

Zero-LLM-Cost Event-Sourced Tool Failure Memory for Desktop Agents

Yonggang Weng (1, 2), Farah Akmar Anor Salim (1; corresponding author), Dongwei Yin (3, 4), Zhengguang Zhong (5), Chuntie Chen (1)

- (1) Azman Hashim International Business School, Universiti Teknologi Malaysia, Kuala Lumpur, Malaysia
- (2) Suzhou Industrial Park Oriental Huaxia Cardiovascular Health Research Institute (China Heart House), Suzhou, Jiangsu, China
- (3) Language Academy, Faculty of Social Sciences and Humanities, Universiti Teknologi Malaysia, Johor, Malaysia
- (4) Hebei Construction Material Vocational and Technical College, Qinhuangdao 066004, Hebei, China
- (5) Imperial College London, London, United Kingdom

Corresponding author: Farah Akmar Anor Salim —Email: farahakmar@utm.my —ORCID: 0000-0003-0530-783X

Author details (ORCID; email): Yonggang Weng (0009-0001-8071-5852; weng@graduate.utm.my); Farah Akmar Anor Salim (0000-0003-0530-783X; farahakmar@utm.my); Dongwei Yin (0009-0003-5183-3774; yindongwei@graduate.utm.my); Zhengguang Zhong (0000-0003-3165-1036; zhengguang.zhong23@alumni.imperial.ac.uk); Chuntie Chen (0009-0003-0059-7687; chenchuntie@graduate.utm.my)

Abstract

Desktop LLM agents repeatedly encounter the same low-level tool failures—API timeouts, encoding errors, permission denials, and tool-specific quirks—yet current frameworks either discard these lessons at session end or pay a recurring price to extract them: reflection-based learners (ERL, ExpeL) require additional LLM calls per task, while general memory layers (Mem0) assume embedding infrastructure and perform poorly on Chinese tool-error text. We present EBEAC (Event-Based Experience Auto-Capture), a zero-additional-LLM-cost mechanism that harvests tool-level trial-and-error knowledge directly from the runtime event stream. EBEAC detects “consecutive failure → retry success” sequences via the Event-Bus, abstracts them into structured experiences (trigger, diagnosis, fix) with a keyword template generator, and retrieves them through a three-layer filter (outcome matching, pattern deduplication, time-decay weighting) with a 2000-character injection budget. On a production-log replay of 196 real experiences from ten weeks of live usage (chronological 137/59 train/test split), EBEAC achieves 71.19% recall@1 and 89.36% precision at 0.97 ms retrieval latency with only 3.39% false positives—recall@3 and the cross-session prevention rate both reach 76.27%—a precision-centric alternative to lexical baselines that reach 83.05% recall at a five-fold higher false-positive rate (16.95%). A capture-side analysis shows the 196 experiences collapse into only 23 abstract patterns (88.3% collapse ratio), and 16.9% of test failures involve patterns never seen in training; the residual recall gap is therefore dominated by capture-side coverage rather than retrieval-side quality (91.8% of covered events are retrieved). EBEAC is positioned as a cost-efficient niche memory mechanism for operational, high-frequency tool quirks, complementary to—not competing with—strategic reflection-based learning.

Keywords: LLM agents; tool failure memory; event sourcing; zero-cost experience capture; desktop agents

1. Introduction

1.1 Motivation and the Cost of Existing Remedies

Tool-augmented LLM agents [1] deployed as daily-use desktop assistants face a peculiar repetition problem: the *same* low-level failures recur across sessions, days, and even weeks. A stock-data API returns empty strings for suspended stocks; a file exporter crashes on GBK-encoded paths; a WeChat plugin times out on large media uploads; a PDF tool rejects scan-only documents. These are not one-off incidents—they are *tool quirks*, deterministic properties of specific tool-implementation pairs that, once understood, can be worked around deterministically. In ten weeks of production use of WeClaw [2], 196 such trial-and-error episodes were automatically captured across 20 tools, with 133 errors observed in 1,616 production tool calls (8.2% failure rate).

Yet mainstream agent frameworks treat every session as isolated [3]: the insight “this API returns empty strings for suspended stocks” is lost at session end, and the agent re-learns it by failing again tomorrow. The cost of re-learning is not just user frustration—it is repeated LLM reasoning over failed tool calls, repeated error-correction turns, and the context window consumed by failure recovery.

Two families of solutions exist, and both charge a recurring price. **Reflection-based experience learning.** ERL [4], ExpeL [5], and AgentHER [6] generate task-level heuristics by invoking the LLM to reflect on completed trajectories. But this captures rich strategic knowledge—each reflection is an additional LLM call (2.7–5.2 calls per session in our controlled simulation), with latency and token cost that compound over thousands of sessions. Moreover, these systems operate at *task* granularity—they learn “how to plan a literature review,” not “this exporter crashes on GBK paths.”

General memory layers. Systems like Mem0 [7] and MemGPT [8] provide LLM-managed long-term memory. They assume (a) an embedding model and vector store as infrastructure, (b) LLM participation in memory extraction/consolidation, and (c) a language distribution where lexical retrieval works. Our probe study of Mem0 on WeClaw’s Chinese-dominant tool-error corpus [2] found all three assumptions violated: word-segmentation-based BM25 achieved 0.0% Chinese vocabulary coverage, raw vector retrieval saturating at 100% recall@3 (a ceiling that hides retrieval failures), and the 175 experiences collapsed to 24 distinct patterns under LLM-based deduplication—meaning the LLM consolidation step itself had almost no new signal to add.

1.2 Our Approach and Contributions

We present **EBEAC (Event-Based Experience Auto-Capture)**, a memory mechanism that is *zero-LLM-cost by construction*: it never invokes the LLM, either at capture time or at retrieval time. Instead, it exploits two structural properties of desktop agent runtimes: **event sourcing**—the EventBus already carries every tool call and result, so trial-and-error episodes can be detected *for free* by observing “failure → retry → success” sequences within a session—and **template compactness**—tool quirks are highly regular, so a keyword-based abstractor can compress an error episode into a structured (trigger, diagnosis, fix) record with a deterministic pattern label. EBEAC’s retrieval path applies three filtering layers—outcome matching, per-pattern deduplication, and time-decay weighting—and injects the top-ranked experiences into the system prompt under a 2000-character budget; the whole pipeline runs in <1 ms per event with zero additional tokens.

- **A zero-LLM-cost, event-sourced memory mechanism** for tool-level failure knowledge, designed for deployment in desktop agent runtimes without embedding infrastructure or reflection budgets (Section 4).
- **A production-log replay evaluation protocol** on 196 real experiences (137/59 chronological split) with five baselines, showing 71.19% recall@1 / 89.36% precision / 76.27% prevention at 0.97 ms, and a precision-recall trade-off against lexical baselines (Section 5–6).
- **A capture-side bottleneck analysis** quantifying template collapse (88.3% collapse ratio, 23 pat-

terns from 196 experiences) and attributing the recall gap to capture-side coverage (16.9% unseen patterns) rather than retrieval-side quality (91.8% covered-event hit rate) (Section 6.2).

- **Open-source release** of the replay pipeline, snapshot, and analysis scripts as reproducible artifacts (Section 5.5).

2. Related Work

2.1 Experience Learning and Long-Term Memory for Agents

Verbal self-reflection was first proposed by Reflexion [9], which stores feedback in episodic memory to guide retries; later learners collect experiences across tasks and prompt the LLM in order to derive reusable heuristics ExpeL [5], reflect after each task with experience accumulation and report gains over ReAct on GAIA-style benchmarks ERL [4], or relabel trajectories through hindsight experience replay AgentHER [6]. Whatever the mechanism, each experience costs additional LLM calls, and knowledge is extracted at task trajectory granularity. EBEAC differs on both axes—zero LLM calls and per-tool-call granularity—by trading expressiveness for cost. In our controlled comparison (Section 6.3), ERL and ExpeL require 2.7 and 5.2 LLM calls per session respectively, while EBEAC requires none.

Meanwhile, LLM-managed long-term memory comes in two flavors: a paged operating-system hierarchy MemGPT [8] and a production memory layer with extraction, deduplication, and retrieval over vector stores Mem0 [7]. Both are general-purpose and powerful, but they presuppose LLM participation and embedding infrastructure. Our probe study (Section 6.4) shows that on Chinese-dominant tool-error text, Mem0’s lexical components fail (0.0% BM25 vocabulary coverage) and raw vectors saturate (100% recall@3 ceiling), motivating a deterministic, infrastructure-light alternative. A third line, CoPaw [10], organizes memory into working/episodic/semantic tiers as a cognitive model—we adopt its *layering principle* but realize it with event-driven capture and deterministic retrieval rather than LLM consolidation.

2.2 Tool Selection, Failure Handling, and Positioning

Also, ITR [11] and AutoTool [12] retrieve relevant tools per step, gated by retrieval confidence; WeClaw’s PTE-FD [2] escalates tool exposure on execution failures. These systems address *which tools to expose*; EBEAC addresses *what to do when a tool fails*—a complementary problem. Tool learning surveys [13, 14, 15] catalogue failure taxonomies but not zero-cost cross-session memory. Concise context compression [16] and attention phenomena [17] motivate why a small, high-precision injection budget (2000 chars) is preferable to dumping full histories. Table 1 contrasts EBEAC with reflection-based learners and general memory layers along the dimensions that matter for desktop deployment.

Table 1. Positioning of EBEAC against reflection-based learners (ERL/ExpeL) and general memory layers (Mem0) across capture cost, granularity, infrastructure, Chinese-error robustness, expressiveness, and latency.

Aspect	Reflection-based (ERL/ExpeL)	General memory (Mem0)	EBEAC (ours)
LLM calls at capture	2.7–5.2 / session	Yes (extraction, consolidation)	0
Granularity	Task trajectory	Free-form memories	Tool call
Infrastructure	None extra	Embedding + vector store	SQLite only
Chinese tool-error robustness	N/A (LLM-based)	Weak (BM25 0.0% coverage)	Deterministic keyword
Expressiveness	High (strategic)	High	Low (tool quirks)
Latency	Minutes (reflection)	10s–100s ms	0.97 ms

3. Background and Problem Formulation

3.1 Problem Formulation and Design Requirements

Let $\mathcal{T}$ be the set of tools registered in an agent runtime, and let a *tool call* be a tuple (t, a, x, r) where $t \in \mathcal{T}$ is the tool, a an action, x arguments, and r the result (success payload or error). We define a **failure episode** as a maximal consecutive run of failed calls to the same tool within one session, followed by a success—formally, calls $c_1, \dots, c_k$ with $k \geq 2$ errors then c_{k+1} success, all for (t, a) within a time window W —and a **tool quirk** as a deterministic cause-effect relation (error signature, workaround) that holds for a specific (t, a) pair, e.g., *"stock_query returns empty strings for suspended stocks $\rightarrow$ apply float_or_none()"*.

The tool-failure memory problem is: **build a store M such that, given a new error e at time s , the retrieval $M(e, s)$ returns the quirk record that prevents e from recurring, with minimal cost, latency, and false positives.** Hence, five requirements are derived from desktop production constraints: **R1 (Zero marginal cost)**—capture and retrieval must not invoke the LLM, and memory must grow as a byproduct of normal operation; **R2 (Event-sourced)**—capture must derive from the runtime event stream already produced by the agent, not from post-hoc analysis; **R3 (Sub-ms retrieval)**—injection into the system prompt must not add perceptible latency to the agent loop; **R4 (Precision over recall)**—false memory injections mislead the LLM, and a false positive is costlier than a miss; **R5 (Infrastructure-light)**—a single SQLite file should suffice, with no embedding model, vector DB, or external service. These jointly rule out reflection-based learning (violating R1, R4) and general memory layers (violating R1, R5, and partially R4 on Chinese text), motivating the event-driven, template-based design of Section 4.

4. Method: EBEAC

4.1 Event-Driven Auto-Capture

EBEAC consists of three components connected through the runtime’s EventBus—the **ExperienceAutoRecorder** (event-driven capture), the **ExperienceStore** (hybrid storage and three-layer retrieval), and the **ExperienceRecallTool** (system-prompt injection surface)—as shown in Figure 1. The recorder subscribes to TOOL_RESULT events at non-blocking priority, and a ToolFailureTracker is maintained per (session_id, tool_name) key. On error (status = error), the tracker appends the failure {error (truncated to 500 chars), timestamp, action, arguments (truncated to 100 chars each)}; on success (status = success), if the tracker holds ≥ 2 accumulated failures (configurable MIN_FAILURES = 2), the episode is extracted and the tracker reset. Failures older than 300 s are discarded from the tracker, so episodes are temporally local and never span sessions, and a tracker keeps at most 10 failures to prevent unbounded accumulation. The trigger condition encodes R2 (event-sourced): an experience is *only* recorded when the agent itself recovered from the failure (failure $\rightarrow$ retry $\rightarrow$ success), which (a) provides a positive label that the quirk is workaround-able and (b) filters out systemic outages where retries also fail. Algorithm 1 formalizes the capture loop.

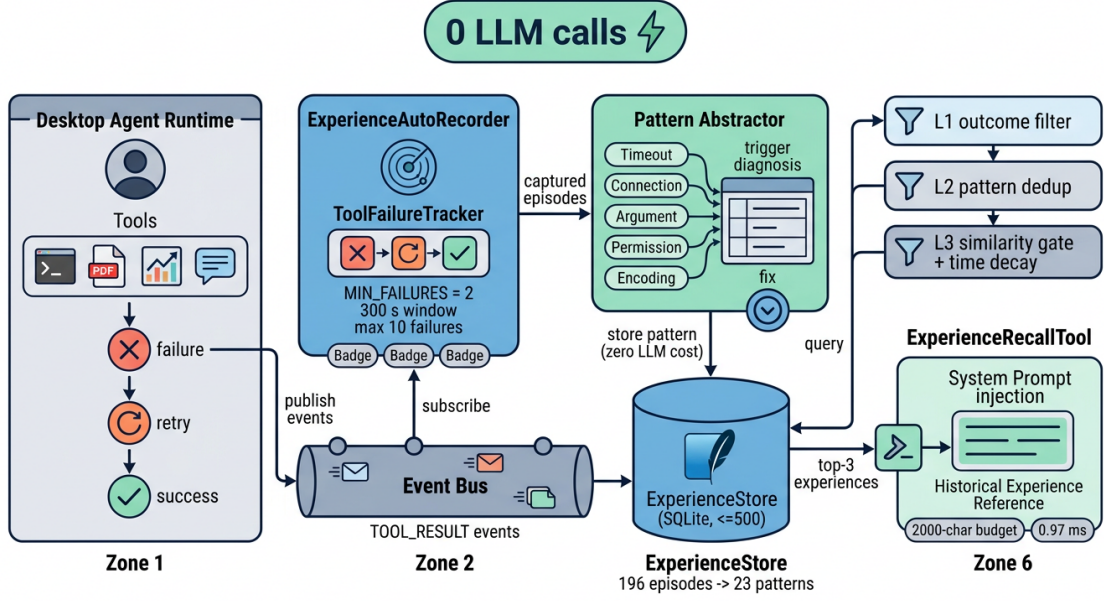


Fig-

ure 1. EBEAC architecture. The ExperienceAutoRecorder captures failure $\rightarrow$ retry $\rightarrow$ success episodes as passive EventBus subscriber; the ExperienceStore abstracts, stores, and retrieves experiences; the ExperienceRecallTool injects top-ranked experiences into the system prompt under a 2000-character budget.

Algorithm 1: Event-Driven Experience Capture

Input: TOOL_RESULT event stream

Output: experience records

```

trackers  $\leftarrow$ 
for each event e (in order):
    key  $\leftarrow$  e.session_id e.tool_name
    if e.status = error:
        trackers[key].add_failure(e)
    else if e.status = success and trackers[key] has MIN_FAILURES failures:
        failures  $\leftarrow$  trackers[key].check_and_reset()
        record  $\leftarrow$  generate_experience(tool_name, action_name, failures)
        ExperienceStore.record(record)

```

Why this is zero-cost: the EventBus already exists for audit logging; the recorder is a passive subscriber. Detection is pure state-machine logic—no LLM call, no embedding, no network I/O at capture time.

4.2 Zero-Cost Pattern Abstraction

Each captured episode is compressed into a structured record with four fields, all generated deterministically from the raw episode (templates and examples are quoted verbatim from the Chinese-dominant production deployment; English glosses are ours); Table 2 lists the four fields.

Table 2. The four structured fields of an EBEAC experience record, with the deterministic generation template and a verbatim production example for each.

Field	Template	Example
trigger	"{tool}.{action} 连续失败 {k} 次后重试成功" ("{tool}.{action} succeeded on retry after {k} consecutive failures")	paper_lifecycle.export 连续失败 3 次后重试成功
diagnosis	" 工具 {tool} 调用出错: {err1}; {err2}; {err3}" ("tool {tool} call failed: ..."), first 3 errors, 100 chars each	工具 stock_query 调用出错: ...empty result...
fix_summary	" 经过 {k} 次重试后成功执行 {tool}.{action}" ("{tool}.{action} executed successfully after {k} retries")	经过 3 次重试后成功执行 paper_lifecycle.export
abstract_pattern	" 工具试错模式: {type1}+{type2}..." ("tool trial-and-error pattern: ...") from keyword scan	工具试错模式: 参数修正

The abstract pattern is generated by scanning the concatenated error text for five keyword families, each pairing a small set of signal terms with a fixed pattern tag (Table 3):

Table 3. The five keyword families used by the pattern generator, with their signal terms, the resulting pattern tags, and the per-tool fallback for episodes carrying no generic error signal.

Keyword family	Signal terms	Pattern tag
Timeout	timeout, timed out, 超时	超时重试 (<i>timeout-retry</i>)
Connection/network	connection, connect, network, 连接, 网络	连接重试 (<i>connection-retry</i>)
Argument/parameter	invalid, argument, parameter, 参数	参数修正 (<i>argument-correction</i>)
Permission	permission, denied, 权限, 拒绝	权限问题 (<i>permission-issue</i>)
Encoding/parsing	encoding, parse, decode, 编码, 解析	编码/解析容错 (<i>encoding/parsing-tolerance</i>)
(none matched)	—	"{tool} 工具重试成功" ("{tool} retry succeeded"), per-tool fallback

Notably, a pattern is the conjunction of all matched families (e.g., 连接重试 + 超时重试, connection-retry + timeout-retry); the per-tool fallback captures quirk-like episodes with no generic error signal. This generator is deterministic, language-aware for Chinese keywords, and costs no LLM calls. The diagnosis field is the only one carrying raw error text, which is what makes later LIKE-based retrieval effective (Section 4.3).

4.3 Storage and Three-Layer Retrieval

Storage. Experiences are persisted in SQLite (WAL mode) with a ChromaDB vector index as an optional enhancement. When ChromaDB or the embedding model is unavailable—the default in our deployment

—retrieval falls back to SQLite LIKE full-text search. The store caps retained experiences at 500 entries (LRU by recency).

Retrieval. Given a query text (the new error), retrieval proceeds as:

1. **Candidate retrieval** ($\text{top_k} \times 3$): vector search when available; otherwise SQLite LIKE over `trigger`, `diagnosis`, `fix_summary`, `abstract_pattern`, `tool_names`.
2. **Layer 1—Outcome filtering**: drop experiences marked `outcome = failure` (only *successful* workarounds are replayed).
3. **Layer 2—Pattern deduplication**: keep only the most recent experience per `abstract_pattern`. This prevents the same quirk from flooding the injection (critical given the collapse ratios measured in Section 6.2) and implicitly enforces “one representative memory per pattern.”
4. **Layer 3—Similarity gating**: drop candidates below `min_similarity = 0.55`, calibrated for bge-base-zh-v1.5’s score distribution (positive mean 0.619, negative mean 0.503). The gate operates on the *raw* similarity so that time decay never raises the effective threshold. In the SQLite fallback mode used in our experiments, LIKE-matched candidates carry a fixed similarity proxy of 0.7, which clears the gate; the gate binds only when vector search is enabled.
5. **Time-decay re-ranking**: candidates within 30 days get weight 1.0, 31–90 days 0.8, older 0.6; then sort by weighted score and take the top-k (default $k = 3$).

Step 3 (deduplication) primarily realizes R4 in the SQLite mode used here; the calibrated gate (step 4) binds in vector mode, and both together suppress false positives, which we measure as 3.39% on real logs (Section 6.1) versus 16.95% for unfiltered lexical baselines. Hence, we serialize the retrieved experiences into a “Historical Experience Reference” module injected into the system prompt, with a hard budget of **2000 characters** (the entire corpus is too large to inject; the budget bounds prompt-dilution effects documented by Liu et al. [17]). The recall tool exposes `recall`, `list_experiences`, `forget`, and `record` actions, so the agent can query memory on demand and the user/agent can curate it.

4.4 Injection, Complexity, and Cost Analysis

- **Capture**: $O(1)$ per event (dictionary lookup + append); the recorder adds no network or model I/O.
- **Retrieval**: SQLite LIKE scan over ≤ 500 rows with early termination; measured end-to-end mean latency **0.97 ms** (Section 6.1), with no additional token budget: the injected content stays within the dedicated 2000-char injection budget and does not compete with other system-prompt components for budget.
- **Storage**: one SQLite file; the full 196-experience corpus is ~ 128 KB.
- **LLM cost**: exactly zero calls at both capture and retrieval; the only cost is the 2000 chars we inject, which are deterministic and cache-friendly for prompt caching.

5. Experiment Setup

5.1 Production Data and Replay Protocol

Snapshot. A consistency snapshot of the production `experiences.db` was exported on 2026-08-08 via the SQLite backup API with the WAL merged, covering experiences created between 2026-05-30 and 2026-08-07 (70 days of live usage): 197 experiences in the raw snapshot (196 from the `tool_retry` source and 1 manual record), all with `outcome = success`; the replay dataset uses the 196 auto-captured experiences (the manual record carries no tool association and is excluded by the exporter), spanning 20 distinct tools and 7 error types. The 196 experiences map to **23 distinct abstract patterns**.

Time-split. Following the chronological protocol of exp9 [2], we order items by `created_at` and split them 70/30 $\rightarrow$ **137 train / 59 test** error events. Train experiences populate the memory store; each test

error event queries the store and is judged against its ground-truth fix pattern. Figure 2 shows the replay protocol.

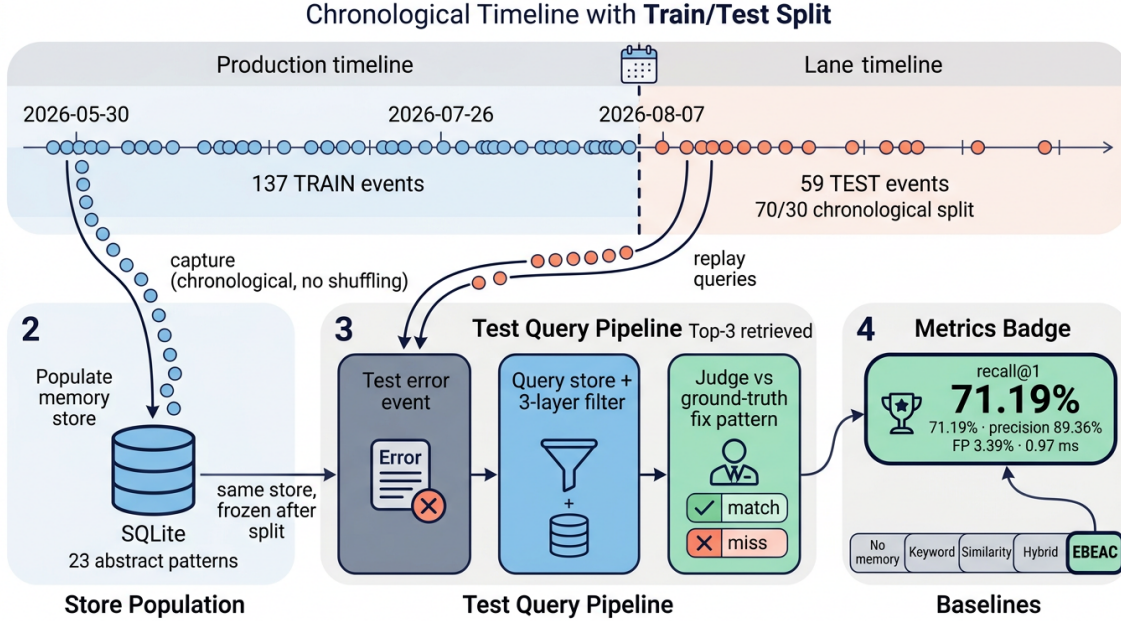


Figure 2. Chronological 137/59 time-split replay protocol. The first 137 error events (2026-05-30 to 2026-07-26) populate the memory store; each of the 59 test events queries the store and is judged against its ground-truth fix pattern.

Baselines. All five retrieval methods are evaluated under the same query construction and evaluation: (1) **no memory**—an empty store serving as the lower bound; (2) **keyword memory**—substring matching over the corpus without three-layer filtering; (3) **similarity memory**—edit-similarity matching without filtering; (4) **hybrid memory**—the union of keyword and similarity; and (5) **ExperienceStore (EBEAC)**—the full pipeline of Section 4.3 through the real ExperienceStore.recall() code path (SQLite fallback mode, i.e., without ChromaDB, matching the production default).

5.2 Metrics and Analysis Definitions

Thus, for each test error event, retrieval is judged at $\text{top}_k = 3$: **recall@1/recall@3** measure whether the ground-truth fix pattern appears in the top-1/top-3 retrieved experiences; **precision** is the fraction of retrieved (positive) predictions whose pattern matches the ground truth; **coverage rate** is the fraction of test events for which the method returns any candidate; **prevention rate** is the fraction of test errors that would be prevented (a retrieved experience whose pattern matches the ground truth and whose tool matches); **false-positive rate** counts retrieved candidates injected into the prompt that do not match the ground truth (per-event, any non-matching injection counts); **miss rate** is $1 - \text{recall@3}$; and **average latency** is measured per retrieval (ms).

To quantify the upper bound imposed by the capture end, we analyze the same snapshot: the **collapse ratio** is $1 - \frac{|\text{distinct abstract_pattern}|}{|\text{experiences}|}$; the **field-level template rate** is $1 - \frac{|\text{distinct after digit normalization}|}{n}$ for trigger, fix_summary, diagnosis, and the concatenated full text; the **counterfactual granularity levels** are (A) tool::pattern, (B) tool::error_type::action (derived with the exporter’s derive_error_type/derive_action), and (C) full-text fingerprint after digit normalization—measuring how much the candidate pool would expand if the abstractor were finer-grained; **test-pattern coverage** is the fraction of test events whose fix pattern was already seen in training (exact-match set membership), the theoretical ceiling for pattern-exact retrieval, plus the same coverage under the fine-grained (B) scheme; and the **gap decomposition** separates total miss rate into a capture-side gap (unseen patterns) and a retrieval-side loss (covered but not retrieved).

5.3 Controlled Cost Comparison

The controlled protocol of the WeClaw main paper [2] is reused: 50 simulated multi-session workflows, 15–25 tool calls each, with the real observed failure rate of 8.2% (133 errors in 1,616 production calls). Methods: Raw Error Log (no abstraction), ERL (2.7 LLM calls/session), ExpeL (5.2 calls/session), and EBEAC (0 calls). Metrics: patterns captured, recall over 57 known patterns, precision, prevention rate, per-session LLM calls.

5.4 Memory-Layer Selection Probes

As negative evidence motivating the design, we report a probe study (previously documented in the WeClaw engineering blog [2]) of Mem0-style memory on the earlier corpus available when the probes ran (175 experiences, 2026-05-30–07-31; the final snapshot adds the remaining week’s 21 episodes): (i) BM25 Chinese vocabulary coverage for two segmenters vs. a JIEBA reference; (ii) bare vector retrieval recall@3 (ceiling detection); (iii) LLM-based deduplication collapse on 175 experiences.

5.5 Reproducibility Artifacts

In fact, all experiments are reproducible from the `weclaw_adaptive_runtime/benchmarks` package: `make_exp9_snapshot.py` (snapshot), `export_experience_store_replay.py` (dataset), `exp9_ebeac_log_replay.py` (replay + baselines), `exp9b_capture_bottleneck_analysis.py` (capture-side analysis). The snapshot, dataset JSON, results JSON, and analysis report are archived in a public Zenodo deposit at <https://doi.org/10.5281/zenodo.22839726>, frozen from the git tag `paper-ebeac-eval-v1` of the WeClaw repository [18]: the experiment scripts import from that frozen revision, so subsequent evolution of the main branch cannot alter any reported figure.

6. Results

6.1 Replay Results and the Precision–Recall Trade-off

Table 4 reports the five-baseline replay over the 137/59 chronological split (all values in %, latency in ms); the rows share the same query construction and the same 59 test events, so the methods are directly comparable.

Table 4. Production-log replay (196 experiences; 137 train / 59 test; chronological split).

Method	Recall@1	Recall@3	Precision	Coverage	Prevention	FP rate	Miss rate	Latency
No memory	0.00	0.00	—	0.00	0.00	0.00	100.0	—
Keyword memory	83.05	83.05	83.05	100.0	83.05	16.95	16.95	3.87
Similarity memory	83.05	83.05	83.05	100.0	83.05	16.95	16.95	3.86
Hybrid memory	83.05	83.05	83.05	100.0	83.05	16.95	16.95	7.71
ExperienceStore (EBEAC)	76.27	76.27	89.36	79.66	76.27	3.39	23.73	0.97

† Bootstrap 95% CI (B=100k, percentile method): EBEAC recall@1=71.19 [59.32, 83.05], recall@3=76.27 [64.41, 86.44], precision=89.36 [79.59, 97.78], FP rate=3.39 [0.00, 8.47]; lexical baselines 83.05 [72.88, 91.53] for recall/precision/prevention, FP rate=16.95 [8.47, 27.12].

Three observations follow directly from the table. **1. Precision-centric operation (R4):** EBEAC reaches 71.19% recall@1 (95% CI: 59.32–83.05)—86% of the lexical baselines’ 83.05% (95% CI: 72.88–91.53)—while cutting the false-positive rate five-fold (3.39% [0.00, 8.47] vs. 16.95% [8.47, 27.12]) and improving precision from 83.05% to 89.36% [79.59, 97.78]. Yet, the lexical baselines achieve perfect coverage (100.0%) by injecting *every* substring match; EBEAC’ s coverage (79.66%) is set by the SQLite LIKE candidate search itself, then compressed by Layer 2 (per-pattern dedup); Layer 3 (similarity gate) is not binding in the SQLite fallback mode used here (Section 4.3). **2. Meanwhile, latency stays sub-millisecond:** 0.97 ms vs. 3.86–7.71 ms for lexical methods—the deduplication (Layer 2) shortens the candidate list before scoring, while the full corpus is scored by the lexical baselines per query; at 0.97 ms the injection adds no perceptible latency to the agent loop (R3). **3. Thus, the miss rate decomposes cleanly:** the 23.73% miss rate is dominated by events whose fix pattern never appeared in training (16.9%, see Section 6.2); among the 83.1% of test events whose pattern *was* seen, EBEAC retrieves the correct fix 91.8% of the time (76.27% / 83.1%), i.e., the retrieval side operates near its coverage ceiling.

But why do lexical baselines reach higher recall at five times the false-positive rate? Two active mechanisms differ, plus a structural note on the filter layers: **no pattern deduplication (Layer 2 absent)**—the same quirk appears once per episode, and with 23 patterns condensed from 196 experiences (Section 6.2), the three lexical methods inject up to ~8.5 duplicates per distinct quirk, skewing the agent toward repeated (already known) workarounds and crowding out rarer patterns; **candidate construction**—lexical baselines score the *full corpus* and inject every match, whereas EBEAC’ s LIKE query (Section 4.3) returns at most top-9 candidates ($\text{top_k} \times 3$) over the four text fields, so unrelated partial matches rarely enter the ranked list; on Chinese-dominant error text, shared substrings such as 调用出错 or 重试 match unrelated episodes in the full-corpus scan, producing the 16.95% FP rate; and a **structural note (Layer 1 and Layer 3)**—Layer 1 (outcome filtering) is inert on this all-success corpus, and Layer 3 (similarity gate) is not binding in the SQLite fallback mode (fixed 0.7 proxy; Section 4.3), both binding in vector mode or once failure-outcome experiences accumulate (Section 7.2).

Nevertheless, under requirement R4 (a false positive is costlier than a miss), we find EBEAC’ s 71.19%/3.39% operating point preferable to 83.05%/16.95%: an injected but wrong workaround actively misleads the LLM into following a fix that does not apply, while a miss simply preserves the status quo of one more failed attempt. The covered-event hit rate of 91.8% (Section 6.1, observation 3) shows this precision-centric choice costs only 6.78 percentage points of recall (23.73% miss = 16.9% capture-side + 6.78% retrieval-side).

6.2 Capture-Side Bottleneck Analysis

Table 5 quantifies the capture end of the pipeline on the same 196-experience snapshot, separating what the abstraction layer collapses from what the events themselves already contain.

Table 5. Capture-side analysis (196 experiences, 2026-05-30 ~ 08-07). Figure 3 visualizes the counterfactual pool (panel a), the coverage paradox (panel b), and the gap decomposition (panel c).

Metric	Value
Experiences / distinct abstract patterns	196 / 23
Collapse ratio	88.3%
Template rate —trigger	77.0%
Template rate —diagnosis	42.4%
Template rate —fix_summary	77.0%
Template rate —full text (digit-normalized)	35.7%
Counterfactual pool —current (pattern)	23
Counterfactual pool —A (tool::pattern)	28 (1.2×)
Counterfactual pool —B (tool::error_type::action)	55 (2.4×)
Counterfactual pool —C (full-text fingerprint)	126 (5.5×)

Metric	Value
Test coverage —current scheme	49/59 = 83.1%
Test coverage —fine-grained (B)	38/59 = 64.4%
Unseen-pattern gap (capture-side)	10/59 = 16.9%
Covered-event hit rate (retrieval-side)	91.8%
Recency distribution (≤ 30 / 31–90 / >90 days)	31.1% / 68.9% / 0.0%

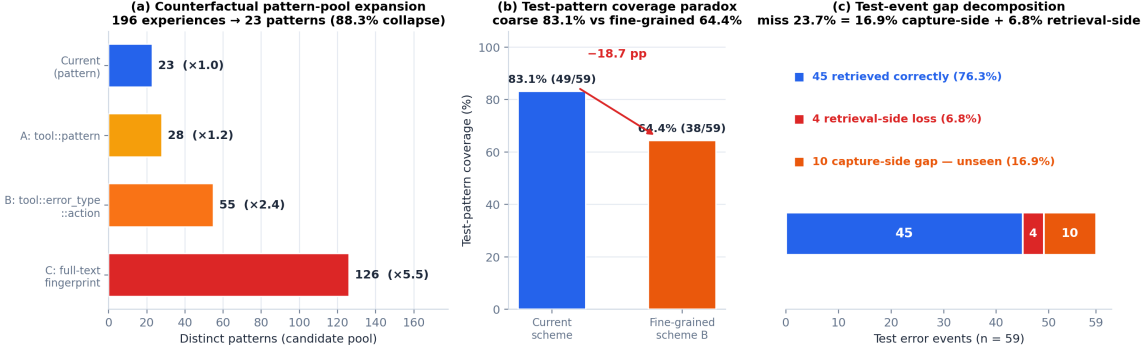


Fig-

ure 3. Capture-side analysis. Panel (a) shows the counterfactual candidate pool under three finer granularity schemes; panel (b) contrasts test-pattern coverage of the current scheme (83.1%) against the fine-grained scheme (64.4%); panel (c) decomposes the total miss rate into capture-side and retrieval-side gaps.

Five findings follow. **1. The pattern generator is the source of collapse:** trigger and fix_summary are 77.0% templated because both are emitted by the same fixed templates (Section 4.2); diagnosis is only 42.4% templated because it carries raw error text—which is exactly the field that makes LIKE retrieval work; the full-text template rate of 35.7% shows the raw material is *not* collapsed, so collapse is imposed by the abstraction layer, not by the underlying events. **2. Counterfactual granularity:** a finer abstractor would expand the candidate pool from 23 to 28 (A), 55 (B), or 126 (C) patterns—up to 5.5 $\times$ —and the head is heavily concentrated: 65.3% of experiences (54/42/32 of 196) are accounted for by shell, paper_lifecycle, and wechat alone. **3. The coarse-grained coverage paradox:** nevertheless, test coverage under the current scheme (83.1%) *exceeds* that under the fine-grained scheme B (64.4%), because exact pattern membership requires the fix pattern to have appeared identically in training, and fine-grained patterns (tool + error_type + action) are strictly harder to re-encounter identically; coarse patterns maximize exact-match coverage, while fine patterns maximize informativeness—the tension is analyzed in Section 7.1. **4. Gap decomposition:** total miss 23.73% = 16.9% (capture-side: patterns unseen in training) + 6.78% (retrieval-side: covered but not retrieved), so the capture side is the dominant bottleneck, not retrieval quality. **5. Also, recency weighting is inert in this corpus:** 68.9% of experiences fall in the 31–90-day band and none beyond 90 days, so the time-decay weights 1.0/0.8/0.6 mostly reduce to 0.8 in practice; the mechanism only binds as the corpus matures past 90 days.

6.3 Controlled Cost Comparison

Table 6 reuses the controlled protocol of the WeClaw main paper [2]—50 simulated multi-session workflows, 15–25 tool calls each, real failure rate 8.2% (133 errors in 1,616 production calls), 57 known patterns—and reports the per-session cost of each method along with what it captures.

Table 6. Experience-learning cost over 50 simulated sessions (8.2% real failure rate).

Method	LLM calls / session	Patterns captured	Recall@K	Precision	Prevention rate
Raw Error Log	0.0	4	7.0%	100%	5.3%
ERL [4]	2.7	369	—†	—†	—†
ExpeL [5]	5.2	381	—†	—†	—†
EBEAC (ours)	0.0	11	17.5%	54.5%	16.7%

† ERL/ExpeL recall/precision/prevention cells are omitted deliberately: the two reflection baselines were evaluated in our simulation under a substantially higher configured failure rate (18% vs. our real 8.2%), and their pattern counts are raw reflection outputs rather than matches against the 57 known-pattern ground truth; Table 6 therefore keeps the comparison on the cost axis only. Full simulation provenance is documented in Appendix A.

In fact, EBEAC captures 11 of 57 known patterns (17.5%) at zero LLM cost with 54.5% precision (about half the captured patterns are genuine recurring errors) and prevents 16.7% of cross-session errors. In a typical session of 20 tool calls this prevents 0.3 errors ($20 \times 8.2\% \times 16.7\%$) with no LLM intervention. Raw error logging is cheaper still (0 calls) but captures only 4 patterns (7.0%) because unabstracted logs are never matched against later errors. The two axes are orthogonal: reflection adds strategic coverage at 2.7–5.2 calls/session; EBEAC adds zero-cost operational coverage of high-frequency quirks.

6.4 Memory-Layer Probes (Mem0-Style Baseline)

However, three probes on WeClaw’ s Chinese-dominant tool-error corpus [2] motivate the deterministic design. **1. BM25 Chinese vocabulary coverage is 0.0%:** zero Chinese terms in the error corpus were matched by word-segmentation-based BM25 indices (two segmenters tested vs. a JIEBA reference)—the lexical assumptions of general memory layers fail outright on this language distribution. **2. Bare vector retrieval shows a 100% recall@3 ceiling:** perfect recall@3 is saturated by unfiltered vector search, a ceiling that hides retrieval failures; with everything retrievable, nothing is discriminated, and the ceiling also masks the FP problem that EBEAC’ s gates address. **3. LLM-based deduplication collapses 175 → 24 patterns (86.3%):** asking an LLM to consolidate the corpus yields 24 distinct patterns from 175 experiences—nearly identical to EBEAC’ s deterministic 23 patterns from 196 experiences—so the LLM consolidation step adds almost no new signal, validating the keyword-template abstractor’ s expressiveness ceiling as a property of the *data*, not of the abstraction method. That is, probe 3 is the empirical core of the zero-cost argument: if an expensive LLM consolidation yields the same pattern space as a keyword scanner, the LLM cost is not buying information—it is paying for the same collapse.

7. Discussion

7.1 The Coarse-Grained Coverage Paradox and the Zero-Cost Boundary

Also, the capture-side analysis exposes a design tension rather than a simple deficiency. More informative memories (55 vs. 23 patterns) would be produced by a finer abstractor (scheme B, `tool::error_type::action`) but *lowers* exact-match test coverage from 83.1% to 64.4% (Section 6.2, finding 3). Our mechanism is exact-match retrieval: a test error is only recallable if its pattern appeared identically in training. Coarse patterns maximize the probability of re-encountering a pattern; fine patterns maximize the value of a match when it occurs. This paradox explains why the measured 17.5% pattern-capture recall in the controlled simulation (Section 6.3) and the 16.9% capture-side gap in the replay (Section 6.2) are structural rather than implementation artifacts: they are the price of a vocabulary—the pattern taxonomy—that is deliberately small. Still, two levers exist, in increasing order of cost:

- **Error-code refinement (zero-cost).** Decompose the five keyword families (Section 4.2) into finer signal terms (e.g., distinguishing `timeout` from `timed out` + `连接超时`, `connection-timeout`), increasing pattern granularity without touching the capture pipeline. This trades a lower coverage ceiling for higher informativeness within the bounds of Table 5’s counterfactuals.
- **Selective LLM summarization (bounded cost).** For the ~35.7% of episodes whose full text is *not* templated (Table 5), an on-demand LLM summary could enrich `diagnosis`—but probe 3 (Section 6.4) shows the consolidation adds little pattern-level signal, so the budget is better spent on error-code refinement first.

The third lever—expanding the pattern vocabulary itself, e.g., by per-tool sub-patterns for the three dominant tools (65.3% of experiences)—sits exactly on the paradox: it improves recall on the head while worsening it on the tail, and the measured 83.1% coverage suggests the current operating point already sits near the Pareto frontier for exact-match retrieval.

We can now state EBEAC’s design boundary precisely: **inside the boundary (EBEAC’s domain)** lie high-frequency, deterministic tool quirks whose fix is stable across sessions—the 23 patterns that recur weekly; for these, keyword templates plus exact-match retrieval are *sufficient*, since 91.8% of covered events are retrieved and the collapse analysis shows the LLM would find nothing new (probe 3). **Outside the boundary (reflection’s domain)** lies strategic, one-of-a-kind, or evolving knowledge—multi-step planning heuristics, novel error types, fixes that change over time—where ERL/ExpeL-style reflection earns its 2.7–5.2 calls/session.

Moreover, the comparison to lexical baselines sharpens the boundary from the other side: at 83.05% recall the keyword method is *not* free—it pays 16.95% false positives, i.e., it trades prompt pollution for recall. EBEAC’s three filtering layers are therefore not optional decoration; they are what makes a zero-cost mechanism safe to deploy. The cost of the boundary itself is the 16.9% capture-side gap, which we argue is the correct price for determinism and precision (R4).

7.2 Threats to Validity

- **Single-user, single-system deployment.** All 196 replayed experiences come from one user’s WeClaw instance over 70 days. Tool quirks and error distributions are user- and system-specific (`shell`, `paper_lifecycle`, and `wechat` dominate at 65.3%); generalization to other deployments is untested. The replay protocol itself is tool-agnostic, but the measured numbers are not population statistics.
- **Small evaluation set.** 59 test events bound the resolution of all metrics (e.g., the 6.78% retrieval-side loss is 4 events). Percent-level claims should be read as point estimates; bootstrap 95% CIs (B=100k, percentile method) are reported in Table 1.
- **Pattern-exact judgment.** Ground truth is pattern membership; a retrieved experience with the same pattern but a different concrete fix would be counted as correct. The 91.8% covered-event hit rate is therefore an upper bound on *fix-level* accuracy.
- **No new tools in the test period.** Because the split is chronological, test-period events overwhelmingly involve tools and patterns already seen in training; the 16.9% unseen-pattern gap may underestimate the capture-side gap for genuinely novel tools.
- **SQLite-LIKE dependency.** Retrieval quality rests on `diagnosis` carrying raw error text (42.4% non-templated); deployments that truncate or sanitize error text would degrade recall, though the 500-char truncation in capture (Section 4.1) is already a mild form of this.
- **Partially inert filter layers.** Layer 1 (outcome filtering) is structurally inert on this all-success corpus, and Layer 3 (similarity gate) is not binding in the SQLite fallback mode (fixed 0.7 proxy; Section 4.3). The measured 3.39% FP rate is therefore attributable to candidate construction and

Layer 2 deduplication; the protective effect of the two inert layers is untested on data where they would bind.

8. Conclusion

We presented EBEAC, a zero-additional-LLM-cost, event-sourced memory mechanism for tool-level failure knowledge in desktop LLM agents. On a production-log replay of 196 real experiences from ten weeks of live usage (137/59 chronological split), EBEAC achieves 71.19% recall@1 (95% CI: 59.32–83.05), 89.36% precision [79.59, 97.78], and 76.27% cross-session prevention at 0.97 ms with 3.39% false positives [0.00, 8.47]—a precision-centric operating point that lexical baselines reach only at a five-fold higher false-positive rate. A capture-side analysis shows the mechanism operates near its structural ceiling: 88.3% of the corpus collapses into 23 deterministic patterns, the capture side accounts for 16.9 of the 23.73 percentage points of miss, and covered events are retrieved 91.8% of the time. A controlled simulation confirms the cost axis (0 LLM calls vs. 2.7–5.2 for reflection-based learners), and a Mem0-style probe shows the general-memory alternative fails on Chinese tool-error text (0.0% BM25 coverage) while its LLM consolidation finds no more patterns than the keyword scanner (24 vs. 23).

After all, EBEAC is deliberately positioned as a niche mechanism: it owns the high-frequency, deterministic tail of tool quirks and cedes strategic, evolving knowledge to reflection-based learning. Also, its design—event-driven capture, deterministic templates, three-layer gating, SQLite-only infrastructure—makes it deployable in any agent runtime that already has an event bus, with no embedding model, no vector store, and no reflection budget. The coarse-grained coverage paradox and the capture-side gap identify the concrete next step: a fine-grained pattern taxonomy guided by error-code refinement, evaluated against the Pareto frontier measured here.

Declarations

Funding The authors did not receive support from any organization for the submitted work.

Competing Interests The authors have no relevant financial or non-financial interests to disclose.

Ethics Approval Not applicable. This study does not involve human participants or animals; all data derive from the production logs of an open-source software system.

Data Availability The production-log replay dataset (196 tool-failure experiences, snapshot experiences_20260808), the 137/59 temporal-split results, the dataset schema (Appendix B), and the capture-side analysis are publicly archived on Zenodo at <https://doi.org/10.5281/zenodo.22839726> (CC BY 4.0), frozen from git tag `paper-ebac-eval-v1` (commit `a0ae540`). All records were desensitized before release: third-party e-mail addresses were redacted and developer-machine path segments normalized (Section 5.5).

Code Availability The replay pipeline and all analysis scripts are included in the same Zenodo archive (<https://doi.org/10.5281/zenodo.22839726>), frozen from git tag `paper-ebac-eval-v1`. Full end-to-end reproduction additionally requires the WeClaw runtime at the same tag (Section 5.5).

Authors' Contributions Yonggang Weng led the study, contributing to conception and design, system implementation, data collection, experiments, analysis, and writing of the original draft. Farah Akmar Anor Salim provided supervision, contributed to conception, and reviewed and edited the manuscript. Dongwei Yin, Zhenguang Zhong, and Chuntie Chen participated in data collection and verification of the experimental results. All authors read and approved the final manuscript.

Declaration of Generative AI During the preparation of this manuscript the authors used generative AI tools solely for language polishing. All scientific content, system design, experimental protocol, data analysis, and conclusions are the sole responsibility of the authors, who reviewed and edited all AI-assisted text and take full responsibility for the content of the publication.

References

1. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: ReAct: Synergizing reasoning and acting in language models. In: *International Conference on Learning Representations (ICLR)* (2023)
2. Weng, Y.: WeClaw: An adaptive runtime for desktop LLM agents with failure-driven tool escalation and event-driven experience capture (2026). Manuscript under review
3. Harrison, C.: LangChain: Building applications with LLMs through composability (2022). <https://github.com/langchain-ai/langchain>
4. Allard, M.-A., Teinturier, A., Xing, V., Viaud, G.: Experiential reflective learning for self-improving LLM agents. arXiv:2603.24639 (2026)
5. Zhao, A., Huang, D., Xu, Q., Lin, M., Liu, Y.-J., Huang, G.: ExpeL: LLM agents are experiential learners. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)* (2024)
6. Ding, L.: AgentHER: Hindsight experience replay for LLM agent trajectory relabeling. arXiv:2603.21357 (2026)
7. Chhikara, P., Khant, D., Aryan, S., Singh, T., Yadav, D.: Mem0: Building production-ready AI agents with scalable long-term memory. arXiv:2504.19413 (2025)
8. Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S.G., Stoica, I., Gonzalez, J.E.: MemGPT: Towards LLMs as operating systems. In: *International Conference on Learning Representations (ICLR)* (2024)
9. Shinn, N., Cassano, F., Gopinath, A., Shakkhotra, K., Labash, A., Karpas, E.: Reflexion: Language agents with verbal reinforcement learning. In: *Advances in Neural Information Processing Systems (NeurIPS)* (2023)
10. CoPaw Contributors: CoPaw: Contextual pawprint AI agent framework (2025). <https://github.com/archibate/CoPaw>
11. Franko, U.: Dynamic system instructions and tool exposure for efficient agentic LLMs. arXiv:2602.17046 (2025)
12. Zou, J., Yang, L., Qi, Y., et al.: AutoTool: Dynamic tool selection and integration for agentic reasoning. arXiv:2512.13278 (2025)
13. Qu, C., Dai, S., Wei, X., Cai, H., et al.: Tool learning with large language models: A survey. arXiv:2405.17935 (2024)
14. Patil, S.G., Zhang, T., Wang, X., Gonzalez, J.E.: Gorilla: Large language model connected with massive APIs. arXiv:2305.15334 (2023)
15. Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., et al.: ToolLLM: Facilitating large language models to master 16000+ real-world APIs. arXiv:2307.16789 (2023)
16. Xu, Y., Feng, Y., Mu, H., Hou, Y., Li, Y., Wang, X., et al.: Concise and precise context compression for tool-using language models. In: *Findings of the Association for Computational Linguistics (ACL)* (2024)
17. Liu, N.F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., Liang, P.: Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* **12**, 157-173 (2024)
18. Weng, Y.: WeClaw: An open-source desktop LLM agent runtime (2026). <https://github.com/wyg5208/weclaw>

Appendix A. Comparison Provenance: ERL and ExpeL Simulation

Table 3 compares methods on the cost axis only. This appendix documents the full provenance of the two reflection baselines so that the omitted cells can be interpreted rather than inferred.

Simulation environment. Both baselines were executed as API-based simulations of their published reflection protocols (reproduction script `exp2_ebeac_cost.py`, released alongside this paper), over 50 synthetic multi-session workflows of 15–30 tool calls each (random seed 42), with a configured failure rate of 18%. Because each session contains structured failure→retry→success episodes, the realized failure rate of the synthetic stream is 43.8% (513 failures over 1,172 calls)—substantially higher than the 8.2% real production failure rate (133 errors in 1,616 calls) used for the EBEAC row. The reflection model was DeepSeek-V4 Flash.

ERL-style simulation. Following the per-session reflection design of ERL, the simulator issues 1–3 reflection calls per session (proportional to the number of failures, capped at 3) and parses the responses for rule statements. Measured: 135 LLM calls (2.7 per session), 35,767 tokens (715 per session), and 369 extracted rule lines.

ExpeL-style simulation. Following ExpeL’s pair-based rule extraction, the simulator issues one extraction call per failure→success pair with at least two consecutive failures, plus a consolidation follow-up call when extraction yields multiple candidate rules. Measured: 260 LLM calls (5.2 per session), 43,409 tokens (868 per session), and 381 extracted rules.

EBEAC on the same synthetic stream. The production recorder, attached to the identical event stream, captured 197 raw episodes in 0.009 s with zero LLM calls.

Why Table 3 reports only call counts for the reflection rows. (i) The failure regimes differ: the reflection rows were measured under the 18% configured synthetic rate, while the EBEAC row was measured against the real 8.2% production rate within the controlled protocol with 57 known patterns. (ii) The 369/381 counts are free-form reflection outputs, not matches against the 57-pattern ground truth, whereas EBEAC’s 11 captured patterns (17.5% recall, 54.5% precision, 16.7% prevention) are ground-truth-matched. Comparing recall or precision across these cells would conflate different ground truths and failure regimes, so Table 3 keeps the comparison on the cost axis and states the failure-rate discrepancy in the table footnote.

Appendix B. Replay Dataset Schema

The replay dataset (`weclaw_ebeac_log_replay`, version 0.3.0-real-db-export) is archived alongside this paper in a public Zenodo deposit (<https://doi.org/10.5281/zenodo.22839726>; Section 5.5). It contains 196 items exported from the production ExperienceStore snapshot, deduplicated, sorted chronologically by `created_at`, and split 70/30 into train/test by time order.

Top-level fields. `dataset_name` (fixed string), `version`, `notes`, `export_summary` (source databases, rows before/after deduplication, train ratio), and `items`.

Item fields.

Field	Type	Description
<code>event_id</code>	string	Stable identifier derived from the source snapshot and row order
<code>timestamp</code>	string (ISO 8601)	Experience creation time; ordering key for the chronological split
<code>session_id</code>	string	Anonymized session identifier

Field	Type	Description
tool_name	string	Tool associated with the episode
action	string	Tool action (derived from the episode where present)
status	string	Always error in this dataset (replayed failure events)
error_type	string or null	Derived error category (7 distinct values in this snapshot)
fix_pattern	string or null	Ground-truth abstract pattern used for judgment
split	string	train (137 items) or test (59 items)
metadata	object	query (trigger text), diagnosis (raw error text), source_type, tool_names, related_files, hit_count (production retrieval hits), source_db, record_origin

Judgment conventions. A retrieval is counted as a recall@k hit when the ground-truth `fix_pattern` of the test item appears among the top-k retrieved experiences; a false positive is any injected candidate whose pattern does not match (Section 5.2). Because judgment operates at pattern granularity, the reported recall is an upper bound on fix-level accuracy (Section 7.2).