breaks, no frontmatter field is lost. Yet the page is now wrong by omission, and a `prune` that "deduplicated" the fact would be accepted.

We close this gap with *canaries*: a small set of `(page, pattern)` assertions, five in the present corpus, e.g. "`training-frontier.md` must match the Self-Harness benchmark figures." The same gate checks them mechanically, and an operation that legitimately moves a fact must update its canary in the same commit. With canaries active, the deleted-figure injection that previously passed now **fails the gate** ("`canary failed in training-frontier.md: lost Self-Harness Terminal-Bench gain`") and is reverted. To quantify the gap and its closure we ran a controlled ablation over seven structurally-valid semantic-loss injections (five guarded by a canary, two not; each alters a fact while

Table 3. Simulated read-weighted staleness (fraction of reads hitting a stale page) under consumption-triggered vs. round-robin maintenance at *equal* budget, across access skew; below, mean staleness of the hottest vs. coldest pages at s=1. Consumption-triggering keeps hot pages fresh and wins the read-weighted metric once access is skewed.

Zipf s	0.0	0.4	0.8	1.0	1.2	1.5
consumption	34.2%	32.4%	28.7%	26.0%	21.9%	**16.2%**
round-robin	22.7%	23.3%	23.1%	23.4%	23.1%	23.5%

mean staleness at s=1	hottest 10%	coldest 10%
consumption	**12.3%**	68.5%
round-robin	24.0%	22.7%

leaving form intact): the structural-only validator caught **0 of 5** canaried losses, the canary-augmented gate caught **5 of 5**, and *neither* caught the 2 deliberately un-canaried edits. Canaries thus extend the deterministic gate from structure to a curated set of load-bearing facts at zero token cost, but their coverage is targeted, not total. A before/after question-set benchmark remains the principled, heavier complement for un-canaried facts (Section 6); the outcome study (Section 5.1) is a first step toward it.

5.7 Consumption-triggered vs. scheduled maintenance

Does triggering maintenance on reads actually beat a fixed schedule of equal cost? A live end-to-end answer is beyond our current evaluation, but the *scheduling principle* can be tested in a discrete-event simulation under a stated model. We simulate $N = 50$ pages over T ticks. Each tick, every fresh page drifts stale with probability d (the world changes, independent of reads); M reads arrive drawn from a Zipf(s) distribution (a read *fails* if its page is stale); and a fixed budget of B refreshes is spent, either by *consumption* (refresh the pages just read) or by a *round-robin schedule*, on the identical budget. Neither policy observes drift; consumption is the only freshness signal, as in GROOM (Section 3.2).

Two findings (Table 3). First, the mechanism behaves as the thesis predicts: consumption-triggering keeps the *most-read* pages freshest. At moderate skew (s=1) the hottest 10% of pages are stale 12% of the time under consumption-triggering versus 24% under the schedule—it spends its budget where reads land—at the cost of the rarely-read tail (68% vs 23%). Second, the net read-weighted effect is regime-dependent. Under near-uniform access the schedule's exhaustive coverage wins, but consumption-triggering overtakes it as access skews, cutting read-weighted staleness by 31% at s=1.5. Consultation of a knowledge base is itself heavily skewed—a handful of pages answer most questions—so this is the operative regime. We are explicit that the advantage is regime-dependent and that this is a model, not a live-system measurement.

5.8 Generalization: structural entropy degrades retrieval across domains and retrievers

The evaluation so far uses one corpus. To test whether grooming's benefit generalizes, and to connect it to standard information retrieval, we built three more unrelated corpora, each modeling a kind of knowledge base that agents actually ground on: an internal API/SDK reference, a cloud deployment and SRE runbook, and a SaaS product support knowledge base. All three are fictional *internal* systems—which is both what real agent corpora are (proprietary, unseen by the model) and what guarantees the corpus is the only source of truth. Each has 12 content pages (plus an index) and 20 gold query-to-passage labels. GROOM is retrieval-agnostic: it maintains clean markdown, and any retriever consults it. We measure retrieval quality with two pluggable retrievers—BM25 (lexical) and a dense neural retriever (a sentence-transformer)—over each corpus in two states: *groomed* (the clean corpus) and *degraded* with the structural entropy grooming removes (duplicate and near-duplicate pages, boilerplate).

Across all three domains and both retrievers, removing that entropy lifts every standard IR metric (Table 4). Macro-averaged, the groomed corpus lifts recall@1 from 0.52 to 0.78 for BM25 and 0.56

Table 4. Grooming vs. structural entropy: standard IR metrics over three unrelated corpora, macro-averaged, for two pluggable retrievers. *Degraded* adds the duplicate/boilerplate pages grooming removes; *groomed* is the clean corpus. The groomed (clean) state wins every metric for both retrievers.

Retriever	State	recall@1	recall@5	nDCG@5	MRR
BM25 (lexical)	degraded	0.52	0.97	0.80	0.75
BM25 (lexical)	groomed	**0.78**	**0.98**	**0.92**	**0.90**
Dense (neural)	degraded	0.56	0.98	0.82	0.78
Dense (neural)	groomed	**0.81**	**0.98**	**0.93**	**0.92**

Table 5. Per-domain recall@1, degraded → groomed, for each retriever. Grooming helps in every one of the six domain×retriever cells; the magnitude varies with how often the injected noise pages collide with the real ones. MRR and nDCG@5 follow the same pattern (Table 4 gives the macro-average over all metrics).

	BM25		Dense	
Domain	degraded	groomed	degraded	groomed
internal API	0.60	0.72	0.58	0.70
SRE runbook	0.55	0.95	0.55	0.95
support KB	0.40	0.68	0.55	0.78

to 0.81 for the dense retriever (a 45–51% relative gain on the unrounded macro values), with matching gains in MRR and nDCG@5. The benefit is retriever-agnostic, not an artifact of lexical matching. It also holds in every domain×retriever cell, though the size varies (Table 5): the gain is largest for the runbook, whose terse, templated pages collide most with the injected duplicates, and smallest for the API reference, whose distinctive endpoint names resist the noise. Grooming also shrinks the corpus by 40% (aggregated over the three corpora), which is what the "load the whole corpus if it fits the context window" strategy pays for: a groomed corpus is cheaper to load wholesale. The single-corpus caveat thus weakens for the retrieval claim, which holds across three domains a model has no priors for. As with the fault harness, we are precise about the construct. This measures the corpus-*state* difference grooming targets—clean versus the duplicate/boilerplate entropy—not GROOM's operations performing the cleanup live. Running `prune`/`lint` to *produce* the groomed state and re-measuring is future work, alongside the live consumption-vs-schedule experiment (Section 6).

5.9 Interoperability: maintaining a different format (OKF)

A format claim should be measured, not asserted. We exercise GROOM's optional OKF mode four ways. *(i) Interop.* GROOM's OKF validator accepts Google's three published OKF bundles—`crypto_bitcoin` (5 concepts), `ga4` (11), and `stackoverflow` (49)—with zero errors, including their hierarchical subdirectories and path-based links; nothing is special-cased to our corpus. *(ii) Round-trip.* Exporting our 19-page wiki to OKF and back is content-lossless: every body and every field round-trips, the sole delta being OKF's required `type`, synthesized on export. *(iii) Safety is format-agnostic.* Re-running the nine-class fault matrix (Section 5.4) against the wiki exported to OKF, gated by the OKF validator, reproduces the native result: the gate rejects all nine classes and recovery is byte-identical to the checkpoint, 50/50 per class (n=450), in tens of milliseconds—git recovery and the gate predicate do not depend on the format. *(iv) Live.* On Claude Code defaults, a real `lint` run over an OKF bundle completed and was accepted by the gate (the already-clean corpus warranted no edits, so none were fabricated); a turn-truncated run was rejected and reverted—the recoverable-no-op guarantee, observed live on OKF. Two findings sharpen the boundary. The runtime is format-agnostic but the maintenance *prompts* are not: told nothing, a maintenance agent "fixes" OKF frontmatter toward our native schema and burns its turn budget; a one-line format note in the prompt resolves it. And the retrieval gain of Section 5.8 is, by construction, a corpus-state property our harness measures over page *bodies*—independent of the frontmatter dialect—so it transfers to OKF unchanged rather than constituting a separate result. OKF is a v0.1 draft; we track it as it

evolves.

5.10 Case study: the canary set came from real corrections

The canary set is not hypothetical. While preparing a companion survey that consumes this corpus, the consultation workflow surfaced two substantive errors already committed to the wiki—a misattributed authorship and an over-stated causal mechanism. We corrected both, and they are now among the guarded facts. We report this not as a controlled result but as the provenance of the canaries, and as an existence proof that consume-and-maintain puts attention on the corpus when it is used.

6 Discussion and Limitations

Correctness beyond curated facts is unmeasured. Canaries (Section 5.6) extend the gate to a hand-picked fact set, and structural validation covers form, but neither certifies that an *arbitrary* edit preserved truth. A `prune` can still subtly weaken an un-canaried claim. We make this explicit rather than claim a capability we have not measured: GROOM guarantees safety and recoverability, and verifies a curated fact set, but not general knowledge correctness. The principled remedy is to turn a before/after question-set into an *automated* semantic gate that the maintenance loop must pass, scored by an LLM judge with spot human review. Our outcome study (Section 5.1) is a manual instance of that check; turning it into a gate, with the canary set as its scaffold, is the most important next step.

Ingestion is an injection surface. `expand` and `research` write web-derived content into a corpus that future agents consume. This makes the corpus a potential prompt-injection vector that can, in principle, amplify across maintenance runs. Path confinement and capability scoping bound the blast radius of a single run, and the git history makes any injected change reviewable and revertible. For higher-stakes deployments, though, provenance tagging of web-sourced text and a human review gate before publication are warranted.

Single-instance evaluation. Our core latency, safety, and outcome measurements use one 19-page corpus. The retrieval generalization (Section 5.8) adds three unrelated domains, though those corpora and their relevance labels are author-constructed, so they test genre diversity rather than statistical independence; a live end-to-end deployment at scale remains future work. The validation gate and launcher are $O(\text{pages})$ and $O(1)$ respectively. The maintenance operations as written, though, re-read the full corpus per run, which will not scale to hundreds of pages. An incremental variant (operate only on pages changed since the last run) is necessary at scale and is straightforward given the existing per-operation stamp.

Trust in the trigger. Because the skill auto-runs a repository script on activation, GROOM should be installed from a trusted source; a malicious fork could turn consultation into code execution. We recommend running the trigger only from within the repository (rather than a user-level symlink) for untrusted corpora.

Threats to validity. Our timings are from a single machine and are load-sensitive, so we treat absolute latencies as order-of-magnitude rather than precise: across runs the validator's per-page cost ranged over 34–70 μs and checkpoint recovery over roughly 11–25 ms. The claims we rely on are machine-independent—byte-identical recovery, the 9/9 gate rejection, the 500/500 lock, and the validator's linear scaling ($R^2 \geq 0.97$)—and the order-of-magnitude gaps (a read path under the 100 ms budget, recovery far below a multi-second maintenance run) hold across hardware. The fault-injection harness evaluates the gate predicate *transcribed* from the production runner rather than the runner driving a live model, so it certifies the gate and recovery logic, not the agent that precedes them—faithful for the recovery claim, but a construct boundary we state plainly. The outcome study uses three runs per condition over seven questions on one corpus, and its five affected facts coincide with the canary set, so it speaks to canaried facts rather than arbitrary drift. The effect is large and clean, but it establishes that staleness propagates to a faithful consuming agent, *not* the end-to-end

lift of consumption-triggered over scheduled maintenance. And the canary set is small and author-curated, so the ablation measures coverage of *those* load-bearing facts, not of all of them.

7 Conclusion

Agent knowledge bases rot, and the maintenance that would prevent it is deferred forever because it is nobody's immediate concern. GROOM removes the need for anyone to remember. It makes consultation the trigger, so the corpus is maintained in proportion to how much it is used, and it makes that maintenance safe under autonomy by checkpointing every operation behind a deterministic gate. The result is a knowledge base maintained as a side effect of being read, with no latency on the read path and every autonomous edit checkpointed and reversible. The honest open problem is semantic correctness: structural gates cannot tell that a true page became a false one. We narrow this problem with fact-level canaries but do not close it. The natural next step is a live, end-to-end demonstration that consumption-triggered maintenance beats a schedule on a real agent task; our simulation predicts it will, under skewed access. We release GROOM as a content-agnostic, format-agnostic pipeline—maintaining its native corpus or an Open Knowledge Format bundle through the same gate—so that any agent-consumed corpus can adopt the same discipline.

Availability. GROOM is open source; the implementation, the validator, the fault-injection harness, and the harness-engineering corpus that serves as its first instance are available at `https://github.com/beconfident-ai/groom`.

References

[1] Anthropic. Effective context engineering for AI agents. Engineering blog, 2025.

[2] B. Böckeler. Harness engineering for coding agent users. martinfowler.com, 2026.

[3] J. Liu et al. Dive into Claude Code: the design space of today's and future AI agent systems. arXiv:2604.14228, 2026.

[4] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[5] P. Lewis et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *NeurIPS*, 2020.

[6] M. Nottingham. HTTP Cache-Control extensions for stale content (stale-while-revalidate). RFC 5861, 2010.

[7] K. Meng, D. Bau, A. Andonian, and Y. Belinkov. Locating and editing factual associations in GPT (ROME). In *NeurIPS*, 2022.

[8] C. Packer et al. MemGPT: towards LLMs as operating systems. arXiv:2310.08560, 2023.

[9] N. Shinn et al. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

[10] S. Yao et al. ReAct: synergizing reasoning and acting in language models. In *ICLR*, 2023.

[11] H. Zhang et al. Self-Harness: harnesses that improve themselves. arXiv:2606.09498, 2026.

[12] A. Karpathy. LLM wiki: an LLM-maintained markdown knowledge base. Public gist, April 2026. `https://gist.github.com/karpathy/442a6bf555914893e9891c11519de94f`.

[13] J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein. Generative agents: interactive simulacra of human behavior. In *UIST*, 2023.

[14] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: an open-ended embodied agent with large language models. arXiv:2305.16291, 2023.

[15] A. Asai, Z. Wu, Y. Wang, A. Sil, and H. Hajishirzi. Self-RAG: learning to retrieve, generate, and critique through self-reflection. In *ICLR*, 2024.

[16] A. Madaan et al. Self-Refine: iterative refinement with self-feedback. In *NeurIPS*, 2023.

[17] K. Meng, A. Sen Sharma, A. Andonian, Y. Belinkov, and D. Bau. Mass-editing memory in a transformer (MEMIT). In *ICLR*, 2023.

[18] Y. Yao, P. Wang, B. Tian, S. Cheng, Z. Li, S. Deng, H. Chen, and N. Zhang. Editing large language models: problems, methods, and opportunities. In *EMNLP*, 2023.

[19] T. Vu, M. Iyyer, X. Wang, N. Constant, J. Wei, et al. FreshLLMs: refreshing large language models with search engine augmentation. In *Findings of ACL*, 2024.

[20] Z. Jiang, F. F. Xu, L. Gao, Z. Sun, Q. Liu, J. Dwivedi-Yu, Y. Yang, J. Callan, and G. Neubig. Active retrieval augmented generation (FLARE). In *EMNLP*, 2023.

[21] A. Halfaker and R. S. Geiger. ORES: lowering barriers with participatory machine learning in Wikipedia. *Proc. ACM Hum.-Comput. Interact.*, 4(CSCW2), 2020.

[22] D. M. Wegner. Transactive memory: a contemporary analysis of the group mind. In *Theories of Group Behavior*, pp. 185–205. Springer, 1987.

[23] Google Cloud. The Open Knowledge Format (OKF), v0.1 draft. 2026. `https://github.com/GoogleCloudPlatform/knowledge-catalog`.