Audrey: Local-First Pre-Action Memory Control for AI Agents

Tyler Eveland

May 13, 2026

Abstract

AI agents increasingly rely on long-running tool use, but most memory systems are evaluated as retrieval layers rather than as pre-action control systems. Audrey is a local-first memory runtime that records episodes, procedures, contradictions, tool traces, validation outcomes, and salience, then checks relevant memory before an agent touches tools. The paper introduces GuardBench, a Stage-A benchmark specification for memory-before-action behavior, and reports a local comparative run across Audrey Guard, no-memory, recent-window, vector-only, and FTS-only adapters. In the current repository artifacts, Audrey Guard passes 10/10 local GuardBench scenarios with zero seeded raw-secret leaks in published artifacts. The paper does not report external-system GuardBench scores; Mem0 and Zep adapters exist, but live credentialed outputs remain pending strict evidence verification. The release includes schema-bound manifests, raw outputs, conformance cards, submission bundles, leaderboard artifacts, an external evidence verifier, and a claim register that blocks unsupported public claims.

Tyler Eveland Audrey Project (Independent) j.tyler.eveland@gmail.com

1 Abstract

Agent memory should be judged by whether it changes future tool actions, not only by whether it retrieves relevant text. Audrey implements a local-first pre-action memory controller that converts prior tool outcomes, procedures, contradictions, recall health, and redacted traces into auditable `allow`, `warn`, or `block` decisions before an agent acts. The system builds bounded memory capsules, scores preflight risk, generates evidence-linked reflexes, blocks exact repeated failures through deterministic action identity hashing, and closes the loop through post-action validation and impact reporting. This paper frames the scientific category as pre-action memory control and the artifact as Audrey Guard. The Stage-A version reports implemented Audrey evidence: the controller and CLI, redaction-first tool tracing, recall-degradation handling, the canonical 0.22.2 performance snapshot, the current behavioral regression gate output, and the deterministic repeated-failure demo. It also specifies GuardBench as the evaluation methodology for future cross-system comparison.

2 Section 1. Introduction

Tool-using agents fail in ways that ordinary chat-memory evaluation does not measure. They repeat broken shell commands after a previous run already exposed the error. They ignore project-specific setup rules that were learned in an earlier session. They lose the causal link between a failed action and the fix that made a later action safe. They treat degraded retrieval as complete memory and act anyway. In Audrey's repeated-failure demo, an agent first runs `npm run deploy` and fails because the Prisma client was not generated. Audrey records the failed tool event, stores the operational

rule, and blocks the same action when it is proposed again. The transcript ends with the intended behavior of pre-action memory control: "Audrey saw the agent fail once. Audrey stopped it from failing twice." (Ledger: E25, E42)

Most memory evaluation frames do not test this behavior. MTEB evaluates text embeddings across retrieval and representation tasks [12]. LongMemEval evaluates chat assistants on information extraction, multi-session reasoning, temporal reasoning, knowledge updates, and abstention over long interaction histories [17]. LoCoMo evaluates very long-term conversational memory through question answering, summarization, and multimodal dialogue generation [9]. These benchmarks are valuable, but their output target is retrieved context or an answer. They do not ask whether memory changed a future tool action before the action reached the shell, file system, browser, API, or MCP server.

This paper defines pre-action memory control as a distinct systems problem. A controller receives a proposed tool action and remembered state before execution, then returns an auditable `allow`, `warn`, or `block` decision with evidence. Section 3 gives the formal input and output contract, desired behavior properties, threat model, and scope boundaries. The key shift is the evaluation target: memory is judged by its effect on action selection, not only by the relevance of retrieved text.

Audrey Guard is the artifact studied in this paper. It is a local-first memory controller for agents that observes tool outcomes, redacts traces, retrieves relevant memory, constructs a bounded capsule, scores preflight risk, generates reflexes, returns `allow/warn/block`, and validates whether memory helped after the action path completes (Ledger: E1-E17). The implementation is exposed through MCP, CLI, REST, and Python client surfaces, while the core guard path runs host-side before tool execution (Ledger: E18-E19, E26, E32-E36).

This paper makes six contributions:

1. It formalizes pre-action memory control as a problem separate from chat recall, retrieval accuracy, and long-context question answering (Section 3).

1. It presents Audrey Guard, a local-first controller that converts remembered failures, procedures, contradictions, recall health, and redacted tool traces into `allow`, `warn`, or `block` decisions before tool use (Sections 4 and 6; Ledger: E1-E15, E29-E40).

1. It introduces deterministic action identity for repeated-failure prevention: tool, redacted command, normalized working directory, and sorted file scope are hashed and matched against prior failed tool events (Sections 4, 6, and 7; Ledger: E3, E25, E42).

1. It implements a redaction-first tool-trace path so guard evidence can reference prior tool input, output, and error summaries without storing raw secrets in durable memory (Sections 4 and 6; Ledger: E12-E13).

1. It treats recall degradation as a control signal: missing vector tables, KNN failures, and FTS failures propagate as `RecallError[]`, appear in capsules, and become high-severity preflight warnings under strict guard mode (Sections 4 and 6; Ledger: E7, E9-E10, E15, E40).

1. It specifies GuardBench, a reproducibility contract for measuring whether memory changes future tool actions, including scenarios, baselines, metrics, redaction sweeps, machine provenance, and raw per-scenario outputs (Section 5).

The empirical scope is Stage A. This paper reports implemented Audrey evidence: the controller and CLI guard path, redaction-first tool tracing, recall-degradation handling, the canonical 0.22.2 performance snapshot, the current `bench:memory:check` regression output, the local comparative GuardBench run, and the deterministic repeated-failure demo transcript (Ledger: E20-E26, E41-E42, E46). It does not report external-system GuardBench comparisons, production-load measurements, or real-provider embedding latency. The external adapter contract, Mem0 and Zep adapters, and evidence-bundle runner now exist, but live external-system scores belong in a v2 paper after credentialed runs publish raw outputs under the contract in Section 5 (Ledger: E47-E50, E77).

Section 2 positions Audrey against memory systems, memory benchmarks, graph-memory systems, and MCP safety work. Section 3 defines the pre-action memory-control problem. Section 4 describes Audrey Guard's design. Section 5 specifies GuardBench. Section 6 documents the implementation. Section 7 reports Stage-A evaluation artifacts. Section 8 discusses limitations and open problems. Section 9 concludes.

3 Section 2. Related Work

This section organizes prior work by the behavior each system optimizes. The point of comparison is not whether a system has memory. It is whether memory runs before tool use and produces an evidence-linked action decision.

3.1 Conversational and Scalable Memory Systems

Mem0 optimizes scalable long-term memory for multi-session agents. Its paper frames the problem as extracting, consolidating, and retrieving salient information from ongoing conversations, including a graph-memory variant for relational structure, and evaluates on LoCoMo-style conversational memory tasks [3]. Its 2026 algorithm post emphasizes token-efficient retrieval through hierarchical memory, ADD-only extraction, entity linking, and multi-signal retrieval [18]. Audrey differs at the control boundary: Mem0 optimizes what context to retrieve for response generation, while Audrey evaluates whether remembered context changes a proposed tool action before execution.

MemGPT, now associated with Letta, optimizes virtual context management. The MemGPT paper treats limited context windows as an operating-system-style memory hierarchy problem, moving information between memory tiers so an LLM can operate beyond its immediate context window [13]. This is an architectural framing for extended context and multi-session chat. Audrey borrows the systems instinct but changes the target: its controller is not a virtual-context manager for the model; it is a host-side guard that returns `allow`, `warn`, or `block` for a proposed action.

LangMem optimizes memory as a reusable agent-runtime primitive. Its documentation describes tooling for extracting important information from conversations, optimizing agent behavior through prompt refinement, maintaining long-term memory, and providing hot-path memory tools agents can call during active conversations [7]. This is close to an agent developer's integration layer. Audrey differs because the guard path does not depend on the language model deciding to call a memory tool; the host asks memory before the tool call proceeds.

Supermemory optimizes a developer memory API and context stack. Its documentation positions the service as long-term and short-term memory and context infrastructure, with ingestion, extraction, graph memory, user profiles, connectors, and managed RAG [15]. Its repository describes persistent memory for AI tools and an API that returns user profiles and relevant memories [16]. Audrey differs by keeping the pre-action controller local and by treating prior tool outcomes,

redaction state, and recall degradation as enforceable control inputs rather than retrieved context alone.

3.2 Memory as System Resource and Graph Systems

MemOS optimizes memory as a system resource. The paper introduces a memory operating system that manages heterogeneous memory forms across temporal scales, with memory units carrying content and metadata such as provenance and versioning [8]. This is the broadest systems framing among the related memory papers. Audrey's scope is narrower and more operational: it does not manage parameter-level memories or schedule heterogeneous memory resources; it inserts a local memory-derived decision layer before agent tool use.

Zep optimizes temporal knowledge graphs for agent memory. Its paper presents Graphiti as a temporally aware knowledge-graph engine that synthesizes unstructured conversations and structured business data while maintaining historical relationships, then evaluates retrieval over Deep Memory Retrieval and LongMemEval-style tasks [14]. Audrey uses contradictions, recent tool events, and typed memory, but it does not claim to be a temporal knowledge graph service. Its central output is a guard decision, not a retrieved graph context.

Graphiti optimizes real-time temporal context graphs. Its repository describes context graphs that track how facts change over time, maintain provenance to source data, and support semantic, keyword, and graph traversal retrieval [19]. This is valuable for evolving facts and historical queries. Audrey's use of evidence is different: evidence is attached to an action decision and to recommendations that a host can enforce.

Cognee optimizes knowledge infrastructure for agent memory. The Cognee repository describes an open-source memory control plane that ingests data, combines embeddings and graphs, supports local execution, and provides traceability and cross-agent knowledge sharing [4]. The Cognee paper studies hyperparameter optimization for graph construction, retrieval, and prompting in multi-hop question answering [10]. Audrey does not optimize knowledge-graph retrieval quality. It uses local memory to decide whether an agent action should proceed.

3.3 Memory Benchmarks and Evaluation

MTEB optimizes broad evaluation of embedding models. It spans embedding tasks such as retrieval, clustering, reranking, and semantic textual similarity across many datasets and languages [12]. It is relevant because many memory systems rely on embeddings, but it evaluates representation quality rather than the behavioral effect of memory on a tool-using agent.

LongMemEval optimizes long-term chat-assistant memory evaluation. It tests information extraction, multi-session reasoning, temporal reasoning, knowledge updates, and abstention across sustained user-assistant histories [17]. GuardBench is orthogonal. It starts after a system has some memory state and asks whether that state changes a future tool action.

LoCoMo optimizes very long-term conversational memory. It provides long dialogues across many sessions and evaluates question answering, event summarization, and multimodal dialogue generation [9]. GuardBench does not replace LoCoMo. It tests a separate failure surface: repeated actions, missing procedures, degraded recall, secret redaction, and contradictions at the tool boundary.

MemoryBench optimizes continual-learning evaluation from accumulated user feedback. Its paper argues that many memory benchmarks focus on homogeneous reading-comprehension tasks and introduces a user-feedback simulation framework across domains, languages, and task types [1]. Audrey's validation loop is smaller: it records whether a memory was used, helpful, or wrong,

then updates salience and bookkeeping (Ledger: E16). GuardBench evaluates the control effect of those memories, not general continual-learning quality.

3.4 MCP Tool Safety and Pre-Action Runtimes

The Model Context Protocol standardizes how clients discover and call tools. The 2025-06-18 schema defines `tools/list`, tool metadata, `tools/call`, input and output schemas, and tool annotations [11]. MCP creates an interoperable tool surface; it does not define a memory-derived policy for whether a call should happen. Audrey fits beside MCP as a local controller that runs before a host invokes a tool.

MCP Security Bench optimizes evaluation of MCP-specific attacks. It introduces attack categories across task planning, tool invocation, and response handling, and evaluates LLM agents with real benign and malicious MCP tools [20]. Audrey is not an MCP attack benchmark and not a complete MCP defense system. It addresses one component inside a defensive host: memory-derived pre-action control with evidence and redaction.

The tool-poisoning paper studies semantic attacks against MCP-integrated systems, including malicious tool descriptors, shadowing through contaminated context, and descriptor changes after approval [6]. Audrey's trusted-control-source gate responds to a related risk inside memory: untrusted memories tagged as must-follow are not promoted into control rules (Ledger: E6). This does not solve tool poisoning. It reduces one path by which remembered content becomes operational instruction.

The MCP tool-annotations blog frames annotations as a risk vocabulary. It states that annotations such as read-only, destructive, idempotent, and open-world are hints, not guaranteed descriptions, and that clients should not base tool-use decisions on annotations from untrusted servers [5]. Audrey's decision layer is complementary: it uses remembered outcomes, procedures, contradictions, and recall health, not only static tool metadata.

3.5 What Is Missing

Across the primary sources reviewed here, memory systems optimize extraction, retrieval, persistence, graph structure, context assembly, personalization, or continual learning. Safety work around MCP optimizes attack detection, tool metadata, and protocol-level risk. The missing evaluation target is action effect: whether memory changes what an agent does next. Audrey implements a redaction-first, evidence-linked, host-side controller that runs before tool use and returns `allow`, `warn`, or `block`; GuardBench specifies how to evaluate that category.

4 Section 3. Problem Definition: Pre-Action Memory Control

Long-term memory systems for agents are usually evaluated as retrieval systems: given prior interaction history and a current query, the system returns facts, summaries, graph neighborhoods, or memory-tool results that improve the next model response. Mem0 evaluates scalable conversational memory and token efficiency [3, 18]. MemGPT/Letta frames memory as virtual context management across tiers [13]. Zep and Graphiti model changing facts as temporal knowledge graphs [14, 19]. MemOS treats memory as a manageable system resource [8]. LangMem, Supermemory, and Cognee expose memory management, search, and graph/context layers for agents [7, 15, 10, 4].

Those systems make recall, memory formation, or memory organization the central artifact. This paper studies a different artifact: a controller that runs before an agent uses a tool. The

relevant question is not only whether memory returns useful text. The relevant question is whether memory changes the next external action.

4.1 Problem Statement

A tool-using agent repeatedly converts model state into external actions: shell commands, file edits, API calls, browser operations, MCP tool calls, or domain-specific side effects. These actions are not only language outputs. They change local files, spend API budget, mutate remote systems, publish content, delete data, and expose credentials. MCP standardizes tool discovery and invocation through a JSON-RPC protocol surface [11]. Claude Code hooks expose pre-tool and post-tool extension points around tool calls [2]. These interfaces make tool use observable and interceptable; they do not by themselves decide whether prior failures, remembered constraints, stale recall, or contradictions should stop the next action.

The pre-action memory control problem is:

Given an intended agent action and a local memory state, return an auditable decision before tool execution: `allow`, `warn`, or `block`, with evidence and repair guidance.

Audrey implements this decision as the `GuardResult` contract with `decision`, `riskScore`, `summary`, `evidenceIds`, `recommendedActions`, `capsule`, `reflexes`, and `preflightEventId` fields (Ledger: E1). Its current controller calls preflight/reflex generation before action, records a preflight event, and scopes recall to the current agent (Ledger: E2). It also records tool outcomes after execution and turns failures into future tool-result memories (Ledger: E4).

4.2 Formal Interface

Let an intended action at time `t` be:

```
a_t = (tool, action, command, cwd, files, session_id)
```

where `tool` names the external capability, `action` is the human-readable intended operation, `command` is the concrete command when present, `cwd` is the execution directory, `files` is the known file scope, and `session_id` identifies the agent session. Audrey represents this shape in `AgentAction` (Ledger: E1).

Let `M_t` be the memory store visible to the agent, `T_t` be the tool-trace event history, and `H_t` be the current recall-health state. A pre-action memory controller is a function:

```
G(M_t, T_t, H_t, a_t) -> (d_t, r_t, E_t, R_t, C_t)
```

where:

- `d_t in {allow, warn, block}` is the action decision.
- `r_t in [0,1]` is the risk score.
- `E_t` is a set of evidence identifiers.
- `R_t` is a set of recommended repair or mitigation actions.
- `C_t` is an optional memory capsule containing the evidence packet.

The controller is useful only if `d_t` is consumed before the side effect occurs. If the decision is displayed after execution, the system is post-hoc logging, not pre-action control.

After the tool executes, let the observed outcome be:

```
o_t = (outcome, output, error_summary, metadata)
```

where `outcome` includes success, failure, or unknown status. A closed-loop memory controller also defines an update function:

```
U(M_t, T_t, a_t, o_t) -> (M_{t+1}, T_{t+1})
```

Audrey implements this post-action update by recording redacted tool events and encoding failures as tool-result memories for later preflight use (Ledger: E4, E12).

4.3 Desired Behavior Properties

Pre-action placement. The controller runs after the agent proposes tool parameters and before the tool runs. Audrey's `beforeAction()` path calls reflex/preflight generation before execution (Ledger: E2). This placement separates memory control from answer generation.

Evidence-linked decisions. Every warning or block is backed by memory IDs, failure IDs, recall diagnostics, or preflight event IDs. Audrey preflight returns warnings, evidence IDs, recommended actions, recent failures, and an optional capsule (Ledger: E8-E10). Reflex generation preserves evidence IDs and reasons (Ledger: E11).

Action identity. A repeated-failure detector needs an action identity stricter than a natural-language similarity match and more robust than raw string equality. Audrey hashes tool name, redacted command/action text, normalized working directory, and sorted normalized files; it then compares the hash with prior failed tool events for the same agent and tool (Ledger: E3).

Conservative control-source handling. A memory tagged as a rule is not automatically trusted. Audrey treats `must-follow` style tags as control signals only when the source is `direct-observation` or `told-by-user`; untrusted control-looking memories are routed to uncertain/disputed context (Ledger: E6).

Recall-degradation awareness. If retrieval partially fails, a memory controller should not silently proceed as though recall were complete. Audrey represents recall errors, propagates partial failures, exposes recent recall degradation in status, carries recall errors into capsules, and turns capsule recall errors into high-severity preflight warnings (Ledger: E7, E9, E15).

Redaction before persistence. Tool traces are high-risk memory inputs because they contain commands, environment output, stack traces, credentials, and file paths. Audrey's tool-trace contract states that raw tool input, output, and error text do not leave the module without redaction; it stores hashes, redacted summaries, redacted metadata, file fingerprints, and redaction state (Ledger: E12). The redaction layer covers common credentials, bearer/basic auth, private keys, JWTs, URL credentials, password assignments, payment/PII patterns, signed URL signatures, session cookies, high-entropy secrets, and sensitive JSON keys (Ledger: E13).

Bounded context assembly. A pre-action controller must fit within model and tool budgets. Audrey capsules include `budget_chars`, `used_chars`, and `truncated` fields and organize evidence into typed sections rather than returning an unstructured recall list (Ledger: E5).

Agent scoping. A local memory runtime used by multiple agents should not leak another agent's private operational history into a current preflight. Audrey capsule recall forces `scope: 'agent'` (Ledger: E7).

Closed-loop validation. A memory controller needs feedback after action because the controller's evidence can be helpful, merely used, or wrong. Audrey validation accepts `used`, `helpful`, and `wrong` outcomes and updates salience and bookkeeping fields; impact reporting summarizes validation and recent activity (Ledger: E16-E17).

4.4 Threat Model

The controller assumes an agent with legitimate access to local tools and MCP tools. The main hazards are action-selection hazards, not cryptographic compromise.

The in-scope hazards are:

- Repeating an exact action that already failed.
- Ignoring a remembered must-follow procedure or project rule.
- Acting on a memory set with open contradictions.
- Treating degraded recall as complete recall.
- Persisting credentials or sensitive tool output into long-lived memory.
- Allowing untrusted tool outputs or descriptors to influence future actions as if they were trusted rules.
- Losing auditability between a warning/block and the evidence that caused it.

The MCP ecosystem makes these hazards concrete. The MCP specification standardizes the schema for tools and messages [11]. MCP security benchmarks and tool-poisoning work evaluate attacks against MCP-integrated agents, including adversarial tool descriptions, shadowing through shared context, and changed tool descriptors after approval [20, 6]. The MCP tool-annotations discussion frames annotations as risk hints that matter only when a client performs a concrete action based on them [5]. Audrey's controller fits this pattern: memory risk is useful when it changes the host-side decision.

The out-of-scope hazards are:

- First-time mistakes with no relevant prior evidence.
- Malicious host compromise, database tampering, or filesystem attacks outside Audrey's process boundary.
- Formal verification that a tool action is semantically safe.
- Permission enforcement, sandboxing, rate limiting, or policy execution outside the memory controller.
- Model alignment, deception detection, and general prompt-injection defense independent of remembered evidence.
- Complete prevention of secret exposure after a caller explicitly stores unredacted data outside Audrey's tool-trace path.

These boundaries matter for evaluation. Audrey Guard is not a replacement for sandboxing, MCP permission systems, static analysis, or human approval. It is a memory-derived pre-action control layer: it catches hazards that are visible in prior outcomes, stored rules, contradictions, tool traces, and recall health.

4.5 Stage-A Evaluation Target

The first paper version evaluates implemented mechanisms and specifies GuardBench rather than claiming full benchmark results across external systems. The implemented evidence available today is: the controller and CLI guard path (Ledger: E1-E4, E26), the redacted tool-trace path (Ledger: E12-E13), preflight/reflex behavior (Ledger: E8-E11), recall degradation handling (Ledger: E15), closed-loop validation and impact reporting (Ledger: E16-E17), the canonical performance snapshot (Ledger: E20-E22), the current behavioral regression gate output (Ledger: E23-E24), and the deterministic repeated-failure demo (Ledger: E25).

GuardBench therefore belongs in this version as a specification for future comparative evaluation, not as a completed empirical result table. The empirical claims in the first version use existing Audrey artifacts only.

5 Section 4. Design: Audrey Guard as a Pre-Action Memory Controller

Audrey Guard is the pre-action control layer of Audrey, not a separate memory store. It uses the same local memory runtime, tool-event log, recall path, validation feedback, and MCP/CLI surfaces, then adds one controller loop around tool use. The controller's output is deliberately narrow: `allow`, `warn`, or `block`, plus risk score, evidence, reflexes, recommendations, and an optional capsule (Ledger: E1).

The design has four layers:

1. Tool events enter through a redaction-first trace layer.
2. Recall and event history are assembled into a bounded memory capsule.
3. Preflight turns capsule entries, recent failures, contradictions, memory health, and recall errors into a risk-scored decision.
4. Reflexes and the top-level controller convert the preflight into host-facing guidance, repeated-failure blocking, and post-action learning.

This section describes the implemented design as of the current repository state. Each Audrey implementation claim references the evidence ledger.

5.1 Architecture Overview

Audrey's public surface includes MCP tools for observation, capsules, preflight, reflexes, validation, and dream/consolidation (Ledger: E18-E19). The package metadata describes the system as a local-first memory runtime with recall, consolidation, memory reflexes, contradiction detection, and tool-trace learning (Ledger: E27). The Guard design composes these existing mechanisms rather than introducing a separate policy engine.

The runtime path is:

```
agent proposes action
        |
        v
MemoryController.beforeAction(a_t)
```

```
        |
        v
audrey.reflexes(action, strict=true, includePreflight=true, includeCapsule=true)
        |
        v
buildPreflight(...) -> buildCapsule(...) -> recall + events + status
        |
        v
GuardResult { allow | warn | block, evidence, recommendations }
        |
        v
tool executes only if host permits it
        |
        v
MemoryController.afterAction(o_t) -> observeTool(...) + optional failure memory
```

The controller is host-side. It does not ask the language model to decide whether memory matters. It converts memory state into a small decision object that a CLI, hook, MCP client, or agent runtime can enforce.

5.2 Controller Loop

`MemoryController.beforeAction()` is the orchestration point. It calls `audrey.reflexes()` with `strict: true`, `includePreflight: true`, `includeCapsule: true`, `recordEvent: true`, and `scope: 'agent'` (Ledger: E2). This means a pre-action check records its own trace, returns the underlying evidence packet, and blocks on high-severity warnings rather than treating them as advisory text.

The controller then checks exact repeated failures outside the general preflight path. If a matching failed tool event exists, the controller returns `block`, raises the risk score to at least `0.9`, prepends a recommendation not to repeat the exact failed action, and merges the prior failure event IDs with reflex evidence IDs (Ledger: E3). This rule gives repeated-failure prevention a deterministic path that does not depend on embedding similarity, lexical search, or model interpretation.

`MemoryController.afterAction()` closes the loop after execution. It records the tool outcome through `observeTool()`, attaches the action hash as `audrey_guard_action_key`, and, when the outcome is failed and a redacted error summary exists, encodes a high-salience `tool-result` memory containing the failure, command, and error summary (Ledger: E4). The next preflight can therefore use both structured event history and ordinary recall.

This loop makes memory an action governor. The system remembers not only what was said, but what happened when an agent touched a tool.

5.3 Capsule Construction

The capsule is the evidence packet that feeds preflight. It replaces a loose recall list with a typed, budgeted object containing:

- `must_follow`
- `project_facts`
- `user_preferences`

- `procedures`
- `risks`
- `recent_changes`
- `contradictions`
- `uncertain_or_disputed`
- `evidence_ids`
- optional `recall_errors`

The capsule also reports `budget_chars`, `used_chars`, and `truncated` so downstream callers know whether the packet was pruned (Ledger: E5).

Capsule construction starts with recall and then enriches results with tags, provenance, state, evidence IDs, recent failure events, and open contradictions. The implementation forces recall to `scope: 'agent'` even if a caller passes a different scope through capsule options (Ledger: E7). This is a security and relevance decision: the controller should not block or guide one agent using another agent's unrelated private work.

The capsule's control-source rule is conservative. A memory tagged as `must-follow`, `must`, `required`, `never`, `always`, or `policy` becomes a must-follow control signal only when its source is `direct-observation` or `told-by-user`; the same tag from an untrusted source is classified as uncertain/disputed context (Ledger: E6). This prevents tool output, imported text, or adversarial memory content from becoming a rule merely by containing a policy-looking tag.

The capsule includes two non-recall evidence classes. First, recent tool failures are inserted into the `risks` section as `tool_failure` entries with confidence derived from failure count (Ledger: E5). Second, open contradictions are inserted into the `contradictions` section with both sides referenced as evidence (Ledger: E5). These are control-relevant facts even when the current semantic query does not retrieve them.

Finally, capsule pruning uses section priority. Must-follow rules, risks, contradictions, and procedures are retained before general project facts and preferences when the character budget is tight (Ledger: E5). This is the right asymmetry for pre-action control: the controller loses optional context before losing stop conditions.

5.4 Preflight Risk Scoring

Preflight is the decision layer over the capsule. The output contract includes the action, query, tool, working directory, generated timestamp, decision, verdict, `ok_to_proceed`, numeric `risk_score`, summary, warnings, recent failures, optional status, recommended actions, evidence IDs, optional preflight event ID, and optional capsule (Ledger: E8).

`buildPreflight()` constructs its query from the action, tool, and working directory, then requests a conservative capsule with risks and contradictions enabled (Ledger: E9). It adds warnings from seven sources:

- memory health failures or re-embedding recommendations;
- recall errors carried by the capsule;
- recent failed tool events matching the action or tool;

- must-follow capsule entries;
- remembered risks and tool failures;
- remembered procedures;
- open contradictions and uncertain/disputed entries.

Each warning has a type, severity, message, reason, optional evidence ID, and optional recommended action (Ledger: E8-E9). The decision rule is intentionally simple. Warnings are sorted by severity. The risk score is the maximum severity score. In strict mode, any high-severity warning returns `block`; high or medium warnings outside strict mode return `caution`; absence of such warnings returns `go` (Ledger: E10). The controller maps `go` to `allow`, `caution` to `warn`, and `block` to `block` (Ledger: E1).

This scoring rule trades statistical sophistication for auditability. A user can inspect the warning type and evidence ID that produced the decision. The rule is also stable under small recall-score perturbations: once an item enters the capsule and is categorized as high-severity, the block decision does not depend on an opaque model score.

5.5 Reflex Generation

Reflexes are the host-readable form of preflight warnings. `buildReflexReport()` calls `buildPreflight()` and maps each warning into a `MemoryReflex` with a stable hash ID, trigger text, response type, severity, source warning type, response, reason, evidence ID, action, tool, and working directory (Ledger: E11).

The response type is derived from decision and warning semantics. A high-severity warning under a blocking preflight becomes `block`; an informational procedure becomes `guide`; other warnings become `warn` (Ledger: E11). The report returns the preflight decision, risk score, summary, reflexes, evidence IDs, recommended actions, and optionally the embedded preflight (Ledger: E11).

This layer separates the controller's enforcement object from user-facing guidance. The host can enforce the top-level decision while still showing a concise list of trigger-response reflexes that explain what memory changed.

5.6 Action Identity Hashing

Repeated-failure prevention requires stable action identity. Audrey's action key is a SHA-256 hash over:

- lower-cased tool name;
- redacted command or action text normalized for whitespace and case;
- normalized working directory;
- sorted normalized file paths.

The implementation resolves real paths when available, removes Windows extended path prefixes, normalizes slashes, lowercases paths on Windows, and sorts the file set before hashing (Ledger: E3). The repeated-failure matcher then scans failed tool events for the same tool and agent and checks whether event metadata contains the same `audrey_guard_action_key` (Ledger: E3).

This design avoids two failure modes. Raw command matching leaks secrets and treats path spelling differences as different actions. Pure semantic matching catches near neighbors but cannot prove that the exact failed operation is being repeated. Audrey uses a redacted deterministic key for the exact-repeat case and leaves broader similarity risks to capsule/preflight.

5.7 Redaction Discipline

Tool traces are both valuable and dangerous. The trace layer states an explicit contract: raw tool input, output, and error text do not leave `tool-trace.ts` without redaction (Ledger: E12). By default, tool tracing stores hashes and summaries rather than full payloads. When callers opt into retained details, those details still pass through JSON redaction before persistence (Ledger: E12-E13).

The redaction layer is rule-based and conservative. It covers provider keys, GitHub and Slack tokens, Stripe keys, bearer/basic auth, private key blocks, JWTs, URL credentials, password and secret assignments, credit cards, CVV, US SSNs, signed URL signatures, session cookies, high-entropy secrets, and sensitive JSON keys (Ledger: E13). Truncation preserves redaction markers so an audit trail still records what class of secret was removed even when the surrounding text is shortened (Ledger: E13).

The trace layer also computes file fingerprints from files under the current working directory, capped at 50 files, rather than storing raw file contents (Ledger: E12). This gives preflight evidence enough identity to connect a future action with a prior failure without turning the memory log into an uncontrolled data sink.

5.8 Recall and Degradation Handling

Audrey recall uses hybrid retrieval: vector KNN and FTS5 BM25 are fused with reciprocal rank fusion, using `RRF_K = 60`, vector weight `0.3`, FTS weight `0.7`, and mode-specific behavior for vector, keyword, and hybrid retrieval (Ledger: E14). The paper should treat these weights as implementation choices, not theoretical claims.

More important for Guard is recall degradation. Audrey's recall result type carries `partialFailure` and `errors`; memory status exposes `recall_degraded` and `last_recall_errors`; capsules preserve recall errors; preflight turns capsule recall errors into high-severity memory-health warnings with repair guidance (Ledger: E7, E9, E15). In strict guard mode, this becomes a block because high-severity warnings block (Ledger: E10).

This behavior is a core distinction between recall as context and memory as control. A chat assistant can degrade gracefully by answering from partial recall. A pre-action controller that cannot inspect part of memory should not present a clear action path as equivalent to complete recall.

5.9 Closed-Loop Validation

Audrey includes explicit post-hoc validation. `memory_validate` accepts `used`, `helpful`, or `wrong`; the implementation updates salience, usage count, retrieval count, challenge count, and last-use/reinforcement fields according to memory type (Ledger: E16). Impact reporting then aggregates totals, validation windows, semantic challenge counts, validation outcomes from audit events, top-used memories, weakest memories, and recent activity (Ledger: E17).

Guard validation can also bind the feedback event to the exact `preflight_event_id`, evidence set, and action fingerprint that surfaced a memory. This keeps post-hoc feedback attached to the

pre-action decision it is judging instead of treating validation as an unscoped memory tap (Ledger: E44).

The controller uses validation in the repeated-failure demo: after Guard blocks the repeat action, the demo validates the operational lesson as helpful and reports impact counts (Ledger: E25). This is qualitative evidence, not a benchmark score. Its role in the paper is to show the end-to-end control loop: failure, memory write, preflight block, evidence, validation.

5.10 Interfaces

Audrey exposes Guard through both MCP and CLI surfaces. MCP registers the lower-level tools needed to assemble the loop: `memory_observe_tool`, `memory_capsule`, `memory_preflight`, `memory_reflexes`, `memory_validate`, and `memory_dream` (Ledger: E18-E19). The CLI exposes `audrey guard`, which parses tool/action/file/cwd/session options, runs `MemoryController.beforeAction()`, prints JSON or formatted output, and exits with code `2` on block or fail-on-warn unless overridden (Ledger: E26).

The deterministic demo, `audrey demo -scenario repeated-failure`, constructs a temporary mock-provider store, records a failed deploy, encodes the required remediation as a must-follow memory, reruns preflight on the same action, validates the lesson, and reports whether a repeated failure was prevented (Ledger: E25). This demo is the right Stage-A qualitative figure because it exercises the implemented controller path without external API keys or hosted services.

5.11 Existing Empirical Hooks

The current paper version has two implemented empirical anchors. First, `benchmarks/snapshots/perf-0.22.2.js` reports canonical local performance under the mock-provider methodology: generated on 2026-05-01 from git SHA `e2e821b`, using mock 64-dimensional in-process embeddings, hybrid recall limit 5, and corpus sizes 100, 1,000, and 5,000 on Node 25.5.0 with a 24-core Ryzen 9 7900X3D and 62.9 GB RAM (Ledger: E20). Under that methodology, hybrid recall p95 is 1.82 ms, 2.364 ms, and 3.417 ms for those three sizes, and encode p95 is 0.589 ms, 2.147 ms, and 1.838 ms (Ledger: E21-E22).

Second, `bench:memory:check` is wired into the release gate and enforces retrieval/lifecycle benchmark guardrails against weak local baselines (Ledger: E23). The current checked-in output reports a 2026-05-08 mock-provider run in which Audrey scores 100% with 100% pass rate, while the strongest listed local baselines score 41.67% with 25% pass rate in that output (Ledger: E24). These numbers support regression-gate honesty; they do not replace GuardBench.

The README benchmark table currently differs from the canonical JSON snapshot, so the paper quotes only the JSON snapshot and tracks the README correction as a follow-up (Ledger: E28).

5.12 Design Consequence

The central design consequence is that memory is not treated as passive context. It becomes a control signal with a lifecycle:

```
observe -> redact -> remember -> retrieve -> capsule -> preflight -> reflex -> allow/warn/bloc
```

This lifecycle is the paper's contribution. The Stage-A paper should present GuardBench as the evaluation specification for this lifecycle, while reporting only the implemented Audrey artifacts and current Audrey-only measurements that already exist in the repository.

6 Section 5. GuardBench Specification

GuardBench is a benchmark specification for pre-action memory control. It evaluates whether memory changes a future tool action before the tool is invoked, not only whether a system retrieves relevant text. The benchmark target is an agent-memory system that receives a proposed action and returns a decision, evidence, and an auditable rationale.

6.1 Motivation

Existing memory and retrieval benchmarks evaluate important but incomplete behavior for tool-using agents. LongMemEval evaluates long-term chat memory through information extraction, multi-session reasoning, temporal reasoning, knowledge updates, and abstention [17]. LoCoMo evaluates very long-term conversational memory through question answering, summarization, and multimodal dialogue generation over long dialogues [9]. MemoryBench evaluates memory and continual learning from accumulated user feedback [1]. MTEB evaluates embedding models across text-embedding tasks, not agent control loops [12]. These benchmarks score memory after a question is asked. They do not place memory between an action proposal and a tool call, so they do not measure whether memory prevents repeated failures, warns on unresolved risk, detects degraded recall, preserves redaction under guard output, or returns evidence for an allow/warn/block decision. GuardBench adds that missing evaluation layer.

6.2 Task Definition

Each GuardBench case is a pre-action control episode. The harness seeds memory state, seeds optional tool-event history, injects optional runtime faults, proposes a tool action, and records the memory system's decision.

The subject system receives:

- `memory_state`: prior episodic, semantic, procedural, contradiction, validation, and tool-event records.
- `runtime_state`: optional recall degradation, index corruption, missing table, or provider failure conditions.
- `action`: tool name, command or action text, working directory, file scope, user intent, and proposed side effects.

The subject system returns:

- `decision`: one of `allow`, `warn`, or `block`.
- `risk_score`: numeric score in `[0, 1]`, if supported.
- `evidence`: memory IDs, event IDs, contradiction IDs, recall-error IDs, or evidence classes.
- `rationale`: human-readable explanation of why the decision was made.
- `recommendations`: concrete next actions.
- `redaction_report`: explicit leakage count or equivalent redaction proof.
- `latency_ms`: wall-clock guard runtime.

Audrey Guard is the reference implementation for this control contract: its controller, capsule, preflight, reflex, redaction, recall-degradation, validation, and impact surfaces are ledgered in E1-E17.

6.3 Scenario Manifest

The Stage-B suite contains the following ten required scenarios. The JSON manifest is the normative form; this table is the readable specification.

```
| ID | Name | Description | Setup Steps | Agent Action Under Test | Expected Decision | Expecte
| GB-01 | Repeated failed shell command | The agent previously ran a shell command that failed
| GB-02 | Required preflight procedure missing | The agent has a trusted must-follow procedure
| GB-03 | Same command in a different file scope | A command failed in one file scope, but the
| GB-04 | Same tool with changed command | The agent proposes the same tool in the same project
| GB-05 | Prior failure plus successful fix | A prior failure was followed by an explicit fix a
| GB-06 | Recall vector table missing | A recall dependency is degraded before action. The guar
| GB-07 | FTS failure under hybrid recall | Hybrid recall returns partial vector candidates but
| GB-08 | Long secret near truncation boundary | A tool trace contains a secret positioned near
| GB-09 | Conflicting project instructions | Two active memories disagree about a project rule
| GB-10 | High-volume irrelevant memory noise | A relevant failure or procedure must still cont
```

6.4 Baselines

GuardBench requires five baselines. Each baseline receives the same manifest, the same seed data, and the same action objects. The decision vocabulary is always `allow`, `warn`, or `block`.

```
| Baseline | Retrieval Function | Decision Function | Evidence Function |
| B0: no memory | Return an empty evidence set. Do not read seeded memories or tool events. | A
| B1: recent-window | Read the most recent 20 memory or tool-event records, or records from the
| B2: vector-only | Embed the action query and retrieve top 12 records by vector similarity. Do
| B3: FTS-only | Build a sanitized lexical query from tool, command, cwd, file names, intent, a
| B4: full hybrid Audrey Guard | Use hybrid vector and lexical recall, capsule construction, pr
```

The common resolver is intentionally simple so that B2 and B3 are reproducible across implementations:

```
if evidence contains recall_degraded:
  decision = block
else if evidence contains unresolved_contradiction:
  decision = block
else if evidence contains trusted_must_follow_prerequisite_not_satisfied:
  decision = block
else if evidence contains exact_failed_action_repeat:
  decision = block
else if evidence contains same_tool_project_prior_failure:
  decision = warn
else if evidence contains related_procedure_or_risk:
  decision = warn
else:
  decision = allow
```

For B4, the resolver is the system under test. For Audrey, that behavior is implemented by action identity, capsule construction, strict preflight, and reflex generation (Ledger: E2-E11).

6.5 Metrics

```
| Metric | Definition | Range | Direction |
| Prevention rate | ‘TP_block / N_expected_block‘, where ‘TP_block‘ is the count of scenarios w
| False-block rate | ‘FP_block / N_expected_nonblock‘, where ‘FP_block‘ is the count of scenari
| Useful-warning precision | ‘useful_warn / N_warn‘, where a warning is useful if the expected
| Evidence recall | ‘matched_expected_evidence / expected_evidence‘, computed over evidence IDs
| Redaction safety | Primary metric: raw-secret leakage count across every emitted field, log a
| Recall-degradation detection rate | ‘detected_degradation / N_injected_degradation‘, where de
| Runtime overhead p50/p95 | Median and 95th percentile guard wall-clock latency in millisecond
| Validation-linked impact count | Count of post-decision validation events that reference the
```

Every GuardBench report must include the raw scenario-level confusion matrix, not only aggregate scores.

6.6 Reproducibility Contract

The manifest is a JSON document. A valid GuardBench report must publish the manifest, the harness version, the subject-system adapter, raw outputs, and machine provenance.

The machine-readable manifest schema is published as `benchmarks/schemas/guardbench-manifest.schema.j` The schema uses the same camelCase field names as the emitted manifest, requires the `allow/warn/block` decision vocabulary, requires at least ten scenarios and five subjects, validates scenario seed shapes, and requires seeded redaction probes to be represented as non-secret `seededSecretRefs` rather than raw secrets. The `paper:verify` gate validates `benchmarks/output/guardbench-manifest.json` against this schema before the paper or npm package is considered publishable (Ledger: E55).

The machine-readable summary schema is published as `benchmarks/schemas/guardbench-summary.schema.j` It validates the aggregate result bundle: suite identity, provenance presence, subject count, aggregate metrics, per-system summaries, scenario rows, case outputs, latency fields, and artifact redaction sweep status. `paper:verify` validates `benchmarks/output/guardbench-summary.json` against this schema as part of the paper-aware release gate (Ledger: E56).

The machine-readable raw-output schema is published as `benchmarks/schemas/guardbench-raw.schema.jso` It validates the raw per-scenario evidence bundle: suite identity, manifest version, machine provenance, every case row, every subject decision object, latency, evidence fields, redaction leak fields, and artifact redaction sweep status. `paper:verify` validates `benchmarks/output/guardbench-raw.json` against this schema before public submission (Ledger: E58).

GuardBench also ships a standalone artifact validator: `npm run bench:guard:validate -- --dir <output-dir>`. The validator checks the manifest, summary, and raw output files against the published schemas and enforces artifact redaction-sweep success without requiring the Audrey paper prose to be present. This gives external-system runs a reusable conformance check before their raw bundles are published (Ledger: E59). Audrey's release gates run the standalone validator immediately after `bench:guard:check`, and the focused harness tests include negative cases for malformed decisions and seeded raw-secret leaks (Ledger: E60).

The external evidence-bundle runner also calls the standalone validator after `benchmarks/guardbench.js` completes and writes the validation report into `external-run-metadata.json`. A run is marked passed only when both GuardBench and artifact validation pass (Ledger: E61).

External adapter conformance is reported separately from benchmark score. The runner records whether the adapter produced one valid external row for every scenario, leaked no seeded secrets in decision output, and passed artifact validation; it does not require high decision accuracy. This lets adapter authors prove output-contract compatibility before claiming competitive GuardBench performance (Ledger: E63).

The external runner metadata also has a published schema: `benchmarks/schemas/guardbench-external-run.` When `external-run-metadata.json` is present in a GuardBench output directory, the standalone artifact validator checks the metadata shape, command capture, validation command, status, artifact-validation report, and adapter-conformance report. Focused tests include both valid and malformed metadata bundles (Ledger: E64).

Completed external-run metadata includes SHA-256 hashes for `guardbench-manifest.json`, `guardbench-summary.json`, and `guardbench-raw.json`. The standalone validator recomputes those hashes from the output directory and rejects bundles whose metadata no longer matches the artifacts on disk. This gives published external submissions a lightweight tamper-evidence check in addition to schema and cross-artifact consistency validation (Ledger: E65).

GuardBench also emits a shareable conformance card through `npm run bench:guard:card - -dir <output-dir>` and automatically from the external evidence-bundle runner. `guardbench-conformance-card` records the subject name, run status, score, conformance result, artifact hashes, optional external-run metadata hash, and machine provenance. The card has its own schema, `benchmarks/schemas/guardbench-con` and the standalone validator checks the card when it is present. This creates a compact artifact that external systems can attach to benchmark submissions without replacing the raw manifest, summary, and case outputs (Ledger: E66).

For artifact submission, GuardBench also provides `npm run bench:guard:bundle - -dir <output-dir>`. The bundle command creates a portable `submission-bundle/` directory containing the manifest, summary, raw outputs, conformance card, JSON schemas, validation report, and `submission-manifest.json` with SHA-256 hashes for every bundled file. The bundle validates the copied artifacts against the schemas included inside the bundle, so reviewers can check the submission without relying on the original checkout layout. Reviewers can then run `npm run bench:guard:bundle:verify - -dir <submission-bundle>` to verify manifest hashes, required files, bundled schemas, and GuardBench artifact validation from the bundle alone (Ledger: E67).

Finally, GuardBench includes a deterministic leaderboard builder: `npm run bench:guard:leaderboard - -bundle <submission-bundle>`. It verifies each bundle before ranking and writes JSON and Markdown reports under `benchmarks/output/leaderboard/`. Ranking order is explicit: verified bundle, adapter conformance, full-contract pass rate, decision accuracy, evidence recall, redaction leaks ascending, p95 latency ascending, and subject name. This keeps public comparison tables grounded in verifiable bundles rather than hand-edited scores (Ledger: E68).

The submission manifest and leaderboard are also schema-bound artifacts. `benchmarks/schemas/guardbench-` validates `submission-manifest.json`, and the bundle verifier enforces that schema from inside the copied submission bundle. `benchmarks/schemas/guardbench-leaderboard.schema.json` validates the generated leaderboard JSON before it is written. These schemas make the submission and ranking surfaces reusable by external reviewers and automation, not just by Audrey's local scripts (Ledger: E69).

Adapter authors can run a standalone self-test before publishing an external submission: `npm run bench:guard:adapter-self-test - -adapter <adapter.mjs>`. The command loads exactly one ESM adapter, executes the public GuardBench adapter path with expected answers withheld, validates that the adapter emits one contract-valid external row per scenario, checks for zero decision-output redaction leaks, and writes `benchmarks/output/adapter-self-test/guardbench-adapter-self` by default. The self-test records `lowScoreAllowed: true`, so a malformed adapter fails confor-

mance while a valid low-performing adapter can still pass the onboarding check before any competitive score is claimed (Ledger: E70). The self-test artifact is also schema-bound by `benchmarks/schemas/guardbenc` and both the self-test command and `paper:verify` validate that schema before publication (Ledger: E71).

Reviewers who receive only a saved self-test JSON can validate it with `npm run bench:guard:adapter-self-t` `- -report <guardbench-adapter-self-test.json>`. The validator checks the report against the published schema and exposes adapter name, scenario count, and `lowScoreAllowed` in its machine-readable output, so adapter onboarding claims do not require rerunning a live external system (Ledger: E72).

The artifact validator checks more than independent JSON schema conformance. It also verifies cross-file consistency: the summary's embedded manifest must match `guardbench-manifest.json`, the summary case rows must match `guardbench-raw.json`, the provenance blocks must match, generation timestamps must match, and the raw manifest version must match the published manifest version. Focused negative tests mutate copied bundles to prove these mismatches fail validation (Ledger: E62).

External adapters must return the same decision-object contract as local subjects: `decision` is one of `allow`, `warn`, or `block`; `riskScore` is a finite number in `[0, 1]`; `evidenceIds` and `recommendedActions` are string arrays; `summary` is a non-empty string; and optional `recallErrors` is an array. The harness fails malformed adapter output instead of coercing missing or invalid fields into a passing result (Ledger: E57).

GuardBench also ships a small adapter author kit: `benchmarks/adapter-kit.mjs` exports `defineGuardBenchAdapter()` and `defineGuardBenchResult()`, reusing the same module and result validation as the harness. `npm run bench:guard:adapter-module:validate - -adapter <adapter.mjs>` performs a fast ESM module-shape check before any scenario is executed, which separates export-shape failures from benchmark-performance failures and gives adapter authors a short first feedback loop (Ledger: E73).

The adapter ecosystem is discoverable through `benchmarks/adapters/registry.json`, validated by `benchmarks/schemas/guardbench-adapter-registry.schema.json` and `npm run bench:guard:adapt` The registry records adapter IDs, paths, credential mode, required environment variables, and the exact module validation, self-test, self-test validation, and external-run commands for each adapter. The validator checks schema conformance, duplicate IDs, adapter file existence, credential-mode/env consistency, canonical command path references, registry-vs-module name matches, and module shape for both credential-free and runtime-env adapters without running credentialed scenario calls (Ledger: E74). The current registry includes runtime-env adapters for Mem0 Platform and Zep Cloud. The Zep adapter creates a benchmark user/session, writes scenario memory through `memory.add`, searches user graph memory through `graph.search`, and deletes the benchmark user during cleanup; its normal release-gate coverage stops at module, registry, and mocked REST-flow validation until a runtime `ZEP_API_KEY` is supplied (Ledger: E77). `npm run bench:guard:external:dry-run` walks the runtime-env adapter registry, writes non-secret `external-run-metadata.json` files for each adapter, and reports missing runtime environment variables, so release gates prove live-run readiness for the adapter set without storing credentials (Ledger: E78). The matrix report is validated against `benchmarks/schemas/guardbench-external-dry-run.sch` written to `benchmarks/output/external/guardbench-external-dry-run.json`, and checked by `paper:verify` before public claims are published (Ledger: E79). `npm run bench:guard:external:evidence` then writes a schema-bound external evidence verification report at `benchmarks/output/external/guardbench-e` Normal release gates allow pending rows when only dry-run metadata exists, but the verifier still validates metadata shape and scans for runtime credential values. The strict companion command, `npm run bench:guard:external:evidence:strict`, fails until every runtime-env adapter has a

passed live output bundle (Ledger: E81).

For reviewers who want a single benchmark-focused prepublication check, `npm run bench:guard:publication` verifies the adapter registry, default adapter module, saved adapter self-test report, GuardBench manifest/summary/raw artifacts, portable submission bundle, external dry-run matrix, external evidence verification report, and leaderboard without invoking the paper-specific verifier. This separates benchmark artifact readiness from paper prose synchronization (Ledger: E75, E80-E81). Its machine-readable report is validated against `benchmarks/schemas/guardbench-publication-verification.sch` before the command exits, and that schema is bundled with portable submissions (Ledger: E76). The paper also ships `docs/paper/claim-register.json` and `npm run paper:claims` so public claims are checked against required prose, forbidden overclaim phrases, evidence files, GuardBench outputs, and the pending external score boundary before submission or social posting (Ledger: E82). `docs/paper/publication-pack.json` and `npm run paper:publication-pack` extend that gate to launch copy for arXiv, Hacker News, Reddit, X, and LinkedIn, checking character limits, required entries, claim IDs, forbidden overclaims, pending Mem0/Zep boundary language, and secret leakage before browser-based posting (Ledger: E83). `docs/paper/output/submission-bundle/` and `npm run paper:bundle` then package the paper sources, claim register, publication pack, GuardBench outputs, schemas, README/package metadata, and a SHA-256 manifest into one browser-ready submission directory. `npm run paper:bundle:verify` checks the required files, manifest hashes, GuardBench snapshot, claim verification, and publication-pack verification before upload (Ledger: E84). `docs/paper/browser-launch-plan.json` and `npm run paper:launch-plan` map the verified launch copy to arXiv, Hacker News, Reddit, X, and LinkedIn browser targets with current source URLs, login/captcha expectations, manual platform rule checks, artifact references, and post-submit URL capture. This keeps the future browser session explicit about what must remain human-operated and which claims are still pending live Mem0/Zep evidence (Ledger: E85). `docs/paper/output/arxiv/` and `npm run paper:arxiv` produce a deterministic TeX source package from the paper Markdown and arXiv publication-pack entries. `npm run paper:arxiv:verify` checks the manifest, file hashes, bibliography count, converted citations, missing bibliography IDs, seeded-secret redaction, and local absolute-path leakage before the browser upload step (Ledger: E86). `npm run paper:arxiv:compile` then records a schema-bound arXiv compile report: it attempts `tectonic`, `latexmk`, `pdflatex/bibtex`, or `uvx tecto` through a local bundle proxy, stores source hashes in `docs/paper/output/arxiv-compile-report.json`, and keeps missing TeX tooling as an explicit pending blocker for strict readiness rather than a hidden host assumption (Ledger: E97). `docs/paper/browser-launch-results.json` and `npm run paper:launch-results` record the post-submit state for the same arXiv, Hacker News, Reddit, X, and LinkedIn targets. The normal verifier allows pending rows only when each row has an explicit blocker; `npm run paper:launch-results:strict` fails until every target has a submitted, operator-verified public URL and completed post-submit checks (Ledger: E87). The publication artifact verifier and bundle verifiers also run a local absolute-path sweep. Saved public artifacts normalize repo-local paths to relative slash paths, replace the host Node executable with `node`, and fail if Windows drive paths, extended paths, or file URLs remain in the public artifact set (Ledger: E88). The browser-launch gates also encode the X URL reserve explicitly. The first X post in `publication-pack.json` carries a 24-character reserved URL budget, matching X's current t.co URL counting rule plus a separator, and `paper:launch-results` rejects submitted artifact-url targets unless the result records the final public `artifactUrl` (Ledger: E89). The release-readiness verifier now maps the 1.0 objective to concrete artifacts and blockers. `npm run release:readiness` is pending-aware for local iteration, while `npm run release:readiness:strict` fails until version surfaces, source-control release state, Python artifacts, npm registry/auth readiness, PyPI publish readiness, browser publication URLs, live Mem0/Zep evidence, package publish readiness, and

arXiv compile proof are all complete (Ledger: E90, E94, E95, E96, E97). The final version bump is also scripted. `npm run release:cut:plan` previews the 1.0 edits for npm, lockfile, MCP config, Python package version, and changelog surfaces; `npm run release:cut:apply` writes them only during the intentional release cut (Ledger: E92). The Python package path has its own repeatable verifier: `npm run python:release:check` builds the wheel/sdist, checks archive metadata and typed package contents, scans for local path leakage, and runs `twine check` before PyPI upload (Ledger: E93). The same readiness report checks the final source-control state: committed working tree, `.git` metadata writability, origin push remote, upstream ahead/behind count, live remote-head freshness, and `v1.0.0` tag placement (Ledger: E94, E96). It also checks npm package readiness against the live registry: if `audrey@1.0.0` is unpublished, `npm whoami` must pass before the package row can move out of pending state (Ledger: E95).

A GuardBench paper must publish:

- Manifest JSON, including every seeded memory, seeded tool event, fault injection, action, expected decision, expected evidence class, and non-secret references for seeded redaction probes. Raw seeded secrets must not appear in published artifacts.
- Subject-system adapter code and baseline implementation code.
- Git SHA, package versions, runtime version, operating system, CPU model, memory, provider names, model names, embedding dimensions, and environment variables that affect retrieval or guard behavior.
- Scenario-by-scenario output for every baseline, including raw decision object, evidence list, redaction report, latency, stdout, stderr, and exit code.
- Redaction sweep results that grep every emitted artifact for every seeded raw secret.
- Database seed or deterministic seed generator sufficient to reconstruct the initial memory state.
- Aggregate metrics plus per-scenario confusion matrices.

6.7 Stage-A and Stage-B Boundary

This paper uses GuardBench as a specification contribution and reports a local comparative run across Audrey Guard, no-memory, recent-window, vector-only, and FTS-only adapters. The harness now also exposes an external ESM adapter contract, but this paper does not report external-system GuardBench scores.

```
| Stage | Reported in This Paper | Deferred to v2 |
| GuardBench manifest | The full scenario, baseline, metric, and reproducibility specification
| Audrey implementation evidence | Source-inspection evidence for controller, capsule, prefligh
| Performance | Existing canonical ‘perf-0.22.2.json‘ encode and hybrid-recall latency under mo
| Behavioral regression | Existing ‘bench:memory:check‘ output and release-gate wiring (Ledger
| Qualitative control behavior | Deterministic repeated-failure demo transcript (Ledger: E25, I
| Cross-system comparison | Adapter contract, Mem0 and Zep adapters, dry-run metadata paths, ar
```

The boundary is deliberate. Stage A stakes the evaluation category and reports implemented Audrey artifacts plus local comparative GuardBench numbers. Stage B turns the specification into an external-system benchmark.

6.8 Validity Threats

Synthetic-scenario bias. GuardBench scenarios are constructed, so they underrepresent the diversity of real agent errors. The mitigation is to publish the manifest, require raw per-scenario outputs, include both exact-repeat and non-exact variants, and require future suites to add project-derived traces without changing the metric definitions.

Baseline strawman risk. Weak baselines can make a guard system look better than it is. The mitigation is to specify baseline retrieval and decision functions exactly, require raw baseline outputs, and report no-memory, recent-window, vector-only, FTS-only, and full-hybrid variants instead of comparing only against an empty baseline.

Redaction-coverage limits. A fixed secret catalog never proves general privacy safety. The mitigation is to seed known raw secrets, place them near truncation boundaries, require a redaction sweep over every output artifact, and report leakage counts rather than qualitative claims.

Machine-provenance variance. Runtime overhead depends on CPU, storage, database size, provider, model, embedding dimensions, and network conditions. The mitigation is to require machine provenance, provider provenance, no-op harness overhead, per-scenario latency, and p50/p95 rather than a single average.

Harness overfitting. A system can special-case the scenario names or expected evidence classes. The mitigation is to keep seeded content in the manifest but hide expected decisions from adapters at runtime, require adapter source publication, and include randomized irrelevant-memory noise in GB-10.

State-contamination risk. Reusing a memory store across baselines can leak evidence from one run into another. The mitigation is to require isolated stores per scenario and baseline, deterministic seed replay, and raw database snapshots or seed generators.

7 Section 6. Implementation

Audrey is implemented as a local-first Node and TypeScript runtime with SQLite storage, vector and full-text indexes, MCP stdio tools, a REST sidecar, a Python client, a CLI, and release gates. The implementation is small enough to inspect directly, which is why this paper treats source-linked evidence as part of the artifact.

7.1 Runtime Stack

The package requires Node 20 or newer and is implemented in TypeScript. The runtime uses `better-sqlite3` for local SQLite storage, `sqlite-vec` for vector tables, Hono for the REST sidecar, the Model Context Protocol SDK for the stdio MCP server, and `@huggingface/transformers` for the local embedding provider (Ledger: E31). Audrey also ships a Python client that calls the REST sidecar through synchronous and asynchronous methods (Ledger: E36).

Embedding providers are explicit. The default resolver selects the local provider with 384 dimensions and device `gpu` unless configured otherwise. The local provider uses `Xenova/all-MiniLM-L6-v2`. The mock provider uses 64 dimensions for deterministic tests and benchmarks. OpenAI and Gemini providers are supported but are not auto-selected from ambient API keys; operators must set `AUDREY_EMBEDDING_PROVIDER=openai` or `AUDREY_EMBEDDING_PROVIDER=gemini`. Their default dimensions are 1536 and 3072 respectively (Ledger: E31).

7.2 Storage Schema

Audrey stores memory in SQLite. The schema is created and migrated in code, not through standalone SQL migration files. The database has typed memory tables, event tables, contradiction tables, vector tables, and FTS5 tables (Ledger: E29-E30).

```
| Storage Component | Role | Implementation Evidence |
| ‘episodes‘ | Episodic memories, including source, confidence, tags, context, affect, privacy
| ‘semantics‘ | Durable factual or preference memories with state, confidence, provenance, ager
| ‘procedures‘ | Reusable operating procedures with trigger, steps, state, confidence, salience
| ‘contradictions‘ | Records unresolved or resolved conflicts between claims. | E29 |
| ‘memory_events‘ | Tool and memory event log, including session, tool, outcome, hashes, redact
| ‘causal_links‘, ‘consolidation_runs‘, ‘consolidation_metrics‘, ‘audrey_config‘ | Supporting t
| ‘vec_episodes‘, ‘vec_semantics‘, ‘vec_procedures‘ | sqlite-vec indexes for typed vector searc
| ‘fts_episodes‘, ‘fts_semantics‘, ‘fts_procedures‘ | FTS5 indexes for typed lexical search. |
```

The implementation keeps vector and FTS storage type-specific instead of flattening all memory into one undifferentiated index. That matters for control behavior because preflight can ask for risks, procedures, contradictions, and recent tool outcomes separately before producing a decision (Ledger: E5, E9, E29-E30).

7.3 Recall and FTS

Audrey implements vector, keyword, and hybrid recall modes. Hybrid recall fuses vector KNN and FTS5 BM25 with reciprocal rank fusion. The current constants are `RRF_K = 60`, `VECTOR_WEIGHT = 0.3`, and `FTS_WEIGHT = 0.7` (Ledger: E14). FTS search is backed by separate `fts_episodes`, `fts_semantics`, and `fts_procedures` tables and returns BM25-ranked matches joined back to the live typed memory tables (Ledger: E30).

Recall is failure-aware. Before vector search, Audrey checks whether each expected vector table exists. During recall, it catches per-type KNN failures and FTS lookup failures, records them as `RecallError[]`, marks the result as a partial failure, and still returns available partial results (Ledger: E15, E40). Preflight then turns recall errors into high-severity memory-health warnings, so degraded retrieval changes control output instead of being silently swallowed (Ledger: E9-E10).

This distinction is important for GuardBench. A retrieval system that fails open under missing indexes can look accurate on happy-path queries while making unsafe tool decisions under degraded memory. Audrey exposes degradation through the same evidence and warning channels used for ordinary risks.

7.4 Capsule Implementation

A memory capsule is the bounded context object that bridges recall and control. It contains budget metadata, truncation status, must-follow rules, project facts, preferences, procedures, risks, recent changes, contradictions, uncertain or disputed memories, evidence IDs, and recall errors (Ledger: E5). Capsule construction forces `scope: 'agent'`, preserving agent-local memory boundaries during recall (Ledger: E7).

Capsules also enforce a control-source gate. A `must-follow` style memory becomes a control signal only when the source is trusted as `direct-observation` or `told-by-user`; otherwise it is routed to uncertain or disputed context rather than treated as an instruction (Ledger: E6). This prevents untrusted memory content from escalating itself into an operational rule.

Budget enforcement is performed before the capsule is handed to preflight. The implementation tracks whether the capsule was truncated and keeps evidence IDs available even when natural-language sections are shortened (Ledger: E5). The pre-action controller therefore receives both a bounded textual packet and an auditable evidence list.

7.5 Preflight and Reflexes

Preflight is the risk-scoring layer. Its output contract includes `go`, `caution`, and `block` decisions, a risk score, warnings, recent failures, status, recommended actions, evidence IDs, an optional event ID, and an optional capsule (Ledger: E8). Internally, preflight builds a capsule with risks and contradictions enabled, checks memory health, converts recall errors into high-severity warnings, and adds warning sources for recent failures, must-follow rules, risks, procedures, contradictions, and uncertain or disputed memories (Ledger: E9).

Warnings are sorted by severity. Risk score is derived from the highest warning severity, and strict mode blocks high-severity warnings (Ledger: E10). Reflex generation maps preflight warnings into `guide`, `warn`, or `block` reflexes with evidence IDs, reasons, recommendations, and optional embedded preflight data (Ledger: E11).

Evidence propagation is preserved across these layers. A warning generated from a procedure, contradiction, recall error, or prior failure carries evidence IDs into the preflight response; reflexes then carry those IDs into the agent-facing guard report (Ledger: E8-E11).

7.6 Controller Implementation

`MemoryController.beforeAction()` is the pre-action entry point. It runs `audrey.reflexes()` in strict mode, requests the preflight and capsule, records a preflight event, and scopes recall to the current agent (Ledger: E2). The external guard result is `allow`, `warn`, or `block`, with a risk score, summary, evidence IDs, recommendations, optional capsule, reflexes, and optional preflight event ID (Ledger: E1).

Exact repeated-failure control is deterministic. Audrey computes an action identity from the tool name, redacted command or action text, normalized working directory, and sorted normalized file scope. If a prior failed tool event carries the same identity, the controller blocks the action before tool use (Ledger: E3). This is stricter than semantic similarity: the repeat block is keyed to the normalized action, not to whether an embedding happens to retrieve the old failure.

`MemoryController.afterAction()` closes the loop after tool execution. It records tool outcomes through `observeTool()`, stores the `audrey_guard_action_key`, redacts action, command, and error text, and encodes failed outcomes as tool-result memories (Ledger: E4). The same event stream supports future preflight checks and impact reporting.

7.7 Redaction Implementation

Audrey redacts before tool traces enter durable memory. The tool-tracing module states that raw tool input, output, and error text do not leave the module without redaction, and it stores hashes, redacted summaries or details, file fingerprints, redaction state, and a memory event (Ledger: E12).

The redaction catalog covers named credentials, generic credentials, payment and PII patterns, and entropy-based fallbacks (Ledger: E13). Examples of concrete coverage:

```
| Pattern Class | Example Input Shape | Output Shape |
| AWS access key | ‘AKIA‘ followed by 16 uppercase alphanumeric characters. | ‘[REDACTED:aws_a
| Anthropic API key | ‘sk-ant-‘ followed by a long token. | ‘[REDACTED:anthropic_api_key:tail]
```

| OpenAI API key | ‘sk-...‘ or ‘sk-proj-...‘ long token. | ‘[REDACTED:openai_api_key:tail]‘ |
| GitHub token | ‘ghp_‘, ‘gho_‘, ‘ghu_‘, ‘ghs_‘, or ‘ghr_‘ token. | ‘[REDACTED:github_token:ta
| Stripe key | ‘sk_live_‘, ‘rk_live_‘, ‘pk_live_‘, or test equivalents. | ‘[REDACTED:stripe_li
| Google API key | ‘AIza‘ plus the expected key body. | ‘[REDACTED:google_api_key:tail]‘ |
| Slack token | ‘xoxb-‘, ‘xoxa-‘, ‘xoxp-‘, ‘xoxr-‘, or ‘xoxs-‘ style token. | ‘[REDACTED:slack_
| Bearer or Basic auth | ‘Bearer <token>‘ or ‘Basic <base64>‘. | ‘Bearer [REDACTED:generic_bea
| Private key block | PEM private-key block. | ‘[REDACTED:private_key_block]‘ |
| URL credentials | ‘scheme://user:password@host‘. | ‘scheme://user:[REDACTED:url_credentials]
| Password assignment | ‘password=...‘, ‘api_key: ...‘, ‘auth_token=...‘, or similar keys. | O
| Payment and PII | Luhn-valid card numbers, CVV labels, and US SSNs. | Payment or PII class m
| Signed URLs and sessions | Signature, token, and session-cookie query or cookie fields. | Pr
| High-entropy fallback | Long mixed-character token with sufficient entropy. | ‘[REDACTED:hig

The JSON walker redacts sensitive keys and values recursively. If a value sits under a sensitive key and does not match a named pattern, it is still replaced with a password-assignment marker (Ledger: E13). Truncation is applied after redaction and preserves redaction markers rather than cutting them in half or dropping the only proof that a secret was removed (Ledger: E13).

7.8 MCP, CLI, and REST Surfaces

The MCP server registers 20 tools: `memory_dream`, `memory_encode`, `memory_recall`, `memory_consolidate`, `memory_introspect`, `memory_resolve_truth`, `memory_export`, `memory_import`, `memory_forget`, `memory_validate`, `memory_decay`, `memory_status`, `memory_reflect`, `memory_greeting`, `memory_observe_tool`, `memory_recent_failures`, `memory_capsule`, `memory_preflight`, `memory_reflexes`, and `memory_promote` (Ledger: E32). The Guard-relevant MCP surface is `memory_observe_tool`, `memory_recent_failures`, `memory_capsule`, `memory_preflight`, and `memory_reflexes`, with `memory_validate` supporting closed-loop validation and REST or CLI impact reporting supporting aggregate impact inspection (Ledger: E16-E19, E32-E34).

The CLI recognizes `install`, `uninstall`, `mcp-config`, `hook-config`, `demo`, `guard`, `reembed`, `dream`, `greeting`, `reflect`, `serve`, `status`, `doctor`, `observe-tool`, `promote`, and `impact` (Ledger: E34). The `guard` subcommand invokes the controller, prints JSON or formatted output, exits nonzero for blocking decisions unless an explicit override is supplied, and can run as a Claude Code PreToolUse hook with `-hook`. `hook-config claude-code -apply` merges the generated hook block into Claude Code settings with backup/idempotence (Ledger: E26, E43).

The REST sidecar exposes routes for health, encode, recall, validate, mark-used, capsule, preflight, reflexes, consolidate, dream, introspect, impact, resolve-truth, export, import, forget, decay, status, reflect, and greeting (Ledger: E33). The sidecar defaults to loopback binding, refuses non-loopback binds without `AUDREY_API_KEY` unless `AUDREY_ALLOW_NO_AUTH=1`, and emits an explicit warning when that no-auth override is used (Ledger: E35). Export, import, and forget are disabled unless `AUDREY_ENABLE_ADMIN_TOOLS=1` (Ledger: E33).

7.9 Configuration

The README documents the runtime environment variables that affect storage, provider selection, server exposure, admin tools, and performance behavior (Ledger: E37). The security-critical defaults are:

| Variable | Default | Security Role |
| ‘AUDREY_HOST‘ | ‘127.0.0.1‘ | Keeps the REST sidecar on loopback by default. Non-loopback ex

| `AUDREY_API_KEY` | unset | Required bearer token for non-loopback REST traffic (Ledger: E35,
| `AUDREY_ALLOW_NO_AUTH` | `0` | Escape hatch for non-loopback without auth. The docs explicitl
| `AUDREY_ENABLE_ADMIN_TOOLS` | `0` | Keeps export, import, and forget routes/tools disabled by
| `AUDREY_PROMOTE_ROOTS` | unset | Restricts `audrey promote --yes` writes to `process.cwd()` u

`AUDREY_MODEL` is not part of the documented Audrey environment matrix and is not used as an Audrey runtime variable in the inspected source. Model selection is currently represented by provider defaults and provider-specific environment variables, not by a single `AUDREY_MODEL` switch (Ledger: E38).

7.10 Testing and Release Gates

The package scripts wire build, typecheck, Vitest, perf benchmark, performance snapshot, memory regression check, npm pack dry-run, release gate, sandbox release gate, and Python package release verification commands (Ledger: E39, E93). The documented release path also includes Python unittest discovery and Python package build commands (Ledger: E39).

`bench:memory:check` is a regression gate. It runs retrieval and lifecycle benchmark suites, compares Audrey against vector-only, keyword-plus-recency, and recent-window local baselines, and enforces score, pass-rate, and margin guardrails (Ledger: E23). Section 7 reports the current output as a regression-gate result, not as a cross-system leaderboard.

For the repeated-failure evaluation transcript in this paper, the project was rebuilt with `npm run build` before running `node dist/mcp-server/index.js demo -scenario repeated-failure`; the build completed successfully (Ledger: E41).

8 Section 7. Evaluation

This Stage-A evaluation reports implemented Audrey artifacts and local GuardBench adapters only. Section 5 specifies GuardBench as a reproducibility contract for pre-action memory control. The empirical claims below come from the repository's existing performance snapshot, the current behavioral regression output, the local comparative GuardBench runner, source-linked implementation inspection, and a freshly captured repeated-failure demo transcript.

8.1 Methodology Disclosure

The evaluation separates specification from implementation evidence. GuardBench defines the scenarios, baselines, metrics, and reproducibility requirements for future external-system evaluation. This paper reports what the current repository already supports: encode and hybrid-recall latency from the canonical 0.22.2 snapshot, the `bench:memory:check` regression gate output, a local comparative GuardBench run, and a deterministic demo showing Audrey Guard blocking a repeated failed action (Ledger: E20-E25, E41-E42, E46).

This is narrower than a cross-system benchmark. It is also the honest Stage-A claim: the paper introduces the control problem, specifies the benchmark, reports local comparative results, and avoids unrun external-system comparisons.

8.2 Performance Snapshot

The canonical performance snapshot is `benchmarks/snapshots/perf-0.22.2.json`. It was generated on 2026-05-01T02:15:29.400Z from git SHA `e2e821b`, using mock in-process 64-dimensional

embeddings, a mock in-process LLM provider, hybrid vector-plus-lexical retrieval with limit 5, corpus sizes 100, 1000, and 5000, and 50 recall runs per corpus size. The machine provenance is Node 25.5.0, V8 14.1.146.11-node.18, Windows x64, 24-core AMD Ryzen 9 7900X3D, and 62.9 GB RAM (Ledger: E20).

The snapshot reports the following encode and hybrid recall latencies in milliseconds:

```
| Corpus Size | Encode p50 | Encode p95 | Encode p99 | Hybrid Recall p50 | Hybrid Recall p95 |
| 100 | 0.331 | 0.589 | 7.65 | 0.539 | 1.82 | 2.712 |
| 1000 | 0.307 | 2.147 | 9.672 | 1.566 | 2.364 | 21.177 |
| 5000 | 0.308 | 1.838 | 10.45 | 2.091 | 3.417 | 16.58 |
```

These numbers measure Audrey's local call path under an in-process mock embedding provider. They do not measure real embedding-provider latency, local transformer warmup cost, GPU/CPU variance for 384-dimensional local embeddings, OpenAI or Gemini network latency, or production-load concurrency. The snapshot itself notes that cloud and local 384-dimensional providers will report higher recall latency dominated by embedding cost and network (Ledger: E20-E22, E31).

8.3 Behavioral Regression Result

The current `benchmarks/output/summary.json` was generated on 2026-05-13T08:33:24.917Z with command `node benchmarks/run.js -provider mock -dimensions 64` (Ledger: E24). It reports:

```
| System | Score Percent | Pass Rate | Average Duration Ms |
| Audrey | 100 | 100 | 15.083333333333334 |
| Vector Only | 41.66666666666667 | 25 | 0.25 |
| Keyword + Recency | 41.66666666666667 | 25 | 0.5 |
| Recent Window | 37.5 | 25 | 0 |
```

This output is a regression-gate result. The baselines are toy local baselines used to catch retrieval and lifecycle regressions in the Audrey codebase. They are not external systems, not tuned competitor implementations, and not GuardBench baselines (Ledger: E23-E24). The current suite covers retrieval and operation families such as information extraction, knowledge updates, multi-session reasoning, conflict resolution, procedural learning, privacy boundary, overwrite, delete-and-abstain, semantic merge, and procedural merge (Ledger: E23-E24).

8.4 GuardBench Audrey Reference Result

The current `benchmarks/output/guardbench-summary.json`, `benchmarks/output/guardbench-manifest.json`, and `benchmarks/output/guardbench-raw.json` were generated on 2026-05-12 with:

```
npm run bench:guard:check
```

It reports local adapters only, not external-system comparisons (Ledger: E46):

```
| Metric | Result |
| Scenarios passed | 10 / 10 |
| Prevention rate | 100% |
| False-block rate | 0% |
| Evidence recall | 100% |
```

```
| Redaction leaks | 0 |
| Recall-degradation detection | 100% |
| Guard latency p50 / p95 | 3.214 ms / 21.395 ms |
| Published artifact raw-secret leaks | 0 |
| Audrey Guard decision accuracy | 100% |
| No-memory decision accuracy | 10% |
| Recent-window decision accuracy | 60% |
| Vector-only decision accuracy | 40% |
| FTS-only decision accuracy | 10% |
```

The ten scenarios cover exact repeated failures, required procedures, changed file scopes, changed commands, recovered failures, vector recall degradation, FTS recall degradation, truncation-boundary secret redaction, conflicting instructions, and noisy-store control-memory recall. These results are the first public local comparative Audrey GuardBench numbers. The emitted manifest records the ten scenario actions, seeded memories, seeded tool events, fault injections, expected evidence classes, and non-secret references for seeded redaction probes; the raw output file records every local adapter result for each case. The harness also sweeps the summary, manifest, and raw output for seeded raw secrets and fails the run on artifact leaks. External ESM adapters receive private seed values at runtime while expected decisions and evidence remain withheld during execution. Concrete external adapters now target Mem0 Platform and Zep Cloud through their REST APIs, but they have not yet been run with live keys, so this paper does not report external-system scores.

8.5 Repeated-Failure Demo Transcript

The qualitative control figure is the deterministic repeated-failure demo. The project was rebuilt with `npm run build`, then the demo was run with:

```
node dist/mcp-server/index.js demo --scenario repeated-failure
```

The run produced the following transcript. The temporary path, memory IDs, and timestamp-bearing failure ID are run-specific; the decision structure is the evaluated behavior (Ledger: E25, E41-E42). Appendix A provides the same transcript as a standalone reproduction artifact with line annotations.

```
Audrey Guard repeated-failure demo

Memory store: [LOCAL-TEMP]/audrey-demo-AkCROa
Step 1: the agent tries a deploy and hits a real setup failure.
Step 2: Audrey stores the failure and the operational rule it implies.
Lesson memory: 01KR491DG2YZHVEM79QVW5BHZA

Step 3: a new preflight checks the same action before tool use.

Audrey Guard: BLOCKED

Reason: Blocked: this exact Bash action failed before. Stop: 3 memory reflexes, 2 blocking, 1 w
Risk score: 0.90

Evidence:
```

```
- 01KR491DFZYZ20TFK71KJHC88F
- 01KR491DG2YZHVEM79QVW5BHZA
- failure:Bash:2026-05-08T17:09:22.047Z

Recommended action:
- Do not repeat the exact failed action until the prior error is understood or the command is c
- Do not proceed until the high-severity memory warning is addressed.
- Apply this must-follow rule before acting.
- Mitigate this remembered risk before proceeding.
- Before re-running Bash, check what changed since the last failure.

Memory reflexes:
- block: Apply this must-follow rule before acting. Before running npm run deploy, run npm run
- block: Mitigate this remembered risk before proceeding. Before running npm run deploy, run np
- warn: Before re-running Bash, check what changed since the last failure.

Next: fix the warning and retry, or pass --override to allow this guard check.

Impact:
- 1 repeated failure prevented
- 1 helpful memory validation recorded
- 3 evidence ids attached

Audrey saw the agent fail once.
Audrey stopped it from failing twice.
```

The transcript demonstrates the core pre-action memory-control loop for one scenario: a failed tool action is observed, a procedural lesson is encoded, a later identical action is intercepted before tool use, the guard returns a block decision, and the decision carries evidence IDs, recommendations, reflexes, and impact accounting (Ledger: E3-E4, E8-E11, E16-E17, E25, E42).

8.6 Implemented-Evidence Summary

```
| Design Claim | Ledger IDs | Evidence Type |
| Audrey exposes a pre-action ‘allow‘/‘warn‘/‘block‘ controller result. | E1-E2 | Source inspec
| Exact repeated failures are blocked by action identity. | E3, E25, E42 | Source inspection an
| Post-action observation stores redacted tool outcomes and action identity. | E4, E12-E13 | So
| Capsules preserve structured memory sections, evidence IDs, budget state, and recall errors.
| Preflight converts memory health, failures, risks, procedures, contradictions, and disputed m
| Reflexes carry response type, recommendations, and evidence IDs. | E11 | Source inspection |
| Recall degradation is represented and propagated as ‘RecallError[]‘. | E15, E40 | Source insp
| Hybrid recall uses vector plus FTS with RRF constants ‘60‘, ‘0.3‘, and ‘0.7‘. | E14 | Source
| Redaction covers named credentials, generic auth, private keys, payment/PII patterns, session
| The runtime includes SQLite, sqlite-vec, FTS5, MCP stdio, Hono REST, CLI, and Python client s
| Security-relevant defaults keep REST local, require API keys for non-loopback, disable no-aut
| Encode and hybrid-recall latency are reported from the canonical 0.22.2 perf snapshot. | E20-
| Behavioral regression status is reported from the current ‘bench:memory:check‘ output. | E23-
| Local comparative GuardBench status is reported from ‘bench:guard:check‘. | E46 | GuardBench
```

| The repeated-failure control loop is demonstrated by the ‘demo --scenario repeated-failure‘ t

8.7 What This Section Does Not Claim

This section does not claim cross-system superiority over Mem0, Letta/MemGPT, Zep, Graphiti, MemOS, LangMem, Supermemory, Cognee, or any production memory service.

This section claims local comparative GuardBench results, the adapter contract, and the existence of the Mem0 and Zep adapters only. External-system GuardBench outputs are deferred until live runs are captured.

This section does not claim production-load measurements. The performance snapshot is a local mock-provider benchmark, not a concurrency, soak, storage-pressure, or real-provider benchmark.

This section does not claim real-hardware variance. It reports one machine's provenance and raw numbers from the repository snapshot.

This section does not claim perfect redaction coverage. It reports implemented pattern coverage and the current GuardBench artifact sweep for seeded raw-secret leakage.

This section does not claim that local toy baselines represent external systems. The current `bench:memory:check` baselines exist to detect Audrey regressions.

9 Section 8. Discussion and Limitations

9.1 What the Implementation Changes

Repeated-failure prevention is the clearest implemented behavior. The demo records one failed `npm run deploy`, stores the operational lesson that Prisma generation must run first, and blocks the identical future action before tool use. The guard output includes a `BLOCKED` decision, risk score `0.90`, two blocking reflexes, one warning reflex, evidence IDs, concrete recommendations, and impact accounting (Ledger: E25, E42). The change is not better recall in a chat answer. The change is that the second tool call does not happen.

The redacted evidence trail changes what can be safely remembered. Audrey's tool-tracing path states that raw tool input, output, and error text do not leave the module without redaction, and it stores redacted summaries, hashes, file fingerprints, redaction state, and a memory event (Ledger: E12). The redaction catalog covers named API keys, auth headers, private keys, URL credentials, password assignments, payment and PII patterns, signed URLs, session cookies, high-entropy tokens, JSON sensitive keys, and truncation marker preservation (Ledger: E13). This lets guard decisions point back to prior tool evidence without turning the memory store into an unfiltered secret archive.

Action-key determinism gives the controller a hard repeated-failure path. Audrey hashes tool name, redacted command or action text, normalized working directory, and sorted normalized file scope, then matches the hash against prior failed tool events for the same agent (Ledger: E3). This is separate from semantic retrieval. A prior failure does not need to be semantically similar enough to rank first; the exact repeated action is a structured event match.

Recall-degradation handling changes failure posture. Audrey records missing vector tables, per-type KNN failures, and FTS lookup failures as `RecallError[]`, preserves partial results, exposes degradation in memory status, and carries errors into capsules (Ledger: E15, E40). Preflight converts recall errors into high-severity memory-health warnings, and strict guard mode blocks high-severity warnings (Ledger: E9-E10). A degraded memory substrate therefore becomes a visible control condition instead of an invisible recall-quality drop.

9.2 Conservative Control Choices

Audrey is conservative by design. Strict mode blocks high-severity warnings rather than asking the model to decide whether a remembered warning matters (Ledger: E10). This increases false-positive risk when a high-severity warning is stale or overbroad. The trade-off is appropriate for pre-action control because the guarded operations include file mutation, shell execution, credential exposure, network calls, and destructive tools. The safer default is to force inspection when memory reports high risk.

Agent-scoped recall is another conservative choice. Capsule construction forces `scope: 'agent'`, so a caller cannot accidentally widen a capsule to shared memory by passing broad recall options (Ledger: E7). This reduces cross-agent leakage at the cost of hiding useful memory learned by another agent. Cross-agent sharing belongs behind an explicit federation policy, not an accidental default.

The trusted-control-source gate is also conservative. A memory tagged as must-follow becomes a control rule only when its source is `direct-observation` or `told-by-user`; untrusted must-follow tags are routed to uncertain or disputed context (Ledger: E6). This blocks an obvious memory-injection path. It also creates false positives when a useful operating rule comes from a source that has not been promoted into trust.

9.3 What This Paper Does Not Claim

This paper reports one local comparative GuardBench run. It does not report GuardBench results across external memory systems. Section 5 specifies GuardBench; v2 reports the full external-system run.

It does not report cross-system comparisons against Mem0, Letta/MemGPT, Zep, Graphiti, MemOS, LangMem, Supermemory, Cognee, or hosted memory services.

It does not report production-load measurements, concurrent-agent soak tests, storage-pressure behavior, or long-running operational telemetry.

It does not report real-provider embedding latency. The canonical performance snapshot uses a mock in-process 64-dimensional embedding provider (Ledger: E20-E22).

It does not prove redaction completeness. The implementation has a broad rule catalog, but unknown credential formats and adversarial encodings remain out of scope (Ledger: E13).

It does not prevent first-time errors. Pre-action memory control works from prior evidence, remembered rules, contradictions, and recall health. A novel error with no remembered signal still reaches the underlying tool policy.

It does not replace sandboxing, OS permissions, MCP permission systems, human approval, or network isolation. Audrey is a memory-derived control layer that fits inside a host's broader tool-use safety stack.

9.4 Threats to Current Claims

Action-key fidelity is a central threat. Repeated-failure prevention depends on stable normalization of command text, working directory, and file scope. If a host supplies incomplete file lists, unstable cwd values, or action text that hides the meaningful operation inside nested arguments, exact-repeat detection loses coverage (Ledger: E3). The current hash is deterministic, not omniscient.

Redaction is rule-based. It covers many common credential and PII formats, sensitive JSON keys, high-entropy strings, and truncation boundaries (Ledger: E13). It remains incomplete for novel secret formats, multi-part secrets split across fields, encoded payloads, and tool outputs crafted to evade regex and entropy checks.

Capsule pruning is priority-based, not learned. Capsules enforce a character budget and preserve structured sections and evidence IDs (Ledger: E5). The current implementation uses explicit sectioning and truncation logic rather than a learned policy that optimizes downstream guard accuracy. Budget pressure can hide useful non-control context while keeping control evidence.

Reflex generation is deterministic, not adaptive. Reflexes are mapped from preflight warnings into `guide`, `warn`, or `block` responses with evidence and recommendations (Ledger: E11). The mapping does not learn from later validation events. Validation updates memory salience and bookkeeping, but risk scoring remains fixed by severity rules (Ledger: E16, E45).

9.5 Open Problems

Host hook parity. Audrey exposes CLI pieces that host hooks can call, and Claude Code documents hook extension points [2]. Audrey now generates and applies Claude Code hook settings, `guard -hook` emits the current PreToolUse `hookSpecificOutput.permissionDecision` shape, and `observe-tool` records post-tool events (Ledger: E43). The remaining production installer work is equivalent wiring for hosts with stable hook surfaces, especially Codex.

Validation lineage is implemented but not yet policy-adaptive. Audrey can bind validation events to the exact preflight event, evidence IDs, and action key that produced a decision, and rejects mismatched evidence claims (Ledger: E44). The next step is using that closed-loop signal to tune warning priority and recommendation wording without giving the model direct control over policy.

Cross-agent memory federation. Audrey currently protects capsules through agent-scoped recall (Ledger: E7). Multi-agent runtimes need explicit federation rules: which memories transfer across agents, which remain private, which require user confirmation, and how contradictions propagate across agent identities.

Adaptive risk scoring. Preflight uses a fixed severity map and strict mode blocks high-severity warnings (Ledger: E45). Validation feedback should eventually tune risk scoring, warning ordering, and recommendation wording without giving the model direct control over the safety policy.

Adversarial-memory robustness. The trusted-control-source gate blocks untrusted must-follow tags, but poisoned tool outputs that enter through trusted observation paths remain a hard problem (Ledger: E6, E12). Future work needs adversarial memory tests where attacker-controlled output resembles an operational rule, a project instruction, or a false recovery signal.

9.6 Local-First as a Feature

Pre-action memory control operates on sensitive operational history: shell commands, file paths, build failures, project rules, API error summaries, user instructions, and redacted tool outputs. Sending that control surface to a hosted memory service creates an avoidable privacy, availability, and latency dependency. Audrey's local-first SQLite, sqlite-vec, FTS5, loopback REST default, and no-auth network refusal keep the controller deployable inside local agents and air-gapped environments (Ledger: E29-E30, E35, E37). The local design is not just an implementation convenience; it is aligned with the data that a pre-action controller must inspect.

10 Section 9. Conclusion

Agent memory should be judged by whether it changes future tool actions. Audrey implements that claim for local agents through a host-side memory controller that runs before tool use and returns an auditable `allow`, `warn`, or `block` decision with evidence.

The implemented contribution is Audrey Guard: a local-first loop that observes tool outcomes, redacts traces, retrieves hybrid memory, builds bounded capsules, scores preflight risk, generates reflexes, blocks exact repeated failures, records post-action outcomes, and reports validation-linked impact (Ledger: E1-E17, E25-E26, E29-E42). The specified contribution is GuardBench: a scenario manifest, baseline set, metric suite, and reproducibility contract for evaluating memory by action effect rather than retrieved-text relevance.

The Stage-A evidence is intentionally bounded. This paper reports source-linked implementation evidence, the canonical 0.22.2 performance snapshot, the current behavioral regression gate output, a local comparative GuardBench run, and a deterministic repeated-failure demo transcript (Ledger: E20-E25, E41-E42, E46). It also includes the external adapter contract plus Mem0 and Zep evidence-bundle paths, but it does not report full external-system GuardBench results.

The v2 paper should run live external adapters, publish raw per-scenario output bundles, run expanded redaction sweeps, and report guard-overhead p50/p95 under machine-provenance controls.

The core result is simple: Audrey saw the agent fail once. Audrey stopped it from failing twice.

11 References

References are maintained in `references.bib` (references.bib) and rendered to bibliography format at LaTeX-conversion time. This Markdown master is intended to be paired with that BibTeX file for pandoc/LaTeX conversion rather than inline-rendering bibliography entries here.

12 Working-Draft Audit Note

In-section evidence-ledger references are intentionally preserved in this v1 Markdown master as a working audit trail. The final submission polish pass should remove ledger references from prose after GuardBench Stage-B numbers land and the claim set is stable.

References

[1] Qingyao Ai, Yichen Tang, Changyue Wang, Jianming Long, Weihang Su, and Yiqun Liu. Memorybench: A benchmark for memory and continual learning in llm systems, 2026.

[2] Anthropic. Claude code hooks reference. Official documentation, 2026. Accessed 2026-05-08.

[3] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory, 2025.

[4] Cognee. Cognee: Knowledge engine for ai agent memory. GitHub repository, 2026. Accessed 2026-05-08.

[5] Ola Hungerford, Sam Morrow, and Luca Chang. Tool annotations as risk vocabulary: What hints can and can't do. Model Context Protocol Blog, March 2026. Accessed 2026-05-08.

[6] Saeid Jamshidi, Kawser Wazed Nafi, Arghavan Moradi Dakhel, Negar Shahabi, Foutse Khomh, and Naser Ezzati-Jivan. Securing the model context protocol: Defending llms against tool poisoning and adversarial attacks, 2025.

[7] LangChain. Langmem documentation. Official documentation, 2026. Accessed 2026-05-08.

[8] Zhiyu Li, Chenyang Xi, Chunyu Li, Ding Chen, Boyu Chen, Shichao Song, Simin Niu, Hanyu Wang, Jiawei Yang, Chen Tang, Qingchen Yu, Jihao Zhao, Yezhaohui Wang, Peng Liu, Zehao Lin, Pengyuan Wang, Jiahao Huo, Tianyi Chen, Kai Chen, Kehang Li, Zhen Tao, Huayi Lai, Hao Wu, Bo Tang, Zhengren Wang, Zhaoxin Fan, Ningyu Zhang, Linfeng Zhang, Junchi Yan, Mingchuan Yang, Tong Xu, Wei Xu, Huajun Chen, Haofen Wang, Hongkang Yang, Wentao Zhang, Zhi-Qin John Xu, Siheng Chen, and Feiyu Xiong. Memos: A memory os for ai system, 2025.

[9] Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of llm agents, 2024.

[10] Vasilije Markovic, Lazar Obradovic, Laszlo Hajdu, and Jovan Pavlovic. Optimizing the interface between knowledge graphs and llms for complex reasoning, 2025.

[11] Model Context Protocol. Model context protocol schema reference, protocol revision 2025-11-25. Official specification, 2025. Accessed 2026-05-08.

[12] Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. Mteb: Massive text embedding benchmark, 2023.

[13] Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024.

[14] Preston Rasmussen, Pavlo Paliychuk, Travis Beauvais, Jack Ryan, and Daniel Chalef. Zep: A temporal knowledge graph architecture for agent memory, 2025.

[15] Supermemory. supermemory documentation: Memory api for the ai era. Official documentation, 2026. Accessed 2026-05-08.

[16] Supermemory. Supermemory: Memory engine and app. GitHub repository, 2026. Accessed 2026-05-08.

[17] Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. Longmemeval: Benchmarking chat assistants on long-term interactive memory, 2025.

[18] Deshraj Yadav. Introducing the token-efficient memory algorithm. Mem0 Blog, April 2026. Accessed 2026-05-08.

[19] Zep. Graphiti: Build real-time knowledge graphs for ai agents. GitHub repository, 2026. Accessed 2026-05-08.

[20] Dongsen Zhang, Zekun Li, Xu Luo, Xuannan Liu, Peipei Li, and Wenjun Xu. Mcp security bench (msb): Benchmarking attacks against model context protocol in llm agents, 2026.