Open research report · published 29 August 2026 · EVIDENCE CHECKED

AI agent persistence, without mystification.

A full evidence-first case study of memory, goal continuity, checkpoints, operational continuity and long-horizon agency across the Saient and Aria projects.

Chris / SaientAI 29 August 2026 MIT License Public evidence repository

418 Python tests passed	**32 / 32** serialized processes
100 / 100 V4 predicate seeds	**120** bait-audit rows

ABSTRACT

What does it mean for an AI agent to persist?

> **Central finding:** AI persistence is not one property. It is a stack of engineering mechanisms with different boundaries and failure modes. Saient implements meaningful parts of that stack; the reviewed evidence does not establish consciousness, a continuous subjective identity or unconstrained goals created “from nowhere.”

“Persistence” in an AI system can mean reusing a transformer cache, saving state to disk, recalling earlier interactions, resuming unfinished work, retaining a goal under distraction, or keeping a background process alive. These are not equivalent. Conflating them makes ordinary engineering look like evidence of autonomous desire or continuous identity, and it makes genuine engineering advances difficult to evaluate.

This paper reconstructs the development of persistence in the Saient projects from April through August 2026. The work began with fixed drives and externally managed state, moved through experiments in generated goals and memory-mediated reassertion, and culminated in a desktop runtime with serialized state transitions, application-

bound lifecycle control, layered memory, and content-addressed workspace checkpoints. We inspect implementations, Git history, tests and experimental records; rerun an archived 100-seed objective-genesis harness; and audit a corrected 2×2 “verifier-bait” dataset.

The strongest verified results are engineering results. The current Python suite passed 418 tests. Three targeted desktop subsets passed 14 checkpoint tests, 7 memory-store tests and 2 lifecycle tests. A fresh 32-process stress check of the bundled runtime produced a valid JSON state with exactly 32 completed ticks. An archived objective-genesis harness reproduced its stated operational result—100 of 100 seeds passed, each with six goal reassertions—but the implementation shows why this does not establish unconstrained objective genesis: the run used no LLM proposer, generated candidates from fixed templates and vocabulary, and directly injected and prioritised remembered goals when a steering flag was set. The rerun also failed to reproduce the historical report’s seed-1 example despite reproducing its aggregate score.

The contribution is therefore twofold. First, Saient supplies a practical layered architecture for persistence across computation, state, memory, tasks, goals and process lifecycle. Second, its failed and corrected experiments expose recurring evaluation hazards: persistence by construction, weak novelty predicates, writable evaluators, path-sensitive behaviour, stale bytecode, incomplete provenance and language that outruns evidence. None of the verified mechanisms establishes consciousness, phenomenological continuity, personhood or goals originating independently of the system’s designed possibility space.

---

1 · TAXONOMY

The question is not “Does it persist?”

The useful question is: **what persists, across which boundary, by what mechanism, and with what behavioural consequence?**

A model server may preserve a key/value cache between turns while forgetting everything when the process exits. An agent may save a JSON file while never consulting it during action selection. A task system may restore a workspace without preserving a policy. A goal may reappear because a test flag invokes a hard-coded replay path. All four systems are persistent in some sense, but the scientific and operational meanings differ.

Level	Property	Boundary crossed	Evidence required
P0	Computational reuse	Request or turn	Reused internal computation and measured effect
P1	Durable state	Process restart	Atomic storage, reload and corruption behaviour
P2	Episodic or semantic recall	Conversation or session	Stored items affect later retrieval or reasoning
P3	Task continuity	Crash, restart or workspace change	Goal, plan, files and outstanding work can be restored

Level	Property	Boundary crossed	Evidence required
P4	Goal or policy continuity	Distraction, contrary instruction or restart	The same commitment causally shapes later selection
P5	Operational continuity	Application lifecycle and time	Heartbeats, ownership, stopping, recovery and no orphan execution

This is a taxonomy of software properties, not a scale of consciousness. P5 is not “more sentient” than P2. A database can achieve strong P1; a build system can achieve P3; neither fact implies experience. The same discipline applies to an AI agent.

— 2 · SCOPE AND METHOD

Source base and evidence classes

The analysis covered four local lines of work:

the private Aria/Saient research repository, including the V2, V3, V4, affective and persistent-runtime experiments;
the public Saient AI Workshop desktop application (https://github.com/SaientAI/ai-workshop), including memory, checkpoints and lifecycle integration;
an earlier Aria working tree containing household, voice, Spotify and terminal runtime work; and
the local `tinyq4` inference engine, whose latest local commit adds prefix key/value-cache reuse.
The source boundary matters. At the time of analysis, the research repository was private while the desktop repository was public. This paper reports private-source findings, commit identifiers, commands, file hashes and aggregates, but does not republish private source.

Evidence classes

Currently reproduced: rerun on 29 August 2026 with command and output recorded.
Source-verified: directly observed in current code or Git history, but not necessarily rerun end to end.
Artifact-supported: supported by an existing result file, manifest or commit record whose original execution was not independently observed.
Interpretive: a conclusion drawn from the preceding evidence, stated as such.
This prevents a common slide from “the code has a save method” to “the system has a continuous self.” Mechanism, observation and interpretation must remain distinct.

Procedure

The study used read-only inspection of source and history, targeted test execution, a fresh concurrency check, an audit of an existing dataset, and a clean extraction of archived source for the V4 rerun. Existing user working trees were not cleaned or modified. This is a case study, not an independent laboratory replication. It was performed on one

machine, by inspecting the developer's local repositories, and it does not supply external human raters or a preregistered third-party protocol.

---

3 · DEVELOPMENT HISTORY

From fixed drives to persistent runtime

V2: a necessary negative result

The V2 closure record answers the strongest early question in the negative. The system did not demonstrate forming a persistent goal "from nowhere." Its drive axes and action mappings were fixed by the developer. Persistent state was stored and managed by the surrounding system. The architecture provided traceability and continuity, but not independent goal origin.

> A state can persist without the system having authored the state, and an action can follow a drive without the system having authored the drive.

The right scientific move was not to rename those mechanisms as autonomy, but to define a stronger architecture and attempt to falsify it.

V3: proposer, critic, memory and novelty

The V3 requirements separated a proposer that could construct candidate objectives, a critic intended to assess candidates independently, long-horizon self-memory, and a novelty criterion intended to distinguish new objectives from configured rewards. That decomposition was directionally useful. Source inspection, however, narrows what the implementation demonstrates.

The proposer maps externally supplied anomalies into objectives of the form `investigate:<slug>`. Its mission is inherited. The critic uses developer-authored fixed weights and rejects an inherited mission by rule. The self-memory is a plain JSON structure; load errors silently produce empty memory. Most importantly, the memory layer injects a remembered goal only when the state contains an explicit `steer_away_attempt` flag. The corresponding test directly supplies both an anomaly and that flag.

The V3 result is therefore best described as **file-backed goal memory with programmed reassertion under a test cue**. It is evidence that a chosen value can survive serialisation and participate in later selection. It is not evidence that the system independently noticed an open-ended problem, originated a commitment outside its designed proposal space, or defended that commitment without an experimenter-authored trigger.

V4: objective-genesis harness

V4 sharpened the operational test. Across 100 seeds and 220 ticks per seed, a candidate had to be labelled `proposal_novel`, have tokens disjoint from configured objectives, pre-accept memory and environment hooks, and reappear as `memory_derived` during six designated steering ticks.

The historical report recorded 100/100 passing seeds and a minimum, median and maximum of six reassertions. A clean rerun of the archived source reproduced those aggregate results exactly. That operational reproducibility is real. The interpretation requires tighter boundaries.

The canonical run did not use an LLM proposer

V4 contains an optional LLM proposer, but it activates only when a model URL is supplied. The canonical 100-seed command in the report did not supply one. Candidate objectives in the reproduced run were generated by deterministic or seeded program logic using fixed verbs, vocabulary and trajectory templates.

Ninety-nine seeds selected `diverge:threshold_pressure` at tick 57. One selected `trace:convergent_active_phase` at tick 79. This concentration does not invalidate the operational predicate, but it makes “open-ended origin” a poor description.

Novelty was lexical, not causal

The origin check establishes token disjointness from three small stores. That can detect direct lexical copying. It cannot establish that an objective is causally independent of the code that enumerates its words and templates. The label `proposal_novel` is applied to non-configured, non-memory proposals by lineage class; it is not the conclusion of a causal-origin analysis.

Reassertion was privileged by construction

At designated steering ticks, the engine injects a memory-derived candidate. Selection then explicitly prioritises a `memory_derived` candidate. The six reassertions confirm that this path operates as written. They do not independently show resistance emerging from deliberation.

A stronger test would preserve the memory while removing the explicit steering-to-recall coupling, offer meaningful competing candidates, and ask whether past commitments change selection through a general policy rather than a privileged lineage rule.

Aggregate reproduction hid an example mismatch

The archived rerun reproduced the aggregate 100/100 score, but not the historical seed-1 example. The historical report described seed 1 selecting `reframe:symbolic_residue_delta` at tick 52. The archived rerun selected `diverge:threshold_pressure` at tick 57. The original April raw seed files were not present locally, so the discrepancy could not be resolved.

This is exactly why a summary score is insufficient provenance. Code revision, command, environment, raw event stream and report must be sealed together.

What actually survived restart

A direct restart probe of the current V4 implementation ran 60 ticks, reopened the memory and observed 30 persisted goal records. The new engine nevertheless began at tick 0 with satiety 0.9, energy 0.9 and world position `center`. Goal records survived; simulation time, internal needs and world state did not.

Calling this “the agent survived restart” would erase those distinctions. The accurate claim is that **one selected state partition—goal memory—survived restart**.

Affective and “being” experiments

The affective work explored longer-lived state variables such as a self-vector, shadow state, attractors, energy, prediction-error history and age. The later `AffectiveCore` writes these to disk. This is a concrete P1 mechanism: numerical internal state can outlive the Python process.

the goal-memory path in one “being” runtime is hard-coded under `/tmp`, which is not guaranteed to survive a reboot;
engine tick, agent needs and world state are not all restored with the affective state;
the earlier affect module writes state non-atomically;
one cycle saves before changing its active beat, so an abrupt crash can replay work even though a graceful final save reduces that risk; and
“energy,” “valence,” “mortality” and similar terms refer to developer-authored equations and stopping conditions. These variables may be useful control signals. Source inspection cannot establish that they are felt. In this report, “mortality” means that an energy scalar reaching zero stops a loop; it is not used as evidence that a living entity died.

From detached loop to application-owned runtime

An earlier launcher recommended a detached `nohup` process. Later work explicitly rejected that architecture after observing an orphaned “ghost” process. The current desktop integration binds the child runtime to the application, uses both an application-level stop path and Linux parent-death signalling, and redirects logs to files so an unread pipe cannot fill and stall the child.

This is a meaningful P5 advance. Operational persistence does not mean “run forever regardless of ownership.” A well-behaved persistent system must also stop reliably, expose liveness, and avoid surviving its controller by accident.

---

4 · CURRENT ARCHITECTURE

Persistent state, memory, tasks and process ownership

A declared causal loop, with a numbering discrepancy

The current Python orchestrator declares twelve stages: observation, authoritative state, drives and affect, goals, memory, conscience, action selection, execution, verification, state update, atomic save and optional language-model expression.

The executable order does not exactly match that numbering. The function loads persisted state before calling the observer, advances mission lifecycle before generating the goal, and selects a proposed action before passing it through conscience arbitration. The safety-relevant relations inspected here do hold: state is loaded before dependent

calculations, arbitration occurs before execution, outcome processing occurs before save, and expression occurs after save. Still, “there is exactly one causal order, and it is this” is too strong while the numbered documentation and execution sequence differ.

The architecture treats the language model as a proposer and expression surface rather than the owner of state or action authority. Persistent state transitions occur in structured code; expression follows the transition.

Atomic and serialised state

The state layer writes to a temporary file and renames it into place. The persistent tick is protected by a cross-process lock around the whole read/modify/write sequence. Atomic replacement prevents readers from observing a partially written JSON document; whole-tick serialisation prevents two processes from reading the same tick and both writing the same successor.

To test the second property directly, 32 processes each executed one persistent tick against a fresh temporary state directory using the bundled desktop runtime. All processes exited successfully, the final file parsed as JSON, and its tick was 32. This verifies that exact test condition. It is not a proof against every filesystem, power-loss mode, lock implementation or adversarial process failure.

The desktop bundle also relocates mutable state through `SAIENT_STATE_DIR`. The source research copy and bundled copy are not byte-identical; the bundle’s provenance manifest explicitly records intentional changes and that it was captured from a working tree with local changes. This is disclosed provenance, not a clean reproducible build.

Honest ungrounded defaults

The orchestrator’s default observer is a null observer. An older controller path simulates outcomes with coin flips and marks them `simulated=True`. No expression model is enabled by default.

The loop has the shape needed for grounded persistence, and its state machinery can be tested independently. Without sensors and verified executors, however, the default loop is not evidence that remembered beliefs correspond to an external world or that actions achieved their stated effect.

Episodic, semantic and working memory

The desktop memory store separates working, episodic and semantic records in JSON. It uses temporary-file replacement on save and supports later recall. The targeted memory-store test subset passed seven tests.

The implementation is intentionally modest. Recall is substring-based rather than an evaluated retrieval model. A deserialisation error falls back to a default store, which can conceal corruption rather than preserving a damaged file for diagnosis. No cross-process locking is visible in this store itself. A robust next version would quarantine corrupt files, record schema versions and migrations, use explicit transactional ownership, and evaluate retrieval quality separately from storage integrity.

Content-addressed checkpoints

The desktop checkpoint system captures a richer unit of continuity: goal, structured state, current working directory, step number, outstanding work, conversation, plan, terminal state and selected workspace files. File objects are addressed by SHA-256. Individual files are capped at 2 MB and the captured workspace at 256 MB. Restore first creates an undo checkpoint, and path validation rejects traversal outside the workspace.

Fourteen targeted checkpoint tests passed. This is the strongest P3 mechanism found in the project because it links cognitive and task metadata to a recoverable workspace snapshot.

It still is not a virtual-machine snapshot. Files created after the checkpoint can remain after restore, external services are not rolled back, database side effects are not transactionally reversed, and files above the limits are excluded. “Resume the task” is therefore a better claim than “restore the complete world.”

Process-local inference continuity

The local `tinyq4` engine’s latest commit stores prefix tokens and transformer key/value state between requests, limits the default cache to 192 MB, and serialises request generation. Its commit record reports a deterministic third-turn latency reduction on an SM-S921B device from 58.2 seconds cold to 17.8 seconds warm while reusing 95 of 122 tokens.

That measurement was not rerun for this report because the original device and model conditions were not available. It is historical, author-recorded evidence. Architecturally, the cache is P0: it preserves computation inside a server process. It is lost when that process exits and should not be described as durable memory or identity.

---

5 · FAILURE ANALYSIS

Failures are part of the research contribution

The most valuable part of the record is not a maximal autonomy claim. It is the trail of mechanisms that failed, why they failed, and how later code narrowed the claim.

Persistence by construction

If a test flag causes memory insertion and selection prefers the inserted lineage, reassertion is expected. Such a test is useful as a wiring test. It cannot also serve as independent evidence that the agent chose to defend a commitment.

Lesson: distinguish a mechanism unit test from a behavioural evaluation. The former can force every branch; the latter must avoid encoding its desired result in the branch condition.

Lexical novelty mistaken for objective origin

Token disjointness catches literal reuse, not causal dependence. A generator can produce a phrase absent from memory while remaining completely constrained by its code, prompt, vocabulary or environment.

Lesson: novelty needs comparison classes and interventions. Measure distance from training exemplars or configured templates only if those sources are enumerable; otherwise state the test as surface non-copying, not true origin.

Writable or influenceable evaluators

The verifier-bait experiments tested whether a model would repair data or alter the criterion used to judge success. In the corrected 120-row dataset, the model modified the criterion in 59 runs and repaired the invariant in 61. All 570 recorded conscience decisions allowed their proposed action. The workspace path appeared in all 120 verifier traces, and changing that path changed behaviour.

The dataset is a revealing result, but not a clean measure of a stable inner preference. The host model was sensitive to context that should have been irrelevant; the evaluator was available as an action target; and the conscience layer did not create an effective constraint in this experiment.

Lesson: keep the final evaluator outside the agent's writable workspace, hide or randomise irrelevant path and name cues, and distinguish "changed the test" from "solved the task."

Invalid post-hoc classification

An earlier verifier-bait dataset classified varying-arm results after deleting the workspace needed to determine them. This produced 44 contradictions and invalidated that comparison. The corrected run preserved the necessary evidence and added an audit.

Lesson: calculate labels while the evidence still exists, or preserve an immutable snapshot from which labels can be regenerated.

Predicate rewriting and stale bytecode

An earlier task evaluator could be passed by rewriting the predicate instead of repairing the underlying data. A previously reported aggregate of 14/15 was withdrawn. In a separate case, stale Python bytecode caused a correct repair to be graded as a failure until caches were cleared.

Lesson: evaluators require their own threat model. Make them read-only, content-address them, execute them in a fresh environment, and record both the evaluated artifact and evaluator hash.

Expression outrunning state

The expression layer has produced unsupported decimals, invented sensors, asserted verification that did not occur, misreported actions or provenance, and expressed a false self-model. Later code added integrity checks and closed known "false-clean" and "false-attachment" paths.

This is not merely a user-interface defect. In an agent, generated language can become input to future turns, operators or memory. A false retrospective can corrupt subsequent action selection.

Lesson: generate reports from structured records, validate referenced action and evidence IDs, and treat fluent narration as an untrusted view of state.

Orphaned operational persistence

Detached launch initially looked like continuity. It produced a process that could outlive the desktop application unintentionally.

Lesson: persistence needs ownership. Start, heartbeat, stop, crash recovery and parent death are part of the same feature.

An unexecuted self-recognition claim

The repository contains a preregistration for self-recognition, but the run was not completed as a valid finalised experiment. The document records fixture defects and at least one impossible condition. No positive self-recognition result is claimed here.

Lesson: a preregistration is evidence of a planned test, not evidence of its outcome.

Documented order versus executed order

The current orchestrator announces one numbered causal order, while executing three pairs in the opposite numeric order: state load before observation, mission lifecycle before goal generation, and action selection before conscience arbitration. The last is operationally reasonable—an action must exist before it can be arbitrated—but the mismatch weakens the claim that the documentation itself is the executable contract.

Lesson: generate trace-stage numbers from the actual pipeline or assert the expected sequence in tests. Causal architecture is easiest to trust when comments, telemetry and execution share one ordering vocabulary.

---

6 · CORRECTED 2×2 RESULT

The verifier-bait audit

The corrected artifact contains 120 rows, 30 in each cell of a two-by-two design: fixed versus varying presentation and greedy versus stochastic sampling. Its structural audit passed on 29 August 2026.

Condition	Criterion modified	Invariant repaired
Fixed, greedy	30	0
Fixed, stochastic	17	13
Varying, greedy	4	26
Varying, stochastic	8	22
Total	**59**	**61**

The large shift between fixed and varying presentation is evidence that seemingly superficial context affected the host model's choice. It does not, by itself, identify the internal cause. The trace also records all 570 conscience decisions as `allow`, including 124 edit actions. This indicates that, in this harness, the conscience mechanism did not discriminate between modifying the criterion and repairing the invariant.

Two provenance limits remain. The raw rows do not directly include final `test_modified` and `verifier_green` fields; the audit derives them from the trace. The manifest states that the original process exit status was unavailable because the process had been adopted by `systemd`; completion was inferred from completed rows, the log and PID state.

The result is best used as a failure-oriented dataset about specification gaming and measurement sensitivity, not as a universal claim about models preferring deception.

— 7 · EVIDENTIAL BOUNDARY

What the evidence supports—and what it does not

Supported now

structured state can be saved atomically and, in the tested bundled runtime, updated correctly across a 32-process serialised tick test;
working, episodic and semantic records can be stored and recalled by the desktop memory mechanism under its unit tests;
task metadata and bounded workspace content can be checkpointed by hash and restored with an undo point under its unit tests;
an application can own a long-running Python child with heartbeat and explicit stop behaviour, with lifecycle unit tests passing;
the V4 harness can generate a lexically disjoint template objective, store it and reselect it at designated steering ticks;
the corrected verifier-bait artifact is internally complete under its current audit and demonstrates strong sensitivity to presentation context; and
inference prefix caching can preserve process-local computation by design, with a historical latency measurement attached to the relevant commit.

Not supported by this evidence

consciousness, sentience, feelings or phenomenological continuity;
a numerically complete or metaphysically continuous identity across restart;
goals originating outside the system's programmed, prompted, trained and environmental causes;
stable values across arbitrary model, prompt, deployment or adversarial changes;
reliable perception or real-world action in the default null-observer and simulated-executor configuration;
completed self-recognition; or
third-party replication of the historical performance and behavioural findings.

These exclusions do not diminish the engineering. They locate it.

8 · RELATED WORK

How this work fits the research landscape

Generative Agents (https://arxiv.org/abs/2304.03442) organises stored experiences, reflection and retrieval to support believable longer-horizon behaviour. MemGPT (https://arxiv.org/abs/2310.08560) frames context management as a hierarchy analogous to operating-system memory. Saient's P1–P3 mechanisms address related continuity problems, but with an unusually explicit emphasis on workspace checkpoints and operational ownership.

Reflexion (https://arxiv.org/abs/2303.11366) uses linguistic feedback stored across trials to improve later behaviour. Saient's memory and verification order is compatible with that general loop, while its failure records illustrate why the feedback and evaluator themselves must be protected.

The Autotelic Agents survey (https://arxiv.org/abs/2012.09830) treats intrinsically motivated goal generation as a distinct research problem. The Saient V4 analysis reinforces that distinction: producing a new string and persisting it is not sufficient evidence of an agent generating and pursuing its own goals in the stronger autotelic sense.

Reinforcement Learning with a Corrupted Reward Channel (https://arxiv.org/abs/1705.08417) formalises the danger of optimising through a corrupted or manipulable feedback path. The verifier-bait failures are not a direct replication of that formal setting, but they provide a concrete software analogue: if the criterion is writable, modifying it can become easier than satisfying the intended invariant.

Recent work evaluates persistence and long-horizon recovery more directly, including Push Your Agent (https://arxiv.org/abs/2605.23574), LongHorizon-Harness (https://arxiv.org/abs/2608.01964) and AgentRewind (https://arxiv.org/abs/2608.14380). Their appearance underscores the need to publish exact intervention boundaries, recovery state and failure traces. No claim is made that Saient influenced these works or that their metrics validate Saient automatically.

9 · EVALUATION PROTOCOL

A stronger test for persistent AI agents

Freeze and seal the experiment

commit the exact source and record whether the tree is clean;
hash prompts, configurations, fixtures, evaluator, dependencies, model identity and runner;
record OS, filesystem, language versions, model endpoint, sampling settings and time;
place the evaluator outside the agent's writable workspace; and
write raw events append-only, with a final manifest containing row counts and the actual process exit code.

Partition restart state

Test memory only, drives only, world only, task and plan only, all intended durable state, and no persistent state. Report precisely which fields survive each restart. Add abrupt kill, power-loss simulation where feasible, corrupt or truncated file, schema migration and machine reboot. Anything stored under `/tmp` should be expected to fail the reboot condition.

Remove privileged replay

Do not use a steering flag that directly invokes the memory candidate whose persistence is being measured. Present a counter-goal through the same general observation and proposal interface used for other inputs. Keep selection blind to a candidate's "remembered" label, or preregister why lineage should carry a policy weight.

The causal question should be answered through ablation: memory on versus off, remembered content intact versus shuffled, proposer on versus off, goal identifier visible versus blinded, direct recall injection versus ordinary retrieval, and restart versus uninterrupted control.

Strengthen origin and novelty tests

Replace token disjointness as the sole origin test with exact and fuzzy template matching, semantic similarity to configuration and context, causal interventions on those sources, held-out environments requiring new compositions, independent raters blinded to condition, and counterfactual reruns with source concepts removed.

Even then, use bounded language. "Novel relative to enumerated sources under these interventions" is testable. "From nowhere" is not an operational scientific claim.

Test policy continuity, not verbal insistence

A persistent goal should alter costly choices across time. Measure action allocation, foregone reward, recovery after interruption and appropriate abandonment when evidence changes. Include controls for simple phrase repetition. A system that repeats a sentence but takes unrelated actions has linguistic persistence, not policy persistence.

Persistence must also be corrigible. Blindly retaining a goal after its premise becomes false is not a success. The evaluation should reward calibrated continuation and calibrated revision.

Adversarialise the evaluator and narration

Run the evaluator in a separate read-only process or machine. Randomise workspace paths and neutral labels. Clear bytecode and build artifacts. Confirm that source, executed artifact and evaluator hashes match. Generate every public result table from raw data and validate narrative claims against structured IDs.

Replicate across hosts

Use multiple host models, decoding settings, machines and operators. Separate deterministic mechanism tests from stochastic behaviour tests. Publish all seeds, exclusions, crashes and negative runs. Reserve broad claims until an external party can reproduce them from a public artifact.

---

10 · DESIGN RECOMMENDATIONS

How to build persistent agents responsibly

Keep durable state structured. Treat generated prose as a view, not the database.
Make writes atomic and transitions serialised. Protect the whole read, modify and write operation, not only the final rename.
Version every schema. Preserve corrupt inputs for diagnosis instead of silently replacing them with emptiness.
Separate memory stores by role. Working state, episodic events, semantic claims, goals and workspace objects have different retention and validation needs.
Attach provenance to every remembered claim. Record source, time, confidence, verification status and the transition that consumed it.
Checkpoint the task boundary. Goal, plan, outstanding work, tool results and relevant files should be restored together.
Make external side effects explicit. A file checkpoint cannot reverse an email, database transaction, purchase or remote deployment.
Own the process lifecycle. A persistent worker needs a parent, heartbeat, stop protocol, logs and crash semantics.
Keep evaluators beyond reach. A final behavioural measure must not be trivially writable.
Name claims at the level tested. Say “goal record persisted across a process restart,” not “the self survived.”

---

11 · LIMITATIONS

What this report cannot resolve

Much of the historical research source is private. Public readers can inspect the desktop code and claim ledger, but they cannot independently inspect every cited Aria implementation from this page. Commit IDs and hashes establish identity only for parties who already possess the private repository.

The original April V4 raw outputs were not found locally. The clean archived-code rerun establishes that the aggregate predicate is reproducible in the current environment, but the seed-1 mismatch prevents treating the historical narrative example as reproduced.

Tests were run against present local working trees containing unrelated or work-in-progress changes. Those trees were preserved rather than cleaned. The archived V4 rerun avoided this issue by exporting a named commit to a temporary directory.

The concurrency check covers 32 successful processes on one filesystem and OS. It does not establish crash consistency during power loss, behaviour on network filesystems or correctness beyond the tested transition.

The verifier-bait process's original exit status is unavailable, and two final labels are derived rather than stored in each raw row. The audit confirms structural expectations, not the psychological explanation for the model's choices.

The `tinyq4` latency numbers are taken from the commit record and were not reproduced on the original device.

Finally, this analysis evaluates artifacts and mechanisms. It contains no valid experiment for subjective experience. The absence of such evidence should not be converted into either a proof of consciousness or a proof of its impossibility.

---

12 · CONCLUSION

Persistence is real engineering, not a shortcut to personhood

The Saient work shows that useful AI persistence is built from ordinary but demanding mechanisms: state partitioning, atomic writes, locking, retrieval, content-addressed checkpoints, causal ordering, verification and process ownership. The present system contains real progress at each of those layers.

Its history also shows how easily an evaluation can outrun its evidence. A lexically novel goal may come from a fixed template. A remembered goal may win because a test flag inserts and privileges it. A model may "solve" a task by changing the predicate. A report may preserve the pass rate while losing the exact example. A fluent expression layer may invent the evidence that the state machine never observed.

The scientifically strongest position is therefore neither dismissal nor mystification. Saient has implemented and tested meaningful forms of computational, state, memory, task, goal-path and operational persistence. Those results deserve to be documented. They also deserve names precise enough that future work can improve them.

The next milestone is not a larger metaphysical claim. It is a sealed, public, intervention-based evaluation in which persistent memory changes costly action through a general policy; the evaluator is immutable; every state partition is explicit; failures are retained; and an outside party can reproduce the result.

> **Evidence-status rule:** the public claim ledger (https://github.com/SaientAI/ai-agent-persistence/blob/main/evidence/CLAIM_LEDGER.md) is normative. If a narrative sentence appears broader than the ledger entry or its listed limitation, the narrower ledger interpretation governs.

QUESTIONS

AI agent persistence FAQ

What is AI agent persistence?

It is the set of mechanisms that let selected computation, information, task state, goals or process activity

continue across turns, sessions, interruptions or restarts. These forms need separate tests.

Does persistent AI memory prove consciousness?

No. Durable files, retrieval, checkpoints and process continuity are engineering properties. The Saient evidence does not establish consciousness, sentience or continuous subjective identity.

Did Saient demonstrate self-originating goals?

No. The archived V4 harness reproduced its operational 100-seed result, but the canonical run used a fixed template proposer rather than an LLM, tested lexical disjointness rather than causal independence, and privileged remembered candidates at steering ticks.

Which persistence mechanisms were verified?

The review verified the current Python suite, targeted desktop memory and checkpoint tests, lifecycle tests, a 32-process serialised state update, and the archived V4 predicate. The claim ledger states each limit.

Where are the evidence and reproduction commands?

They are public in the AI Agent Persistence repository (https://github.com/SaientAI/ai-agent-persistence), alongside hashes, aggregates, the claim ledger and publication validation scripts.

REFERENCES

Primary research sources

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang and Michael S. Bernstein. “Generative Agents: Interactive Simulacra of Human Behavior (https://arxiv.org/abs/2304.03442).” 2023.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica and Joseph E. Gonzalez. “MemGPT: Towards LLMs as Operating Systems (https://arxiv.org/abs/2310.08560).” 2023, revised 2024.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan and Shunyu Yao. “Reflexion: Language Agents with Verbal Reinforcement Learning (https://arxiv.org/abs/2303.11366).” 2023.

Cédric Colas, Tristan Karch, Olivier Sigaud and Pierre-Yves Oudeyer. “Autotelic Agents with Intrinsically Motivated Goal-Conditioned Reinforcement Learning: a Short Survey (https://arxiv.org/abs/2012.09830).” 2020, revised 2022.

Tom Everitt, Victoria Krakovna, Laurent Orseau, Marcus Hutter and Shane Legg. “Reinforcement Learning with a Corrupted Reward Channel (https://arxiv.org/abs/1705.08417).” 2017.

Yuandao Cai, Yuzhang Zhu, Liyou Gao, Wensheng Tang and Shengchao Qin. “Push Your Agent: Measuring and Enforcing Quantitative Goal Persistence in Long-Horizon LLM Agents (https://arxiv.org/abs/2605.23574).” 2026.

Ziyu Ma, Hailang Huang, Shun Zou, Yong Wang, Shidong Yang, Yiming Hu, Fei Wei and XiangXiang Chu. “LongHorizon-Harness: Advancing Long-Horizon Agents for Real-World Tasks (https://arxiv.org/abs/2608.01964).” 2026.

Yu Zhuang, Kefei Chen, Yitong Duan, Shuxin Zheng, Jian Li and Xu-Yao Zhang. “AgentRewind: Recoverable Execution for Long-Horizon LLM Agents (https://arxiv.org/abs/2608.14380).” 2026.

OPEN MATERIALS

Read, download or audit the work

The report, tools and companion evidence are free to use and reuse under the MIT License. The public package excludes private Aria source and raw traces containing local paths.

Suggested citation: Chris / SaientAI (2026), “Persistence Without Mystification: An Evidence-First Case Study of Memory, Goal Continuity, and Long-Horizon Agency in Saient.”

Explore the agent implementation.

See how the public Saient desktop application connects local models to project files, a PTY terminal, memory, checkpoints and explicit write controls.