RESEARCH / PRODUCTION FIELD REPORT · V1.1 · 2026-09-20

Fast Memory, Slow Weights: A Production Architecture for Self-Improving AI Agents at the System Layer

Why production agents should learn continuously without continuously rewriting their models

By **Pavle Lazic** · ScalablyAI ·

canonical https://scalably.io/research/fast-memory-slow-weights

Abstract

We run a multi-tenant AI-agent platform in production and tried to make its worker model learn from its own accumulated experience. The first consolidation attempt, two rank-16 LoRA adapters trained on 423 verified trajectory records, improved the behaviour it targeted and moved an untargeted behaviour the wrong way by more than our pre-registered gate allows. On 73 recovery states, exact correctness rose from 31 to 38 (7 adapter-only wins, 0 losses, p = 0.0156); on 160 known-failure states, exact repeats of a failed action fell from 20% to 15% (p = 0.0215); on 73 clean decisions, correctness fell from 49 to 45, a difference of +3/−7 discordant pairs (p = 0.34) that the suite cannot separate from noise but that fails a −2-point non-inferiority gate. The adapter was rejected by the gate we had written before training it. The lesson is not that fine-tuning forgets, which the literature predicts, but that at production sample sizes a single weight update produces gains and losses that cannot be told apart cheaply, and cannot be reversed separately. That is the case for learning on two timescales. A fast loop writes verified experience into an external memory that is scoped, editable, attributable and reversible, and changes behaviour on the next turn without touching weights; as of 20 September 2026 that loop holds 433 rules and 3,422 memory files across 74 workspaces, with 37 rules retired through a provenance log. A slow loop consolidates only patterns demonstrated repeatedly, through a curated training set, a frozen 484-decision

evaluation, a nine-condition promotion gate and shadow deployment. We describe the implemented mechanisms, the production evidence, the failure modes the design does and does not address, and the sense in which the loop is recursive. We claim a practical architecture for recursive self-improvement at the agent and system layer under human-written gates, not a solution to continual learning and not model-level self-improvement. The effect of the fast loop on task outcomes has not yet been measured; the experiment that would measure it is specified.

1. The production failure that motivated the architecture

Scalably operates an agent runtime for real businesses: one isolated container per workspace per run, tool access through gated integrations, scheduled and on-demand work across five hosts and 169 workspaces (S24). Between February and August 2026 that runtime left behind 4,471 main sessions, 3,623 subagent runs and 143,145 tool-use blocks with their results, 8,300 of them explicit tool errors (S1). Every record is a node in a causal graph: prompt, decision, tool call, observation, verification or recovery, outcome (S3). It is the kind of corpus that invites the obvious question. If the agents have done this much work and left this much evidence, why not train the worker model on it?

We tried. In August 2026 we built a training pipeline that reads the corpus read-only, applies typed redaction and tenant-scoped pseudonyms, keeps raw tenant data out of git, and never touches the production inference machine (S14). From a frozen eligible pool of 14,907 decisions (S18) we curated 423 records per arm and trained two LoRA adapters (R22) on Qwen3.8-27B on a rented service for $53.88 in total: rank 16, alpha 32, all attention and MLP projections, learning rate 5e-5 with a cosine schedule, batch 16, 32,768-token sequences, two epochs, 54 steps (E1, E14). One arm used Qwen-native targets only ("Q-only"); the other mixed sources ("mixed"). Training loss fell 16 to 18% between epochs, which proves that optimisation moved and nothing else (E3).

Then we evaluated both adapters against the exact BF16 base model on a rented GPU, with deterministic paired scoring, on a suite frozen before training: 60 training-sanity decisions drawn from the training distribution to confirm the adapter had moved at all, plus 424 task-disjoint, full-context decisions (S15, E4). The 424 are two sets. The first is 264 positive decisions in three disjoint strata: 73 verified-recovery states, where a tool had just been rejected and the recorded next move was a different tool; 73 clean-trajectory decisions, taken from sessions with no failure; and 118 verified-action decisions. The second is 160 known-failure states, captured just before a historically failed action, sharing no decision with the recovery stratum (E12). A decision is scored correct when the chosen tool equals the recorded tool and the set of argument keys, nested and path-qualified, equals the recorded set; tool choice alone is

reported separately. Correctness therefore measures conformity to what the recorded agent did on a trajectory whose outcome was verified, not an independent judgement of quality (E12).

The result had three parts (E13).

The adapter learned what we wanted it to learn. On the 73 recovery states, exact correctness rose from 31 to 38 for the Q-only arm: seven cases moved in the adapter's favour and none moved against it, exact two-sided sign test p = 0.0156. On the 160 known-failure states, exact repetition of the historically failed action fell from 32 to 24 for both arms, a 25% relative reduction, nine fixes against one regression, p = 0.0215 (E9).

The adapter also moved something we had not asked it to touch. On the 73 clean decisions, correct choices fell from 49 to 45 for Q-only: three cases improved, seven regressed, p = 0.34. Our serious-candidate gate, written in the training programme before any job ran, requires clean-task performance no worse than −2 points (S16). A drop of 4 on 73 is about −5.5 points. The mixed arm did the same in both directions with less force: recovery 31 to 36 (+7/−2, p = 0.18), clean 49 to 46 (+5/−8, p = 0.58). Over all 264 positives the Q-only arm moved +14/−11 (p = 0.69) and the mixed arm +17/−16. Tool choice did not change in any stratum; every discordant pair is an argument-key difference. We report both arms; we quote Q-only as the better arm because it is the only one that passed any gate, and we treat all of these tests as exploratory, uncorrected for the five measures in Table 1.

We rejected both adapters. The ledger entry reads: "behavioral movement and targeted recovery learning are proven, but neither adapter is a serious or production candidate" (E10). Nothing was deployed.

Two things about that rejection matter more than the numbers. First, the gate is a policy over point estimates, and at n = 73 it cannot tell −4 from sampling noise; it fired anyway because the gate is the rule, and the only cure for its resolution is a larger suite, not a looser gate. Second, the training set contained no clean-trajectory examples at all, which is the textbook condition for forgetting (R24, R28) and which our own next curriculum corrects (E11). So the experiment did not show that fine-tuning must forget. It showed something more useful for a production operator: one $54 update produced a real gain and a possible loss in the same artefact, we could not tell them apart at the sample size production affords, and we could not keep the one without the other.

> If every useful experience immediately changes the weights, learning and corruption can become the same operation, and at production sample sizes you cannot tell which one you performed.

External memory can be edited, scoped, attributed, evaluated and rolled back one line at a time. Weights cannot be treated that casually. The rest of this paper is what we built around that sentence, what it already does in production, and what it does not yet do.

2. Why continuously modifying weights is dangerous

Nothing in Section 1 surprises the continual-learning literature. Catastrophic forgetting, the loss of previously learned capability when a network is trained on new data, has been documented for decades, and the standard mitigations, elastic weight consolidation (R23) and experience replay (R24), were designed for it. For large language models, Luo et al. observed forgetting across 1B to 7B models during continual instruction tuning (R25), Li et al. linked its extent to the flatness of the model's loss landscape (R26), a 2026 mechanistic analysis traces where in the network it happens (R27), and Chen et al. showed that mixing random or generic data into fine-tuning mitigates forgetting of memorised facts (R28). A survey organises the field into continual pre-training, domain-adaptive pre-training and continual fine-tuning (R29).

What the literature does not resolve is the operational problem. A production agent generates a stream of experience whose value is uneven, whose correctness is only sometimes verifiable, and whose relevance is scoped: a correction that is right for one client's invoicing workflow is wrong for another's. Weight updates have four properties that make them the wrong first destination for that stream.

They are shared. An adapter serves every request that hits it. Per-workspace adapters are possible in principle, and multi-adapter serving is standard; we do not run them, because 169 workspaces at a few hundred verified records each is a fleet of cold-start adapters with no evaluation behind any of them, and per-user parametric memory (R6) presupposes a base model built for it, which a stock open model is not. Until a workspace has enough verified, repeated experience to pass the same gate a global candidate must pass, its corrections belong in a file, not a tensor.

They are entangled. The 49 → 45 movement in Section 1 happened on decisions that were not in the training set. Whether it was forgetting or noise, the point stands either way: gradient updates that improve one behaviour move parameters that other behaviours depend on, and the training run carries no record of which. Nothing in the adapter records what it forgot, and at n = 73 nothing in our suite could either.

They lack selective reversal. Unloading an adapter is atomic and leaves the base untouched, so weights are not hard to roll back. They are hard to roll back partially. The only way to remove

the 49 → 45 movement was to discard the adapter, which discarded the 31 → 38 gain with it. A memory rule can be retired alone.

They lack attribution. A memory line can carry the quote that produced it, the date and the scope. A weight delta cannot say which trajectory put it there.

None of this argues against training. Our whole pipeline exists to train. It argues that a weight update should be the last thing that happens to an experience, after it has been verified, repeated, curated and evaluated, not the first.

3. Fast memory versus slow weights

The architecture separates learning into two loops that operate on different timescales and different substrates.

Fast memory. Experience that passes outcome verification is written into an external memory that the agent reads every turn. The write is immediate, scoped to the workspace, editable by a person or a maintenance agent, attributable to a source, reversible by moving a line, and subject to caps and expiry. It changes the agent's behaviour on the next turn. No weights move.

Slow weights. Experience that has been verified and has recurred across many sessions becomes training material. It passes through curation, a frozen evaluation suite, explicit numerical gates and shadow deployment before a candidate model is promoted. The base model stays stable between promotions. Nothing enters weights because one user said "remember this."

Both loops learn. The split is by reversibility: fast memory is where learning is cheap to undo, so it is where learning happens first and most often; weights are where learning is expensive to undo selectively, so they change rarely and only on accumulated evidence.

Two consequences follow. First, the model is trained to follow current policy and verify current facts, while mutable rules, skills, tool schemas and world knowledge stay versioned runtime inputs (S21). Second, the two loops share one verification layer. The guard and QA machinery that decides whether an outcome counts as verified is the same machinery that decides whether an experience is eligible for memory and, later, for training.

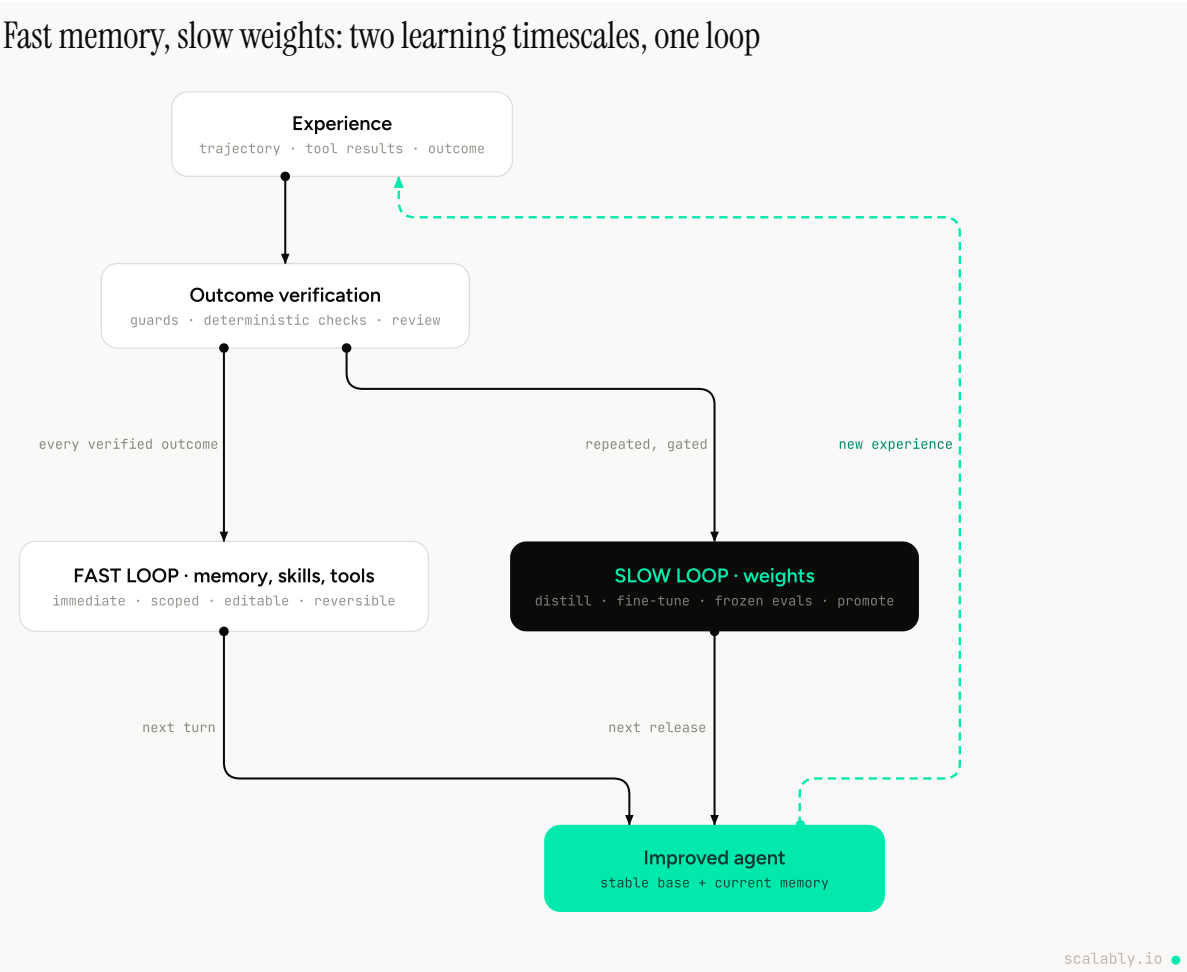


Figure 1. Fast memory, slow weights. Every verified outcome can change the next turn through memory; only repeated, gated experience changes the next release through weights. The green return edge is the loop.

4. Scalably's architecture

Everything in this section is implemented and running unless marked planned. Register rows give the file, commit or document for each claim. For each mechanism we also say who performs the write, because that decides whether "the system improves itself" is a description or a metaphor.

4.1 Verification before anything is learned

A tool-outcome guard sits between every tool result and the model, in every workspace. It classifies each result into one of eight anomaly kinds before the model can read it as success: empty result, failure reported in a success channel, invalid envelope, failed, blocked, partial, pending, no-op (S11). The point is that "the tool returned" is not evidence of "the tool worked",

and an agent that learns from unguarded outcomes learns to repeat the wrong ones. This guard is code; it runs without anyone's decision.

At the end of a run, a stop-time QA step runs deterministic checks first: unresolved placeholders, tool errors the final answer never acknowledged, counts that disagree with the tool results. Then an evidence-bound reviewer, which can run on a different model and endpoint than the worker, reads the run and returns a verdict that can escalate to a human. The step has five modes from off to enforce-cascade, is off by default and is enabled per workspace (S12). As of 20 September 2026 it is enabled on 14 of 171 registered workspaces, all in shadow modes and none enforcing (S26). So today most trajectories carry the guard's label and the runtime's outcome record, and a minority also carry a reviewer verdict; the curation step in the slow loop reads whichever labels exist and treats their absence as a reason to exclude, not to assume.

4.2 The fast loop: file-based, scoped, provenanced memory

Memory is plain files under each workspace's `memory/` tree and `rules/`, injected into every turn as a short summary with detail read on demand through an index (S4). The agent writes it. A standing rule in every workspace's prompt instructs the agent, when a conversation produces a correction or a standing instruction, to append one line of at most 160 characters with a stable identifier under a `## Pending` heading in `rules/learned-corrections.md`, in the same turn and without narrating it; durable facts replace the line they supersede rather than accumulating a contradicting second version (S5, S10). Pending rules are in force from the moment they are written; "pending" refers only to their later integration and de-duplication (S5). People can edit the same files directly, and do.

Every rule has a provenance record in `reference/corrections-log.md`: the rule as it currently stands, its dated history, the kind of evidence (user-explicit or otherwise), its scope, the source quote that produced it, and a wrong/right example. A retired rule is moved to a `## Retired` section with the identifier of the rule that superseded it and the date (S6). The rule file is capped at 60 rules; at the cap, consolidation happens before addition (S5).

Consolidation is performed by scheduled maintenance agents, not by people. A nightly task integrates and de-duplicates Pending entries and writes the provenance record. A weekly task archives the finished week, merges and compresses rules, verifies staleness across the tree, rebuilds the index and ends by running a mechanical check script; its contract states "moves, not deletions" and "caps are numbers, not vibes" (S7). Because each workspace has its own tree, a rule in one tenant's memory never reaches another's (S8), and because the files are plain text with identifiers and a ledger, any rule can be found, read and reversed by a person without model tooling (S9).

This is the whole fast loop: trajectory → outcome → verification → memory candidate → retrieval on the next turn → changed behaviour. The maintenance contract for it is public as the `memory-system` skill (S4).

4.3 The slow loop: curation, frozen evaluation, gates

The slow loop starts from the same corpus, read-only, outside git, with typed redaction and tenant-scoped pseudonyms; the production inference machine is never used for training (S14). A frozen policy audit found 14,907 eligible decisions from 1,883 sessions in 707 task clusters, of which 8,755 are clean, 2,506 are verified actions and 3,650 are recoveries; five tools account for about 84% of actions, so coverage rather than volume is the limiting factor (S18). Curation is code with a policy: replay-proven corrections, meaning a failed action whose corrected replay on the same state succeeded, outrank raw self-generated actions as training targets (E11, S20). A training run is launched by a person, with a funded cap and a written scope (E2).

Candidate models are evaluated on a frozen suite built before training: 60 training-sanity decisions and 424 task-disjoint full-context decisions across 68 clusters and 22 current-schema tools, with every request contract excluding its target action so the model cannot read the answer from the prompt (S15). Evaluation runs on the exact high-precision base model in an isolated rented environment with deterministic paired scoring (E4). The suite is versioned; a new failure case enters the next version, never the one a candidate is being scored on.

A deterministic playground exists as an MVP: a content-free snapshot of current policy, 100 scenarios, 27 stateful tools, six policy families, with 100/100 oracle passes and broken-to-corrected replay on the same state. No scenario is training-authorised yet (S20). A deferred-replay capture path exists in the runtime in capture mode only; it writes content-free replay descriptors and, in the document's words, "does not run Qwen, fetch tools, retain raw prompt or result text, call a teacher or authorize training. Unknown modes fail closed" (S13). Replay against a candidate model in a sandbox is planned.

4.4 Production shape

The worker model in production today is a stock open 27B model, Qwen3.8-27B quantised to NVFP4, served by vLLM on two consumer GPUs, carrying tool loops, drafts and a document digitiser: 28,097 requests in 13.9 days, an 82.55% prefix-cache hit rate, zero engine errors (S23); the full serving configuration and counters are in the production field report, and the same box's controlled comparison of a dense and a mixture-of-experts checkpoint, including a replay of 459 production decisions, is in MoE vs dense inference on two RTX 5090s. It is deliberately not the last word in the building: frontier models stay in the loop for hard

reasoning and review. Its job is to run verified, repetitive work cheaply enough that repetition is not rationed. The target shape, documented as a plan, is deterministic guards → specialised worker → deterministic validators → a small specialised QA verifier → pass, retry, repair or escalate (S22).

5. Production evidence

5.1 The fast loop in numbers

The fast loop has not been measured for its effect on task outcomes, and we do not claim one; the experiment that would produce that number is the first item in Section 10. What can be reported is its state. On 20 September 2026, counted read-only on the five hosts without reading any content (S25): 74 workspaces hold a `learned-corrections.md`, with 433 identified rules in force and 53 awaiting nightly integration; 58 workspaces hold a provenance log, in which 37 rules have been retired with a superseding identifier; 199 workspaces hold a memory tree, 3,422 markdown files in total, of which 438 were modified in the preceding seven days; the largest single rule file holds 47 rules against the cap of 60. One host runs a different consolidation variant and holds no rule file. These are counts of a mechanism in use, not evidence that it helps.

5.2 The slow loop: one full pass

The evidence for the slow loop is the S1 experiment in Section 1, and it is evidence in both directions. Table 1 restates it for the better arm.

MEASURE (Q-ONLY ARM)	BASE	ADAPTER	DISCORDANT PAIRS	GATE	RESULT
Verified recovery, exact (73 states)	31	38	+7 / −0, p = 0.0156	positive paired effect	pass
Exact failed-action repeats (160 states)	32 (20%)	24 (15%)	9 fixes / 1 regression, p = 0.0215	≥ 20% relative reduction	pass (25%)
Clean-decision correctness (73)	49	45	+3 / −7, p = 0.34	≥ −2 points	fail (about −5.5)
Paired correctness, all 264 positives	138	141	+14 / −11, p = 0.69	≥ +5 points	fail (+1.14)
Training sanity (60, training distribution)	34	38			

Table 1. S1 results, Q-only arm (E5 to E9, E13). Mixed arm: recovery 36 (+7/−2, p = 0.18), clean 46 (+5/−8, p = 0.58), all positives 139 (+17/−16). Tool choice alone was unchanged in every stratum. Training cost $53.88, evaluation about $10.47 (E2). Tests are exact two-sided sign tests on discordant pairs, uncorrected for multiple comparisons.

The evidence for the verification layer is structural: the guard's eight kinds and the QA step's checks are in the runtime, and the deterministic checks are the reason trajectories carry outcome labels a curator can trust (S11, S12).

Four caveats bound everything above. The corpus counts are lower bounds from an August snapshot (S1). The evaluation strata are small; 73 is enough for an exact test on discordant pairs and not for a two-point gate. The production worker is quantised while the evaluation ran on the high-precision base, so any deployed gain would also have to survive quantisation, which is a separate gate (S17). And the whole record comes from one company's clients, with five tools covering 84% of actions (S18).

6. Promotion and rollback gates

The gates are the mechanism that turns "we trained a model" into "we may deploy it". They were written before the first job ran, and they are what rejected S1.

The serious-candidate gate has nine conditions (S16): at least +5 absolute points on paired decision correctness; at least 20% relative reduction in the targeted failure families; a paired confidence interval that supports a real positive effect; no new critical rule, privacy or security failures; clean-task non-inferiority no worse than −2 points; no increase in unsupported completion, meaning a final answer that claims work the tool record does not show; no regression in isolated replay task completion; tool, token and latency overhead no worse than 15% without compensating quality; and a confirmation checkpoint or seed that reproduces the gain. S1 passed the second and third, failed the first and fifth, and was not run against the rest.

The production-candidate gate adds six conditions (S17): the high-precision gain survives NVFP4 conversion; fresh chronological tasks confirm the direction; isolated live replay passes; shadow traffic improves on fresh tasks; a low-risk canary and a measured cohort show no safety or reliability regression; and the previous production checkpoint remains immediately rollbackable. No candidate has reached this gate, so everything that follows about model rollback is design, not record.

Rollback is asymmetric by design. In the fast loop, rollback is a line moved to `## Retired` with a date and a superseding identifier, or a direct edit of the file; it takes effect on the next turn and leaves a record (S6, S9). Retiring a rule discards that rule's benefit, but only that rule's. In the slow loop, the plan is that a promoted model that later fails shadow or fresh-task checks is replaced by the previous checkpoint, which the gate requires to be kept ready, and the training set that produced it is annotated with the failure so the next curriculum learns from the rejection. The asymmetry is intentional: the loop that is cheap to reverse selectively is the one that runs continuously.

7. Relationship to existing research

We separate four things: what prior work demonstrated, what we observed, what we propose, and what remains unproven.

Two timescales. MetaSkill-Evolve (R1) is the closest published design: five pipeline agents share a single frozen backbone, task skills evolve on a fast loop and a meta-skill evolves on a slower loop under the same pipeline applied to itself, with reported held-out gains of +23.54, +16.09 and +1.92 points on OfficeQA, SealQA and ALFWorld over the raw backbone. It

demonstrates that recursive improvement with frozen weights is real and measurable. Our fast loop is the same idea applied to production memory and procedures rather than benchmark skills; our addition is the gated path from repeated experience into weights, which MetaSkill-Evolve does not take. Nested Learning (R3) argues from the model side that a learning system is better described as nested optimisation problems at different update frequencies, and Titans (R4) builds a neural long-term memory that learns at test time. Titans is an in-weights fast memory; Nested Learning is the theoretical case for having more than one timescale at all. Tiwari et al. (R30) propose the same split at the model level, published four months before this report: optimised context as "fast" weights and parameters as "slow" weights, which on reasoning tasks reaches the performance of parameter-only reinforcement learning with up to three times fewer samples and stays closer to the base model (up to 70% less KL divergence), with less forgetting. Their fast weights are an optimised prompt for a task; ours are scoped, provenanced production memory with a gated path into weights. We add this citation in revision v1.1. We place fast memory outside the weights at the agent layer, because in a multi-tenant deployment scope and reversibility are properties of files, not of parameters.

Memory from trajectories. ReasoningBank (R2) distils strategies from both successful and failed trajectories into a retrievable memory and reports self-evolving behaviour without weight updates. Agent Workflow Memory (R7) induces reusable workflows from experience. A-MEM (R8), Mem0 (R9) and MemGPT (R10) organise long-term agent memory; Generative Agents (R11) and Reflexion (R12) established memory streams and stored reflections; Voyager (R13) stored skills as executable code. These demonstrate that external memory improves agents. What we add is an operating discipline rather than a retrieval mechanism: caps, provenance, scoped trees, mechanical checks and scheduled consolidation, because a memory that nobody can audit becomes a liability in production.

Verification-gated writes and harness repair. ErrorProbe (R16) keeps a verified episodic memory that updates only when an error pattern is confirmed by executable evidence. HarnessFix (R17) treats failed trajectories as evidence for repairing the harness (tools, context, orchestration, verification) rather than the model, reporting 6.3% to 18.4% improvements. PROBE (R18) turns failure-anchored diagnosis into bounded, evidence-grounded recovery guidance, with 65.37% diagnosis accuracy on 257 unresolved cases. SAGE (R19) routes failure hypotheses to the right abstraction level. Our tool-outcome guard and stop-time QA are the same principle at the runtime layer: no experience counts until something other than the model says it worked. Self-Refine (R14) and Self-Debugging (R15) improve within an episode; the loop here is across episodes.

Memory as a separate axis in the model. DeepSeek's Engram (R5) introduces conditional memory via O(1) N-gram lookup as a sparsity axis complementary to mixture-of-experts, finds a U-shaped allocation law between neural computation and static memory, and reports that a

27B Engram model beats an iso-parameter, iso-FLOPs MoE baseline with long-context retrieval improving from 84.2 to 97.0 on multi-query needle-in-a-haystack. This is strong evidence that memory and computation should be separated. It is not our mechanism. Engram's memory is an embedding table addressed by token N-grams and learned in pre-training; it does not change per tenant or per correction at runtime. Our fast memory is text, edited at runtime, scoped and reversible. The two are complementary, and a production agent could use both. "User as Engram" (R6) internalises per-user memory as local parametric edits that are, by its own design, disjoint across users, composable and inspectable; it is the strongest published form of the parametric route, and we do not argue against it on reversibility or attribution, which it addresses. We do not take it for two narrower reasons: it presupposes an Engram-architecture base model, which the stock model we serve is not, and its edits carry facts, while most of what our fast loop stores are procedures and corrections that a person must be able to read and change without model tooling.

Forgetting. Sections 1 and 2 rely on R23 to R29. Our contribution there is only a small, exact, production-derived instance of a known phenomenon, reported with its discordant pairs.

What "recursive self-improvement" means. Anthropic defines the strong form as AI fully autonomously designing and developing its own successor, and states that we are not there yet (R21). A 2026 survey spans bounded self-refinement to autonomous research loops (R20). Our claim sits at the bounded, system-layer end: the system improves the mechanisms that improve it, under human-written gates, with people performing several of the edges. We do not claim model-level recursive self-improvement, and we do not claim the architecture is optimal.

8. Failure modes

External memory does not solve learning. It makes learning observable, reversible and governable, which is a different and smaller claim. Each failure mode below names what the architecture does about it and what it does not.

Poisoned memories. A malicious or confused input could write a rule. Mitigation: rules are one line, capped, provenanced with the source quote, integrated by a maintenance agent that can reject them, and scoped to one workspace; any rule can be retired with a record. Not solved: a plausible-looking wrong rule survives until a person or a failed outcome exposes it.

Incorrect user corrections. Users are sometimes wrong. Mitigation: the provenance record marks a rule as user-explicit, and a later correction replaces rather than accumulates. Not

solved: the system trusts the workspace owner by design; a wrong instruction is followed until it is corrected.

Retrieval errors. The essentials are injected every turn and the rest is read on demand through an index, so the failure is a missed read, not a wrong hit. Mitigation: caps keep the injected files short enough to be read whole; the index must list every file. Not solved: retrieval quality is not independently measured today (Section 10).

Accumulation and noise. Mitigation: the 60-rule cap, the 160-character line, nightly de-duplication, weekly compression, a check script that fails on cap overruns. This is the part of the design we are most confident in, because it is mechanical, and the counts in Section 5.1 show it holding: the largest file sits at 47.

Tenant contamination. Mitigation: one tree per workspace, learned corrections appended to the workspace's own file, pseudonymised and redacted data in the training corpus. Not solved for the slow loop in principle: a model trained on many tenants' verified behaviour carries behaviour across tenants. The design answer is that only behaviour, never tenant facts or policy, is a training target (S21); the audit answer is the privacy check that runs on every training set (S15).

Reward hacking and evaluator errors. An agent optimised against its own evaluator learns the evaluator. Mitigation: deterministic checks run before any model-based review; the reviewer can run on a different model than the worker; an escalate verdict forces a human; the frozen suite excludes its target actions from the prompt. Not solved: the reviewer is itself a model, a systematic blind spot in it becomes a systematic label error in the corpus, and today the reviewer runs on a minority of workspaces (S26).

Training-data feedback loops. A worker trained on its own trajectories can amplify its own habits. Mitigation: the target-action policy prefers replay-proven corrections over raw self-generated actions; S1's next curriculum is retention-heavy for exactly this reason (E11). Not solved: we have run one iteration.

Catastrophic forgetting. The subject of Section 1. Mitigation: the −2-point non-inferiority gate, the clean-heavy curriculum, and the fact that weights change rarely. Not solved: the gate detects movement on the suite and cannot detect forgetting the suite does not cover; at the current suite size it also cannot tell a real −4 from noise, so it will reject some candidates that did not forget.

Distribution shift and overfitting to one environment. Mitigation: the production-candidate gate requires fresh chronological tasks and shadow traffic, not only the frozen suite. Not solved: our corpus comes from one company's clients and five tools cover 84% of actions (S18).

Tool changes invalidating learned behaviour. A schema change makes a learned procedure wrong. Mitigation: tool schemas are runtime inputs, not training targets (S21); the guard's invalid-envelope kind catches the immediate symptom. Not solved: a memory rule that names a tool argument goes stale silently until a failure retires it.

9. Recursive system-level improvement

The loop is recursive because a verified failure improves more than one thing, and the things it improves are the things that produce the next generation of experience. For each edge we say who performs it today.

A failure improves memory: the correction or the failure signature is written where the next turn reads it. Performed by the agent, in the turn, automatically; consolidated by scheduled agents; editable by people (S5, S7). A failure improves tools and skills: a guard gains an anomaly kind, a schema gains a validator, a procedure gains a step. Performed by engineers, with agent assistance, through code review; the runtime does not modify its own guards. A failure improves evaluators: it becomes a frozen case in the next suite version. Performed by engineers. A failure improves the training curriculum: if it recurs, the corrected trajectory becomes a replay-proven example. Performed by the curation code under a written policy; the run itself is approved and funded by a person (E2, E11). Training improves the worker; a better worker generates trajectories with fewer of the old failures and a different set of new ones, and those become the next round's material. Run once; candidate rejected (E10).

That is what we mean by having built the feedback loop: every stage exists, the stages are connected by artefacts rather than intentions, the fast loop runs continuously without human action, and the slow loop has completed one full pass including its own rejection. Two of the edges are performed by people and will stay that way until an evaluator can be trusted to change an evaluator, which is a claim we are nowhere near making. It is not what Anthropic means by recursive self-improvement, and we are careful not to borrow that meaning.

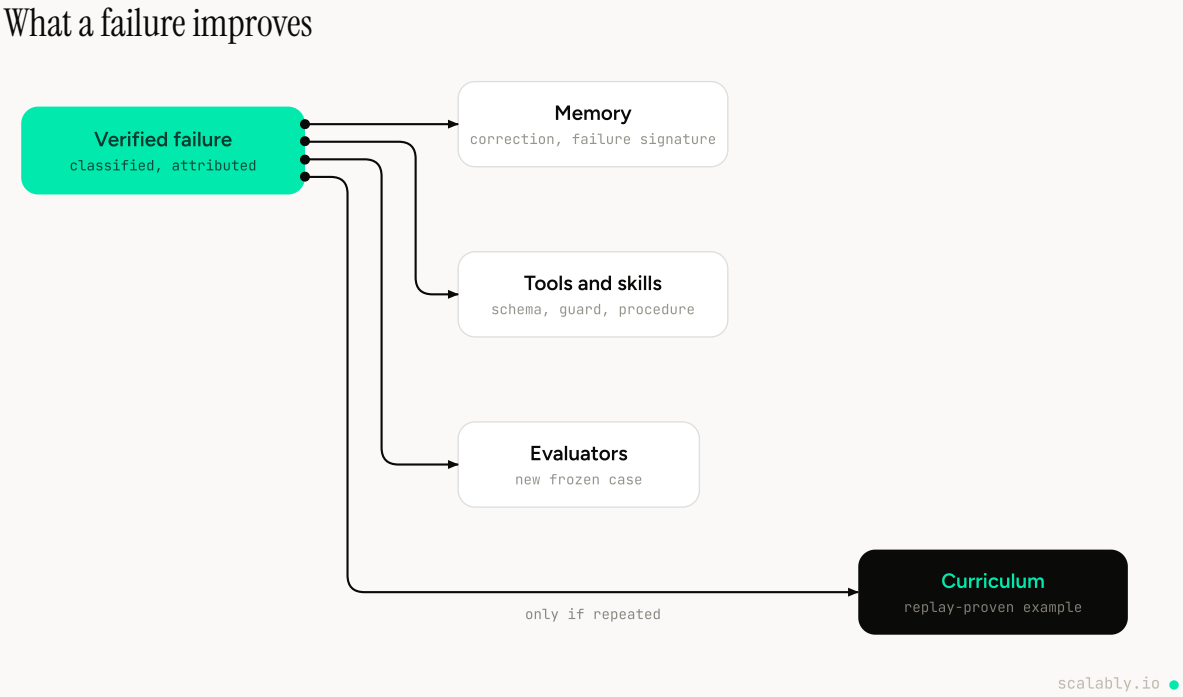


Figure 2. What a verified failure improves: memory, tools and skills, evaluators, and, only if it recurs, the training curriculum.

10. Experiments we intend to run next

Everything here is planned. It is listed so the next version of this report can be held to it, and the first item is the one that tests the thesis.

1. **The memory arm on the same 73 states.** Replay the 73 recovery states against the base model with one workspace-style rule injected ("if a tool call was just rejected, choose a different tool and say why") and with the workspace's actual memory injected, scored identically. If a rule recovers a comparable share of the 7/73 at zero training cost, it is the strongest evidence for the thesis in this paper; if it does not, the thesis needs the slow loop more than the title suggests. The same replay with and without memory on fresh tasks gives the fast loop's effect size. Retrieval quality (index coverage, missed reads) measured independently of task outcome.
2. **S1.1, retention-heavy.** A pilot of about 2,500 examples: 56% clean, 20% replay-proven recovery, 12% policy and skill, 12% negative or no-tool; one epoch, about 157 steps; the same frozen suite and gates (E11). The hypothesis is that the recovery gain survives and the clean-decision movement disappears; if it does, S1 was a curriculum error, and Section 2's argument rests on reversibility and attribution alone.
3. **A larger clean stratum.** Raise the clean stratum so that the −2-point gate has the resolution it claims, and report the discordant pairs for every gate decision.

4. **Deferred replay end to end.** Move the capture path from descriptors to sandboxed replay against a candidate, then gate promotion on replay regression plus the frozen suite (S13).
5. **Quantisation as a gate.** Evaluate the same candidate before and after NVFP4 conversion on the same frozen suite, so a regression is blamed on the right stage (S17, S22).
6. **Reviewer coverage.** Extend Stop-QA from 14 workspaces in shadow mode to a measured majority, and report reviewer-versus-human disagreement before any enforce mode is switched on (S26).
7. **Publish the counters.** Per-workspace memory size, rule churn (added, merged, retired per week) and retrieval reads per turn, so Section 5.1 becomes a time series rather than a snapshot.

11. Conclusion

We set out to make a production agent learn from its experience and found, on a small and exact experiment, that the obvious way to do it, changing the weights, produced a real gain and a possible loss in one artefact that we could neither separate statistically nor reverse separately. The architecture that followed is not novel in its parts. Frozen-backbone recursive improvement, memory distilled from trajectories, verification-gated writes and harness repair are all in the 2025 to 2026 literature, and we cite the work that demonstrated each. The runtime the agents live in is described in the Claude Agent SDK guide. What we add is the production discipline that connects them: a fast loop where every verified experience changes behaviour immediately through memory that is scoped, capped, provenanced and reversible, and that today holds 433 rules across 74 workspaces; and a slow loop where only repeated, evaluated experience earns a place in the weights, behind gates that were written before the first job and that rejected the first candidate.

Fast memory, slow weights. The system learns continuously. The model changes rarely, and only when the evidence says it should. Whether the fast half of that sentence earns its place is the first experiment on the list.

Acknowledgements and use of AI tools

This report was produced with substantial help from AI language models, and we report that use in full. The author set the question, the scope and the claims, supplied the production system, the data and the experiment, and approved the final text. Anthropic's Claude Opus 5 and Claude Fable 5.1, running as coding agents in Claude Code, drafted the text of every

section from the author's brief and the evidence register, wrote and ran the read-only scripts that extracted the production counts, recomputed every reported statistic from the raw evaluation records, and checked every arXiv identifier in the references against the arXiv API. The training and evaluation pipeline for the S1 experiment was built by OpenAI Codex coding agents (GPT-5.6 and GPT-6 family models) to a specification the author approved, and its promotion gates were fixed before training. Before publication, three separate model instances that were not shown the drafting reasoning reviewed the report: Claude Sonnet 5 for evidence and citations, Claude Opus 5 as a hostile expert reader, and Claude Sonnet 5 again to recompute every number after revision. Their findings changed the text, and the author chose the final title after the hostile reader flagged the first one as overclaiming. Claude Opus 5.5 prepared revision v1.1. No AI system is an author of this report. The author takes full responsibility for all of its content, including every number and every reference.

12. References

[R1] Z. Wang, M. Yan, J. Bi et al. MetaSkill-Evolve: Recursive Self-Improvement of LLM Agents via Two-Timescale Meta-Skill Evolution. arXiv:2607.05297, July 2026.

[R2] S. Ouyang, J. Yan, I-H. Hsu et al. ReasoningBank: Scaling Agent Self-Evolving with Reasoning Memory. arXiv:2509.25140, September 2025.

[R3] A. Behrouz, M. Razaviyayn, P. Zhong et al. Nested Learning: The Illusion of Deep Learning Architectures. arXiv:2512.24695, December 2025.

[R4] A. Behrouz et al. Titans: Learning to Memorize at Test Time. arXiv:2501.00663, December 2024.

[R5] X. Cheng et al. (DeepSeek). Conditional Memory via Scalable Lookup: A New Axis of Sparsity for Large Language Models. arXiv:2601.07372, January 2026.

[R6] B. Li et al. User as Engram: Internalizing Per-User Memory as Local Parametric Edits. arXiv:2606.19172, June 2026.

[R7] Z. Z. Wang et al. Agent Workflow Memory. arXiv:2409.07429, September 2024.

[R8] W. Xu et al. A-MEM: Agentic Memory for LLM Agents. arXiv:2502.12110, February 2025.

[R9] P. Chhikara et al. Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory. arXiv:2504.19413, April 2025.

[R10] C. Packer et al. MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560, October 2023.

[R11] J. S. Park et al. Generative Agents: Interactive Simulacra of Human Behavior. arXiv:2304.03442, April 2023.

[R12] N. Shinn et al. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv:2303.11366, March 2023.

[R13] G. Wang et al. Voyager: An Open-Ended Embodied Agent with Large Language Models. arXiv:2305.16291, May 2023.

[R14] A. Madaan et al. Self-Refine: Iterative Refinement with Self-Feedback. arXiv:2303.17651, March 2023.

[R15] X. Chen et al. Teaching Large Language Models to Self-Debug. arXiv:2304.05128, April 2023.

[R16] J. Li et al. Towards Self-Improving Error Diagnosis in Multi-Agent Systems. arXiv:2604.17658, April 2026; Findings of ACL 2026.

[R17] M. Chen et al. From Failed Trajectories to Reliable LLM Agents: Diagnosing and Repairing Harness Flaws. arXiv:2606.06324, June 2026.

[R18] C. Zhao et al. Debugging the Debuggers: Failure-Anchored Structured Recovery for Software Engineering Agents. arXiv:2605.08717, May 2026.

[R19] J. Ma et al. One Reflection Is Not Enough: Self-Correcting Autonomous Research via Multi-Hypothesis Failure Attribution. arXiv:2606.31478, June 2026.

[R20] M. Chen et al. Recursive Self-Improvement in AI: From Bounded Self-Refinement to Autonomous Research Loops. arXiv:2607.07663, July 2026.

[R21] Anthropic. When AI builds itself. anthropic.com/institute/recursive-self-improvement, accessed 20 September 2026.

[R22] E. J. Hu et al. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685, June 2021.

[R23] J. Kirkpatrick et al. Overcoming catastrophic forgetting in neural networks. arXiv:1612.00796, December 2016.

[R24] D. Rolnick et al. Experience Replay for Continual Learning. arXiv:1811.11682, November 2018.

[R25] Y. Luo et al. An Empirical Study of Catastrophic Forgetting in Large Language Models During Continual Fine-tuning. arXiv:2308.08747, August 2023.

[R26] H. Li et al. Revisiting Catastrophic Forgetting in Large Language Model Tuning. arXiv:2406.04836, June 2024.

[R27] G. O. Y. Laitinen-Fredriksson Lundstrom-Imanov. Mechanistic Analysis of Catastrophic Forgetting in Large Language Models During Continual Fine-tuning. arXiv:2601.18699, January 2026.

[R28] H. Chen et al. Continual Memorization of Factoids in Language Models. arXiv:2411.07175, November 2024.

[R29] H. Shi et al. Continual Learning of Large Language Models: A Comprehensive Survey. arXiv:2404.16789, April 2024.

[R30] R. Tiwari, K. Sareen, L. A. Agrawal, J. E. Gonzalez, M. Zaharia, K. Keutzer, I. S. Dhillon, R. Agarwal, D. Khatri. Learning, Fast and Slow: Towards LLMs That Adapt Continually. arXiv:2605.12484, May 2026.

Appendix A. Evidence rows

Every Scalably number and mechanism above cites a row of the form (Sn) or (En). The full register, with file paths, commits and the exact source quotes, is published alongside this report as `evidence.md`. All production metrics were re-read from their sources on 20 September 2026; the S1 per-stratum results were recomputed from the raw evaluation records with the official key-set scorer. Tests are exact two-sided sign tests on discordant pairs: 7 wins / 0 losses gives $p = 2 \times 2^{-7} = 0.0156$; 9 fixes / 1 regression gives $p = 2 \times 11 / 2^{10} = 0.0215$; 3 wins / 7 losses gives $p = 2 \times 176 / 2^{10} = 0.34$.

S rows: system evidence

ROW	CLAIM	SOURCE	CLASS
S1	Production corpus: 4,471 main sessions, 3,623 subagent runs, 485,070 records, 143,145 tool-use blocks, 143,043 tool-result blocks, 8,300 explicit `is_error:true`, five hosts, 2026-02-23 to 2026-08-24 (lower bound)	`nvidia-inception/evidence.md` E1; `field-report/evidence.md` FR47 · scalablyai	OBSERVED
S2	Model mix of main sessions: 3,811 Claude-only, 421 Qwen-only, 6 mixed, 154 DeepSeek, 79 other	E2 (`PROJECT_HANDOFF.md:120-124`) · scalablyai	OBSERVED
S3	The corpus is a causal graph: uuid→parentUuid, tool_use.id→tool_result, subagent sidecars, compaction boundaries, externalized tool results	E4 (`PROJECT_HANDOFF.md:156-177`) · scalablyai	IMPLEMENTED
S4	Fast memory is plain files per workspace: `memory/` tree (index.md, profile, people/, projects/, clients/, reference/, daily/) + `rules/learned-corrections.md` ; essentials injected every turn, detail read on demand	`container/prompts/baseline/global-rules/memory-proactive.md` ; `container/agent-runner/src/index.ts:1240-1248` ; public `skills/memory-system/SKILL.md` · scalablyai; agent-skills	IMPLEMENTED
S5	Corrections enter under `## Pending` as one line ≤160 chars with a stable id (`<!-- lc:slug ⟶ )`; "Pending rules are IN FORCE from the moment they are written"; cap 60 rules, consolidate at the cap	`memory-proactive.md` ; `memory-system/SKILL.md` ("Cap: 60 rules"; "Pending … IN FORCE") · agent-skills	IMPLEMENTED
S6	Provenance per rule: `reference/corrections-log.md` records rule (current), history with dates, evidence type, scope, source quote, wrong/right example; retired rules move to `## Retired` with the superseding id and date	`memory-system/SKILL.md:158, 186-200` · agent-skills	IMPLEMENTED

ROW	CLAIM	SOURCE	CLASS
S7	Consolidation cadence: nightly task integrates and dedupes Pending; Sunday task archives the week, merges/compresses rules, verifies staleness, rebuilds index; "MOVES, not deletions"; ends by running `check-memory.sh` ("Caps are numbers, not vibes")	`container/prompts/tasks/weekly-memory-cleanup.md`; `memory-system/SKILL.md:83` · scalablyai; agent-skills	IMPLEMENTED
S8	Memory is tenant-scoped: one `memory/` tree per group (workspace); learned-corrections are appended to the group's own file and injected as a `<standing_rules>` footer only when non-empty	`index.ts:1140, 1300, 1377-1399` · scalablyai	IMPLEMENTED
S9	Memory is reversible and attributable: files live in per-server git config repos (history), rules carry ids and a provenance ledger; retirement is a move to `## Retired`	S6; config repos `scalably-configs/<server>` · scalablyai	IMPLEMENTED
S10	Memory write is fact-replacing, not append-only: "A changed fact REPLACES the old line — never append a contradicting second version"	`memory-proactive.md` · scalablyai	IMPLEMENTED
S11	Tool-outcome guard classifies every tool result into 8 anomaly kinds before the model sees success: `empty_result`, `failure_in_success_channel`, `invalid_envelope`, `failed`, `blocked`, `partial`, `pending`, `no_op`	`container/agent-runner/src/hooks/tool-outcome-guard.ts:12-21` · scalablyai	IMPLEMENTED
S12	Stop-QA: deterministic checks first (`unresolved-placeholder`, `unacknowledged-tool-error`, `count-mismatch`), then an evidence-bound reviewer (roles `qwen`, `large`); modes `off / shadow-qwen / shadow-dual / shadow-cascade / enforce-cascade`; `escalate` verdict forces human	`hooks/qa-deterministic.ts:26-80`; `hooks/stop-qa-verification.ts:246-354`; FR43 · scalablyai	IMPLEMENTED

ROW	CLAIM	SOURCE	CLASS
	disposition; off by default, enabled per workspace		
S13	Deferred replay capture exists in `capture` mode only: "does not run Qwen, fetch tools, retain raw prompt or result text, call a teacher or authorize training. Unknown modes fail closed"; default off; explicit group allowlist; sample rate default 0	`docs/reference/DEFERRED-REPLAY-SHADOW.md:13-16` and config table · scalablyai	IMPLEMENTED (capture) / PLANNED (replay)
S14	Training boundaries: production logs read-only and outside git; raw tenant data and credentials never enter the training repo; production rig never used for training; shadow-mode data excluded from train/eval until explicitly approved; typed redaction and tenant-scoped pseudonyms; Claude-origin targets blocked from training pending legal review	`scalably-agent-posttraining/README.md:7-13` ; E11, E12 · posttraining; scalablyai	IMPLEMENTED (policy, enforced by pipeline)
S15	Frozen evaluation EV1: 60 balanced training-sanity decisions + 424 task-disjoint full-context decisions (264 positive, 160 known-failure states) across 68 clusters and 22 current-schema tools; 484 request contracts exclude their target action; 228 long-context states deferred intact	`KNOWLEDGE_LEDGER.md` ("S1 evaluation v1 is frozen") · posttraining	IMPLEMENTED
S16	Serious-candidate gate: ≥ +5 absolute points paired decision correctness; ≥ 20% relative reduction in targeted failure families; paired CI supports a positive effect; no new critical rule/privacy/security failures; clean-task non-inferiority no worse than −2 points; no increase in unsupported completion; isolated replay completion does not regress; overhead ≤ 15% without compensating quality; confirmation checkpoint or seed reproduces the gain	`docs/training-program-v1.md:223-233` · posttraining	IMPLEMENTED (gate definition; applied to S1)

ROW	CLAIM	SOURCE	CLASS
S17	Production-candidate gate, six conditions: gain survives NVFP4 conversion; fresh chronological tasks confirm direction; isolated live replay passes; shadow traffic improves on fresh tasks; low-risk canary and measured cohort show no safety/reliability regression; old production checkpoint remains immediately rollbackable	`docs/training-program-v1.md` §7 · posttraining	PLANNED (no candidate has reached it)
S18	Eligible training pool (frozen policy audit): 14,907 decisions (8,755 clean, 2,506 verified actions, 3,650 recoveries) from 1,883 sessions / 707 clusters; five tools dominate ~84% of actions ("Coverage, not volume, is limiting")	`KNOWLEDGE_LEDGER.md` · posttraining	OBSERVED
S19	Policy surface that must NOT be frozen into weights: two model prompt families, 118 active group prompts, 35 global rule files, 40 admin rules; "never freeze current policy text as weight-level truth"	`KNOWLEDGE_LEDGER.md` · posttraining	OBSERVED (counts) / design principle
S20	Deterministic playground MVP: content-free current-policy snapshot, 100 scenarios, 27 stateful tools, six policy families, 100/100 oracle + broken-to-corrected same-state replay passes; no scenario training-authorized	`KNOWLEDGE_LEDGER.md` · posttraining	IMPLEMENTED (MVP)
S21	Weights learn behaviour, environment supplies changing knowledge: "Mutable rules, skills, tool schemas, and world knowledge remain versioned runtime inputs. The model is trained to follow current policy and verify current facts, not to memorize changing policy or global knowledge."	E6 (`scalably-agent-posttraining/README.md`) · posttraining	design principle (documented 2026-08-24)

ROW	CLAIM	SOURCE	CLASS
S22	Quantization path (planned): source → LoRA/QLoRA → high-precision eval → NVFP4 → post-quant eval → optional QAT/QAD → vLLM	E8, FR46 · posttraining	PLANNED
S23	Production inference tier that the slow loop targets: `unsloth/Qwen3.8-27B-NVFP4` on 2× RTX 5090, vLLM 0.27.0, 262,144 context; 28,097 requests in 13.9 days (2026-08-29 11:54 UTC to 2026-09-12 10:15 UTC), prefix-cache hit 712,833,184 / 863,463,591 = 82.55%, 0 engine errors	field report FR1-FR47; scalably.io/blog/qwen3-8-27b-nvfp4-rtx-5090-production · scalablyai	OBSERVED
S25	Fast loop in numbers (read-only counts on the five hosts, 2026-09-20; no content read): 74 workspaces hold a `rules/learned-corrections.md`; 433 rules with ids; 53 under `## Pending`; 58 workspaces hold `memory/reference/corrections-log.md`; 37 retired rules; 199 `memory/` trees; 3,422 memory markdown files; 438 touched in the last 7 days; largest single rule file 47 rules (cap 60). Per host (rules / pending / retired), hosts anonymised: A 243/37/7, B 152/16/22, C 0/0/0 (different consolidation variant), D 9/0/3, E 29/0/5	`/opt/scalably/groups/*/{rules,memory}` counted via ssh with a metadata-only script · servers	OBSERVED
S26	Stop-QA coverage: 14 of 171 registered workspaces have a mode set (13 `shadow-qwen`, 1 `shadow-cascade`); none in an enforce mode; the deterministic tool-outcome guard runs everywhere	`sqlite3 store/messages.db "SELECT value,COUNT(*) FROM group_env WHERE key='SCALABLY_STOP_QA_MODE'"` + `registered_groups` count, per host, 2026-09-20 · servers	OBSERVED
S24	Operational runs across the fleet: 3,821 scheduled agent runs in 30 days, 96.3% finished without error, 169 workspaces, 5 hosts (as of 2026-09-12)	`content/kpi/latest.json` · scalably-content	OBSERVED

E rows: the S1 experiment

ROW	CLAIM	SOURCE	CLASS
E1	Two rank-16 LoRA jobs on a rented service: S1-F (job id withheld), S1-Q (job id withheld), 54/54 steps each, 2 epochs, 423 records / 51,333 target tokens per arm	arms: Q-only (Qwen-native targets) and mixed (F)	arms: Q-only (Qwen-native targets) and mixed (F)
E2	Training cost $53.882484 (cap $60); evaluation ~$10.47 on a rented H100 NVL ($2.6817/h incl. storage), instance destroyed and verified absent after	off-rig by rule (S14)	off-rig by rule (S14)
E3	Mean training loss fell 18.21% (S1-F) and 16.33% (S1-Q) between epochs: "proves optimization movement only"	not a quality claim	not a quality claim
E4	Evaluation on exact BF16 Qwen3.8-27B, vLLM 0.27.0, 32K context, deterministic paired scoring; transport/parser failures zero		
E5	Training sanity (60): base 34, Q-only 38, mixed 37		
E6	264 held-out positive decisions: base 138, Q-only 141 (+1.14 points), mixed 139 (+0.38); both below the +5 gate	S16	S16
E7	Verified recovery: 31/73 → 38/73 (Q-only); 7 adapter-only wins, 0 base-only; exact paired p = 0.015625	the "learned" half	the "learned" half
E8	Clean-decision retention: 49/73 → 45/73 (Q-only); fails the −2-point non-inferiority margin	the "forgot" half	the "forgot" half
E9	Exact failed-action repetition on 160 known-failure states: 32 → 24 (both adapters), 20% → 15%, 25% relative; paired 9 fixes / 1 regression, p = 0.021484375; Q-only also cut failed-tool repeats 122 → 114		

ROW	CLAIM	SOURCE	CLASS
E10	Verdict recorded: "behavioral movement and targeted recovery learning are proven, but neither adapter is a serious or production candidate"; adapters not deployed	the rejection	the rejection
E11	Recommended next: S1.1 pilot of ~2,500 examples (56% clean, 20% replay-proven recovery, 12% policy/skill, 12% negative/no-tool), one epoch (~157 steps), before any 6,000-8,000-example run	PLANNED	PLANNED
E12	Scoring rule: a decision is correct when the chosen tool equals the recorded tool AND the set of argument keys (nested, path-qualified) equals the recorded set ("joint"); tool-only accuracy reported separately. Strata of the 264 positives: `verified_recovery` 73, `clean_trajectory_action` 73, `verified_action` 118 (disjoint). The 160 known-failure states share no decision with the 73 recovery states	`scripts/summarize_eval_matrix.py` (argument_keys, paired); `artifacts/s1-evaluation-v1/ev1-positive-v1.jsonl`; recomputed 2026-09-20 from `artifacts/s1-eval-results-v1/*.jsonl` · posttraining	OBSERVED
E13	Per-stratum paired results, joint correctness, exact two-sided sign test on discordant pairs. Q-only: recovery 31→38 (+7/−0, p = 0.0156); clean 49→45 (+3/−7, p = 0.34); verified_action 58→58 (+4/−4); all 264: 138→141 (+14/−11, p = 0.69). Mixed: recovery 31→36 (+7/−2, p = 0.18); clean 49→46 (+5/−8, p = 0.58); verified_action 58→57 (+5/−6); all 264: 138→139 (+17/−16). Tool-only accuracy unchanged in every stratum for Q-only (61/61, 54/54, 89/89; 204/204 overall)	recomputed 2026-09-20 with the official key-set rule; matches `positive-summary.json` strata totals · posttraining	OBSERVED
E14	Hyperparameters (both arms): Qwen/Qwen3.8-27B base, SFT LoRA r=16, alpha=32, dropout 0, target modules q/k/v/o/gate/up/down, LR 5e-5 cosine, warmup 0.05, batch 16, max sequence 32,768, packing, seed 42, 2 epochs, 54 steps	(provider manifest, path withheld) · posttraining	OBSERVED

Appendix B. What this report does not claim

It does not claim that Scalably solved recursive self-improvement, continual learning, catastrophic forgetting or anything resembling AGI. It does not claim the architecture is optimal, that the fast loop's effect on outcomes has been measured, or that the clean-decision movement in S1 was forgetting rather than noise. It reports one rejected experiment with its discordant pairs, one implemented architecture with the people in it named, and the experiments that would test the parts still unproven.